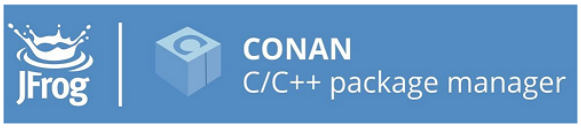


Conan Documentation

Release 1.49.0

The Conan team

Apr 23, 2026

CONTENTS

Conan is a software package manager which is intended for C and C++ developers.

Conan is universal and portable. It works in all operating systems including Windows, Linux, OSX, FreeBSD, Solaris, and others, and it can target any platform, including desktop, server, and cross-building for embedded and bare metal devices. It integrates with other tools like Docker, MinGW, WSL, and with all build systems such as CMake, MSBuild, Makefiles, Meson, SCons. It can even integrate with any proprietary build systems.

Conan is completely free and open source and fully decentralized. It has native integration with JFrog Artifactory, including the free Artifactory Community Edition for Conan, enabling developers to host their own private packages on their own server. The ConanCenter central repository contains hundreds of popular open source libraries packages, with many pre-compiled binaries for mainstream compiler versions.

Conan can manage any number of different binaries for different build configurations, including different architectures, compilers, compiler versions, runtimes, C++ standard library, etc. When binaries are not available for one configuration, they can be built from sources on-demand. Conan can create, upload and download binaries with the same commands and flows on every platform, saving lots of time in development and continuous integration. The binary compatibility can even be configured and customized on a per-package basis.

Conan has a very large and active community, especially in Github repositories and Slack #conan channel. This community also creates and maintains packages in ConanCenter. Conan is used in production by thousands of companies, and consequently, it has a commitment to stability, with no breaking changes across all Conan 1.X versions.

CHAPTER

ONE

INTRODUCTION

Conan is a dependency and package manager for C and C++ languages. It is free and open-source, works in all platforms (Windows, Linux, OSX, FreeBSD, Solaris, etc.), and can be used to develop for all targets including embedded, mobile (iOS, Android), and bare metal. It also integrates with all build systems like CMake, Visual Studio (MSBuild), Makefiles, SCons, etc., including proprietary ones.

It is specifically designed and optimized for accelerating the development and Continuous Integration of C and C++ projects. With full binary management, it can create and reuse any number of different binaries (for different configurations like architectures, compiler versions, etc.) for any number of different versions of a package, using exactly the same process in all platforms. As it is decentralized, it is easy to run your own server to host your own packages and binaries privately, without needing to share them. The free JFrog Artifactory Community Edition (CE) is the recommended Conan server to host your own packages privately under your control.

Conan is mature and stable, with a strong commitment to forward compatibility (non-breaking policy), and has a complete team dedicated full time to its improvement and support. It is backed and used by a great community, from open source contributors and package creators in ConanCenter to thousands of teams and companies using it.

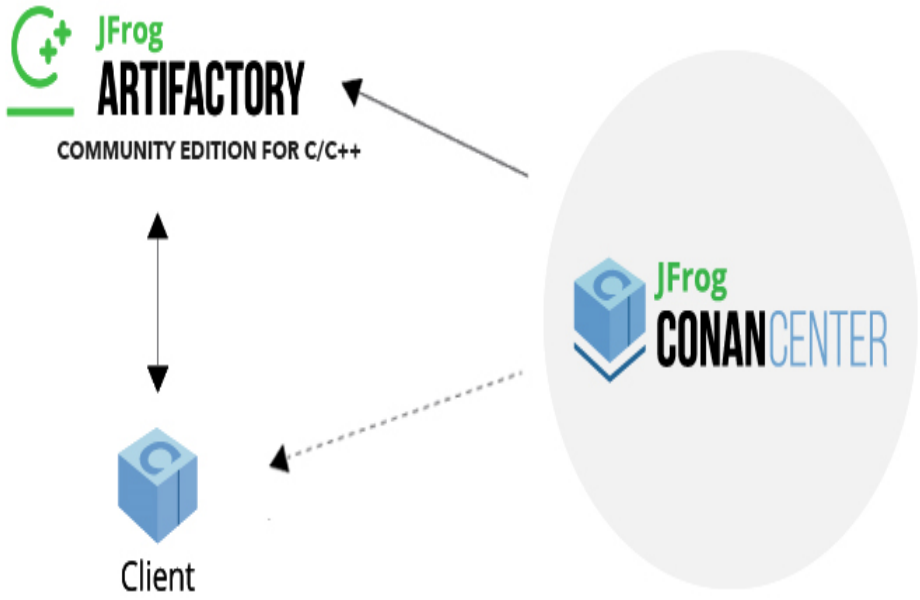
1.1 Open Source

Conan is Free and Open Source, with a permissive MIT license. Check out the source code and issue tracking (for questions and support, reporting bugs and suggesting feature requests and improvements) at https://github.com/conan-io/conan

1.2 Decentralized package manager

Conan is a decentralized package manager with a client-server architecture. This means that clients can fetch packages from, as well as upload packages to, different servers ("remotes"), similar to the "git" push-pull model to/from git remotes.

At a high level, the servers are just storing packages. They do not build nor create the packages. The packages are created by the client, and if binaries are built from sources, that compilation is also done by the client application.



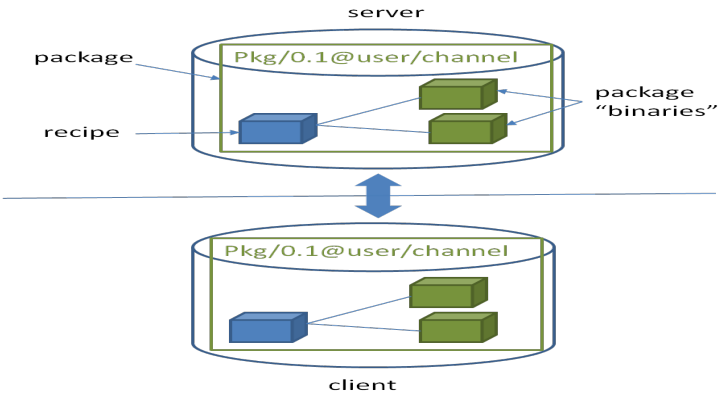
The different applications in the image above are:

- The Conan client: this is a console/terminal command-line application, containing the heavy logic for package creation and consumption. Conan client has a local cache for package storage, and so it allows you to fully create and test packages offline. You can also work offline as long as no new packages are needed from remote servers.
- JFrog Artifactory Community Edition (CE) is the recommended Conan server to host your own packages privately under your control. It is a free community edition of JFrog Artifactory for Conan packages, including a WebUI, multiple auth protocols (LDAP), Virtual and Remote repositories to create advanced topologies, a Rest API, and generic repositories to host any artifact.
- The conan_server is a small server distributed together with the Conan client. It is a simple open-source implementation and provides basic functionality, but no WebUI or other advanced features.
- ConanCenter is a central public repository where the community contributes packages for popular open-source libraries like Boost, Zlib, OpenSSL, Poco, etc.

1.3 Binary management

One of the most powerful features of Conan is that it can create and manage pre-compiled binaries for any possible platform and configuration. By using pre-compiled binaries and avoiding repeated builds from source, it saves significant time for developers and Continuous Integration servers, while also improving the reproducibility and traceability of artifacts.

A package is defined by a "conanfile.py". This is a file that defines the package's dependencies, sources, how to build the binaries from sources, etc. One package "conanfile.py" recipe can generate any arbitrary number of binaries, one for each different platform and configuration: operating system, architecture, compiler, build type, etc. These binaries can be created and uploaded to a server with the same commands in all platforms, having a single source of truth for all packages and not requiring a different solution for every different operating system.



Installation of packages from servers is also very efficient. Only the necessary binaries for the current platform and configuration are downloaded, not all of them. If the compatible binary is not available, the package can be built from sources in the client too.

1.4 All platforms, all build systems and compilers

Conan works on Windows, Linux (Ubuntu, Debian, RedHat, ArchLinux, Raspbian), OSX, FreeBSD, and SunOS, and, as it is portable, it might work in any other platform that can run Python. It can target any existing platform: ranging from bare metal to desktop, mobile, embedded, servers, and cross-building.

Conan works with any build system too. There are built-in integrations to support the most popular ones like CMake, Visual Studio (MSBuild), Autotools and Makefiles, SCons, etc., but it is not a requirement to use any of them. It is not even necessary that all packages use the same build system: each package can use their own build system, and depend on other packages using different build systems. It is also possible to integrate with any build system, including proprietary ones.

Likewise, Conan can manage any compiler and any version. There are default definitions for the most popular ones: gcc, cl.exe, clang, apple-clang, intel, with different configurations of versions, runtimes, C++ standard library, etc. This model is also extensible to any custom configuration.

1.5 Stable

From Conan 1.0 and onwards, there is a commitment to stability, with the goal of not breaking user space while evolving the tool and the platform. This means:

- Moving forward to following minor versions 1.1, 1.2, ..., 1.X should never break existing recipes, packages or command line flows
- If something is breaking, it will be considered a bug and reverted
- Bug fixes will not be considered breaking, recipes and packages relying on the incorrect behavior of such bugs will be considered already broken.
- Only documented features are considered part of the public interface of Conan. Private implementation details, and everything not included in the documentation is subject to change.
- Configuration and automatic tools detection, like the detection of the default profile might be subject to change. Users are encouraged to define their configurations in profiles for repeatability. New installations of Conan might use different configurations.

The compatibility is always considered forward. New APIs, tools, methods, helpers can be added in following 1.X versions. Recipes and packages created with these features will be backwards incompatible with earlier Conan versions.

This means that public repositories, like ConanCenter, assume the use of the latest version of the Conan client, and using an older version may result in failure of packages and recipes created with a newer version of the client.

Conan needs Python 3 to run. It has supported Python 2 until 1 January 2020, when it was officially deprecated by the Python maintainers. From Conan 1.22.0 release, Python 2 support is not guaranteed. See the *deprecation notice* for more details

If you have any question regarding Conan updates, stability, or any clarification about this definition of stability, please report in the documentation issue tracker: https://github.com/conan-io/docs.

1.6 Community

Conan is being used in production by hundreds of companies like Audi, Continental, Plex, Electrolux and Mercedes-Benz and many thousands of developers around the world.

But an essential part of Conan is that many of those users will contribute back, creating an amazing and helpful community:

- The https://github.com/conan-io/conan project has more than 3.5K stars in Github and counts with contributions of nearly 200 different users (this is just the client tool).
- Many other users contribute recipes for ConanCenter via the https://github.com/conan-io/conan-center-index repo, creating packages for popular Open Source libraries.
- More than one thousand of Conan users hang around the CppLang Slack #conan channel, and help responding to questions, discussing problems and approaches..

Have any questions? Please check out our *FAQ section* or .

CHAPTER

TWO

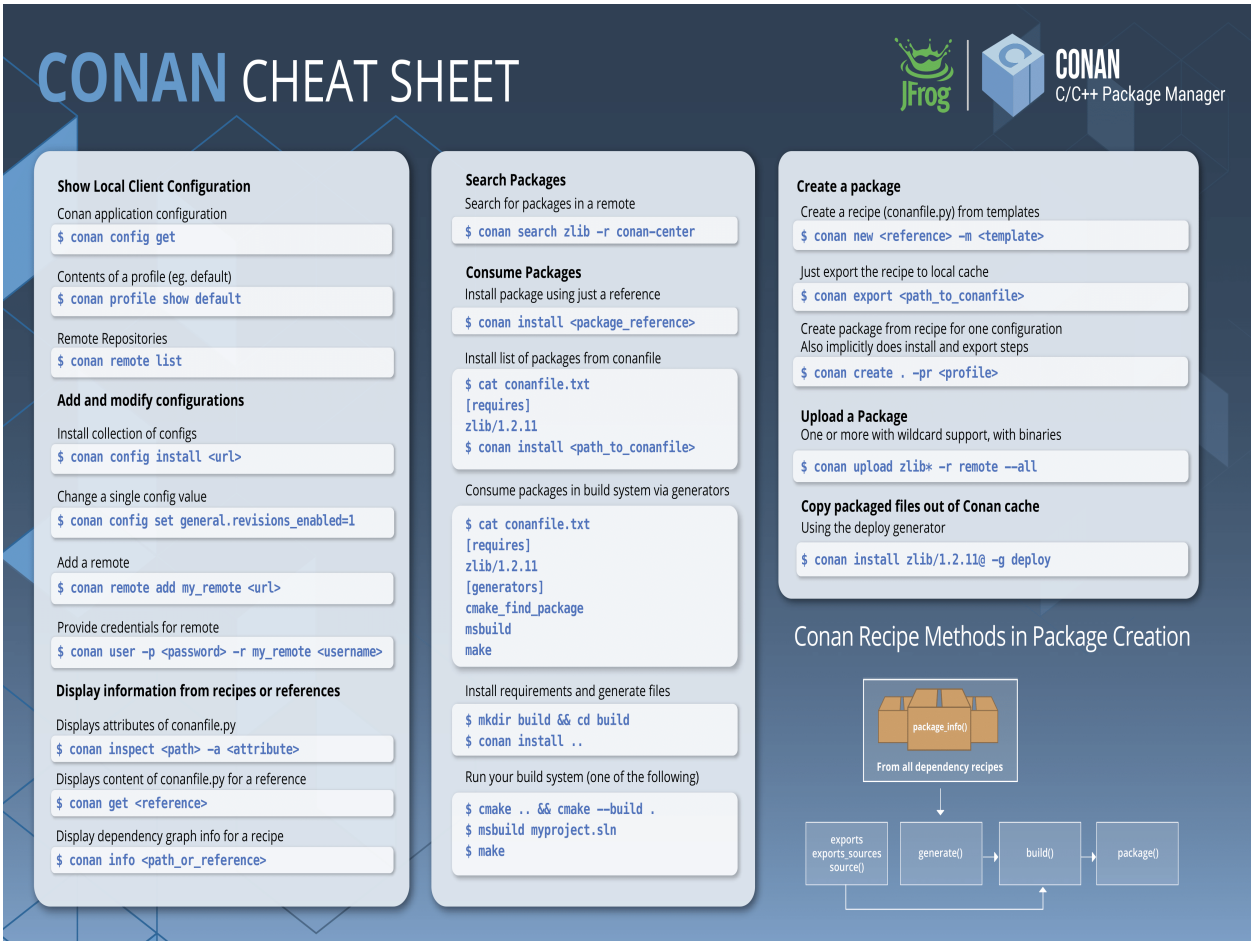
CHEATSHEET

2.1 Single-Page Graphical Format

JFrog has created the following visual cheatsheet for basic Conan commands and concepts which users can print out and use as a handy reference. It is available as both a PDF and PNG.

PDF Format

PNG Format



2.2 Community-Created Format

Community contributors have also created the following extended cheatsheet providing a more narrative and workflow-centric cheatsheet. It is effectively, a single-page summary of other docs pages which community users found most relevant to daily Conan operations.

2.2.1 Setup and configuration

Installation

Conan is available as a Python package, and the recommended way to install is via pip:

```
$ pip install conan
$ pip install conan --upgrade
```

There are other methods of installation available, including standalone installers, which don't require a Python installation.

See Install docs.

Configurations

Configurations contain *hooks*, *profiles*, *remote repositories* and *settings*, making them available for builds once installed. They are installed from a folder, zip, URL or git repo, and the installed items are recorded in ~/.conan/conan.conf.

Install configurations:

```
$ conan config install <item>  # Copy the relevant contents from <item> to the user's ~/.
↪conan directory.

$ conan config install ./my_config.conf
```

Alternatively, copying files and editing conan.conf can be done manually.

Set up configurations:

```
$ conan config init     # Initialize Conan configuration files. If some are already␣
↪present, missing files only are
                        # created
```

Set configuration values:

```
$ conan config set <section>.<config>=<value>

$ conan config set log.level=10
$ conan config set log.print_run_commands=False  # Make conan less verbose
```

Inspect configurations:

```
$ conan config home     # See the Conan home directory

$ conan config get [<section>.<config>]  # Show some or all configuration items

$ conan config get                       # Show the full conan.conf file
$ conan config get log.level             # Show the "level" item in the "log" section
```

See conan config reference.

Profiles

Profiles allow users to set aspects of the build configuration. This includes *settings*, *options*, environment variables and tool requirements. They can be installed into ~/.conan/profiles. They can also be stored in project directories, which can be useful for specific compilation cases, for example cross-compiling.

Profiles are stored in text files with no file extension. An example profile:

```
CROSS_GCC=arm-linux-gnueabihf

include(default)              # Can include other configurations, for example the␣
↪default configuration

[settings]
os=Linux
compiler=gcc
compiler.version=6
compiler.libcxx=libstdc++11
build_type=Release
arch=armv7
os_build=Linux
arch_build=x86_64
OpenSSL:compiler.version=4.8  # Dependency-specific value

[options]
shared=True

[env]                         # Environment variables
CC=$CROSS_GCC-gcc             # Strings can be defined and substituted
CXX=$CROSS_GCC-g++

[tool_requires]               # Requirements for package builds only
cmake/3.16.3                  # Specifying tool requirements here rather than in the␣
↪recipe makes them less binding
```

List profiles:

```
$ conan profile list
```

Show a profile:

```
$ conan profile show <profile>

$ conan profile show default
```

Use profile while executing command (e.g., `conan install` or `conan create`):

```
$ conan <command> . -pr=<profile1> -pr=<profile2>  # Use installed profile name, or file␣
↪path
                                                   # Composable, last -pr wins for␣
↪conflicts
```

See conan profile reference.

Remote repositories

Conan Center is configured by default.

List configured remotes:

```
$ conan remote list
```

Add remote:

```
$ conan remote add <remote ID> <URL of remote repo>
```

See conan remote reference.

2.2.2 Consuming packages

Using packages in an application

1. Write a conanfile.txt. This captures the project configuration:

```
[requires]                      # The Conan packages which are used in the application
boost/1.72.0                    # Versions override versions upstream in the dependency␣
↪graph
poco/1.9.4

[tool_requires]                 # The Conan packages which are used to build the␣
↪application
7zip/16.00

[generators]                    # Generators create build system files that capture the␣
↪dependency information,
cmake                           # as well as configuration information from Conan␣
↪settings and options

[options]                       # Options here override options upstream in the␣
↪dependency graph
boost:shared=True               # Options can be specified on a per-package basis for␣
↪dependencies
poco:shared=True

[imports]                       # Copies files from the cache to the current working␣
↪directory
bin, *.dll -> ./bin             # Copies all .dll files from the packages' bin/ folder to␣
↪the local bin/ folder
```

2. Get dependencies and generate build system files via `conan install`

```
$ conan install . [-o <package>:<option>=<value>]  # Specify options, e.g. shared=True
                  [-s <package>:<setting>=<value>] # Specify settings, e.g. build_
↪type=Debug
                                                   # <package> is optional: if not␣
↪specified, the option/setting
                                                   # applies to all dependencies
```

(continues on next page)

(continued from previous page)

```
                  [-r=<remote ID>]                  # Download dependencies from only the␣
↪specified remote
                  [-g=<generator>]                  # Specify generators at the command␣
↪line
```

3. #include interface files to the Conan packages in the source code
4. Modify the build system to use the files output from the Generator
5. Build the application using the build system

Downloading packages

Download a package, if it isn't already in *the local cache*:

```
$ conan install <package>/<version>@[<user>/<channel>#<revision>]
                [-r=<remote ID>]                                  # Download␣
↪dependencies from only the specified remote


$ conan install .  # Install a package requirement from a conanfile.txt, saved in your␣
↪current directory, with all
                   # options and settings coming from your default profile

$ conan install . -o pkg_name:use_debug_mode=on -s compiler=clang   # As above, but␣
↪override one option and one
                                                                    # setting
```

See conan install reference.

The local cache

The local package cache is located at ~/.conan/data by default (but this is configurable).

Clear packages from cache:

```
$ conan remove "<package>" --force  # <package> can include wildcards

$ conan remove 'boost/*'                      # Remove all versions of Boost
$ conan remove 'mypackage/1.2@user/channel'  # Remove all revisions of mypackage/1.
↪2@user/channel
```

See conan remove reference.

Using packages as standalone applications

Packages can either be copied to the local project folder and run from there, or run directly from the local cache.

In the *conanfile.txt*, this can be done in the [imports] or [generators] section. See below for the relevant generators. In *the package recipe*, this can be done using the `imports()` or `deploy()` methods.

Prepare packages for use via the command line:

```
$ conan install . -g=deploy          # Copy dependencies to current folder
$ conan install . -g=virtualrunenv   # Create shell scripts to activate and deactivate␣
↪environments where you can run
                                     # dependencies from the local cache
```

2.2.3 Searching and introspecting packages

Searching packages

Recipes and binaries can be searched in the local cache or remotes.

List names of packages in local cache:

```
$ conan search              # List names of packages in local cache
```

Show package recipes or builds of a package:

```
$ conan search <package>/<revision>@<user>/<channel>  # Output depends on how much of a␣
↪package reference is given.
                                                      # Wildcards are supported
               [--table=file.html]                    # Save output in an HTML file
               [-r=<remote>]                          # Look in a remote repository␣
↪(default is the local cache)

$ conan search mylib/1.0@user/channel                 # Show all packages of mylib/1.
↪0@user/channel in the local cache
$ conan search "zlib/*" -r=all                        # Show all versions of zlib in all␣
↪remotes
```

Show revisions of a package:

```
$ conan search <package>/<revision>@<user>/<channel> --revisions
```

See conan search reference.

Inspecting packages

Print the package recipe in full:

```
$ conan get <package>/<revision>@<user>/<channel>

$ conan get boost/1.74.0
```

Print attributes of the package recipe:

```
$ conan inspect <package>/<revision>@<user>/<channel>

$ conan inspect boost/1.74.0
```

See conan get and conan inspect reference.

Visualizing dependencies

Show a dependency graph for the package or application:

```
$ conan info . [--graph=file.html]  # Save output in an HTML file
```

See conan info reference.

2.2.4 Creating packages

Package terminology

Each package recipe relates to a single package. However, a package can be built in different ways.

A reference is used to identify packages:

```
<package>/<version>@<user>/<channel>#RREV:PACKAGE_ID#PREV
```

The recipe reference is used to identify a certain version of a recipe:

```
<package>/<version>@<user>/<channel>  # <package> and <version> are defined in the␣
↪recipe; <user> and <channel> are
                                      # defined by the user when exporting the package

lib/1.0@conan/stable
```

The package ID is a SHA-1 hash calculated from the build *options* and *settings* and from dependencies (according to certain *modes*).

See *Revisions* for further details of the recipe revision and package revision (RREV and PREV).

Creating a basic package

Create a template package:

```
$ conan new <package>/<version>@[<user>/<channel>]  # <user>/<channel> is not specified␣
↪in Conan Center, but otherwise they should be
           [-t]                                     # Create a recipe for a basic test␣
↪to verify the package was created successfully
           [-s]                                     # Create a recipe/source template␣
↪for a package with local source code
```

Build a package from a *recipe* and store it in the local cache:

```
$ conan create . <user>/<channel> [-o <package>:<option>=<value>]   # Specify options,␣
↪for example shared=True.
                                  [-s <package>:<setting>=<value>]  # Specify settings,␣
↪for example build_type=Debug.
                                                                    # If <package> is␣
↪not specified, the option and
                                                                    # setting applies to␣
↪all dependencies.
                                  [-pr=<profile name>]              # If -pr is not␣
↪specified, the default profile is used
                                  [--build=missing]                 # Build all␣
↪dependencies if they can't be downloaded
```

See conan new and conan create reference.

The package recipe

A package recipe is a Python class, defined in a file called conanfile.py:

```
class MypackageConan(ConanFile):
    ...                                                 # Various package metadata
    settings = "os", "compiler", "build_type", "arch"  # Defines available settings
    options = {"shared": [True, False]}                 # Defines available options and␣
↪defaults. "shared" is a common
                                                        # option which specifies whether␣
↪a library is static or shared

    default_options = {"shared": False}
    requires = "requiredlib/0.1@user/stable"            # Defines package requirements
    tool_requires = "tool_a/0.2@user/testing"           # Defines requirements that are␣
↪only used when the package is
                                                        # built. These should be build␣
↪and test tools only
    generators = "cmake"                                # Generator for the package:␣
↪specifies which build system
                                                        # type will be generated

    def source(self):                                                   # Obtains the␣
↪source code for the project
        self.run("git clone https://github.com/conan-io/hello.git")  # self.run()␣
↪executes any command in the native shell
        tools.get("https://github.com/conan-io/hello/" +                # tools.get()␣
↪downloads, unzips, and then removes the .zip file
                  "archive/refs/heads/master.zip")                      # The tools module␣
↪contains a lot of helper methods for common
        ...                                                             # tasks, and using␣
↪them is often preferable to using self.run()
                                                                        # See the link␣
↪below for more information

    def build_requirements(self):                                      # Responsible for␣
↪specifying non-trivial build requirements logic
```

(continues on next page)

(continued from previous page)

```
        if self.options.myoption1:                                  # Specify a␣
↪conditional tool requirement
            self.tool_requires("zlib/1.2@user/testing")

    def build(self):                                                # Responsible for␣
↪invoking the build system
        cmake = CMake(self)                                         # Helper classes␣
↪are available for several build systems
        ...

        self.run("bin/unittests")                                   # Run unit tests␣
↪compiled earlier in the build() method

    def package(self):                                              # Responsible for␣
↪capturing build artifacts
        self.copy("*.h", dst="include", src="hello")                # self.copy()␣
↪copies files from the cache to the project folder
        ...

    def package_info(self):                                         # Responsible for␣
↪defining variables that are
                                                                    # passed to package␣
↪consumers, for example
                                                                    # library or␣
↪include directories
        self.cpp_info.libs = ["hello"]                              # The cpp_info␣
↪dictionary contains these variables
        ...

    def requirements(self):                                         # Responsible for␣
↪specifying non-trivial requirements logic
        if self.options.myoption2:                                  # Specify a␣
↪conditional requirement
            self.requires("requiredlib2/0.3@user/stable")

    def package_id(self):                                           # Responsible for␣
↪overriding the way the package
                                                                    # ID is calculated␣
↪from the default, for this package only
        default_package_id_mode = full_version_mode
        if self.settings.compiler.version == "4.9":                # Make compiler␣
↪versions 4.8 and 4.7 compatible
                                                                    # with version4.9:␣
↪i.e., they all result in the same package ID

            for version in ("4.8", "4.7"):
                compatible_pkg = self.info.clone()
                compatible_pkg.settings.compiler.version = version
                self.compatible_packages.append(compatible_pkg)     # The compatible_
↪packages property is used to
                                                                    # define this␣
↪behaviour
```

(continues on next page)

(continued from previous page)

```
    def imports(self):                                          # Copies dependency␣
↪files from the local cache
        ...                                                     # to the project␣
↪directory

    def deploy(self):                                           # Installs the␣
↪project, which can include
        ...                                                     # copying build␣
↪artifacts
```

See tools reference.

Python requires

Python requires allow the re-use of python code across multiple recipes. Complex dependency graphs can be produced, and the *same concepts* apply with python requires as with normal package requirements.

Export a conanfile.py:

```
$ conan export . <user>/<channel>
```

Use the exported conanfile.py:

```
class ConsumerConan(ConanFile):
    python_requires = "<package>/<version>@<user>/<channel>"  # To use functions and␣
↪variables from the exported conanfile.py
    python_requires_extend = "<package>.<base class name>"    # To inherit from a full␣
↪class in the exported conanfile.py

        ...
        self.python_requires["<package>"].module.func()           # To call the method␣
↪func() from the exported conanfile.py
```

See conan export reference.

Hooks

Hooks are recipe methods which are defined globally. They should not affect the built binary. There are `pre` and `post` hooks for many methods in the recipe. Hooks reside in ~/.conan/hooks, and are include in ~/.conan/conan.conf under the [hooks] section.

Install a hook:

```
$ conan config install  # In the directory containing the python script with the hook
```

Specifying build configuration items

Settings

Settings are configuration items which generally apply to all builds of all packages in the dependency tree. *compiler*, *os*, *arch*, and *build_type* (*Release*/*Debug*) are some of the most common.

Available settings are defined in a global settings file: ~/.conan/settings.yml. The settings for a given package are defined in *the package recipe*.

Settings can then be set via *profiles* or via arguments to *conan install* or *conan create*.

Options

Options are configuration items which are generally package-specific.

The available options for a package are defined in *the package recipe*.

Options can then be set via *profiles*, an application's *Conanfile.txt*, or via arguments to *conan install* or *conan create*.

Versioning

Versions

Packages are specified whenever a package is created, and whenever a recipe is consumed via a recipe reference.

Specify ranges:

```
[>min_ver <max_ver] - specify a version range
[*]                 - specify any version
[~maj.min]          - specify any patch in v[maj].[min]
```

The version taken is otherwise the maximum available.

Revisions

Revisions allow changes to a package without increasing the version number or overwriting the existing version number. They are disabled by default.

There are two types of revisions:

- "Recipe Revisions" (RREV) - Revision of the recipe and sources
- "Package Revisions" (PREV) - Revision of a binary package

The recipe revision (RREV) is a SHA-1 hash calculated over the *conan_manifest.txt*, which contains the individual hashes of the *conanfile.py* and all the files exported with *exports* and *exports_sources*. If the *scm* feature is used, Conan can also formulate the recipe revision directly from the version control system. Conan only holds one recipe revision in the local cache. Many recipe revisions can be stored in remote repositories. This helps differentiate between packages that have been changed and built without changing the version number. Recipe revisions can be specified wherever a recipe is consumed. If a recipe revision is not specified, the latest revision is used.

The package revision (PREV) is a SHA-1 hash calculated over the binary contents of the package directory after the build and package steps are completed. Package revisions provide the most precise identification for a built package. They are very rarely used directly by users in commands or configurations, because it's fairly impactical to do so. Instead, they are generally managed by the use of "Lockfiles".

Enable revisions:

```
$ conan config set general.revisions_enabled=True
```

Managing dependencies

Specifying dependencies

Main application dependencies are set in the [requires] section of *Conanfile.txt*.

Package dependencies - normal requirements, tool requirements, conditional requirements - are set in *the package recipe*.

Package ID calculation modes

Conan performs dependency resolution via the calculation of package IDs. A package ID is calculated for a desired dependency, and then Conan searches for that package ID.

The package ID calculation, and therefore the dependency resolution, is affected by the default_package_id_mode and the default_python_requires_id_mode. They determine what exactly affects the calculation: which parts of version numbers; package revisions; immediate or transitive dependencies. This relates to both normal requirements and *Python requires*. By default, only the main version number of direct dependencies are taken into account when calculating the package ID.

These modes can be set in the [general] section of *configurations*, and in *the package recipe*.

Resolving dependency conflicts

Versions defined in the *conanfile.txt* take precedence over versions specified by dependencies. This can be used to resolve conflicts by dictating the use of only one version throughout the whole dependency graph.

Lockfiles

Lockfiles allow a snapshot of a dependency graph used for a build to be taken, and the build to be reproduced exactly at a later time.

Create a lockfile:

```
$ conan lock create <package>/conanfile.py --user=<user> --channel=<channel>
```

Use lockfile during `conan create` or `conan install`:

```
$ conan <command> --lockfile conan.lock
```

See conan lock reference.

Uploading packages to a remote repository

Packages are not uploaded to a remote repository automatically. This needs to be done manually.

```
$ conan upload "<package>" -r <remote ID>  # Wildcards can be specified to upload␣
↪multiple packages
                [--all]                    # Upload all binaries and their recipes␣
↪(recipes only uploaded by default)
                [--confirm]                # Auto-confirm
```

See conan upload reference.

2.2.5 Important points for enterprises

Versioning, revisioning and dependency resolution should be consistent across a company. *Configurations* should be synchronised across all developers, in particular *package id calculation modes*.

In a CI/CD system, use *lockfiles* throughout, so that builds are reproducible.

CHAPTER

THREE

TRAINING COURSES

JFrog has created the JFrog Academy to host a broad range of free online courses surrounding Devops. The Conan team has created the "Conan series" on JFrog Academy, which includes several levels of courses covering both beginner concepts and advanced scenarios.

The courses are completely free and self-paced. They feature interactive exercises which walk users through the running of commands, exploring and editing of important Conan-related files and directories, and quizzes to invoke critical thinking after each section.

For additional information about the Conan training series, see the original blog post announcement here:

- https://blog.conan.io/2020/09/24/New-conan-training-series.html

For the complete list of dedicated Conan courses, see the Conan series page here:

- https://academy.jfrog.com/path/conan

Finally, here is a brief video introducing the series:

CHAPTER

FOUR

INSTALL

Conan can be installed in many Operating Systems. It has been extensively used and tested in Windows, Linux (different distros), OSX, and is also actively used in FreeBSD and Solaris SunOS. There are also several additional operating systems on which it has been reported to work.

There are three ways to install Conan:

1. The preferred and **strongly recommended way to install Conan** is from PyPI, the Python Package Index, using the `pip` command.
2. There are other available installers for different systems, which might come with a bundled python interpreter, so that you don't have to install python first. Note that some of **these installers might have some limitations**, especially those created with pyinstaller (such as Windows exe & Linux deb).
3. Running Conan from sources.

4.1 Install with pip (recommended)

To install Conan using `pip`, you need Python>=3.6 distribution installed on your machine.

Warning: Python 2 has been deprecated on January 1st, 2020 by the Python maintainers and from Conan 1.49 it will not be possible to run Conan with Python 2.7, and at least Python>=3.6 will be required. See *Python 2 Removal Notice* for details.

Install Conan:

```
$ pip install conan
```

Important: Please READ carefully

- Make sure that your **pip** installation matches your **Python>=3.6** version. Lower Python versions will not work.
- In **Linux**, you may need **sudo** permissions to install Conan globally.
- We strongly recommend using **virtualenvs** (virtualenvwrapper works great) for everything related to Python. (check https://virtualenvwrapper.readthedocs.io/en/stable/, or https://pypi.org/project/virtualenvwrapper-win/ in Windows) With Python 3, the built-in module `venv` can also be used instead (check https://docs.python.org/3/library/venv.html). If not using a **virtualenv** it is possible that conan dependencies will conflict with previously existing dependencies, especially if you are using Python for other purposes.
- In **OSX**, especially the latest versions that may have **System Integrity Protection**, pip may fail. Try using virtualenvs, or install with another user `$ pip install --user conan`.

- Some Linux distros, such as Linux Mint, require a restart (shell restart, or logout/system if not enough) after installation, so Conan is found in the path.

4.1.1 Known installation issues with pip

- When Conan is installed with **pip install --user <username>**, usually a new directory is created for it. However, the directory is not appended automatically to the *PATH* and the **conan** commands do not work. This can usually be solved restarting the session of the terminal or running the following command:

```
$ source ~/.profile
```

4.2 Install from brew (OSX)

There is a brew recipe, so in OSX, you can install Conan as follows:

```
$ brew update
$ brew install conan
```

4.3 Install from AUR (Arch Linux)

The easiest way to install Conan on Arch Linux is by using one of the Arch User Repository (AUR) helpers, e.g., **yay**, **aurman**, or **pakku**. For example, the following command installs Conan using `yay`:

```
$ yay -S conan
```

Alternatively, build and install Conan manually using `makepkg` and `pacman` as described in the Arch Wiki. Conan build files can be downloaded from AUR: https://aur.archlinux.org/packages/conan/. Make sure to first install the three Conan dependencies which are also found in AUR:

- python-patch-ng
- python-node-semver
- python-pluginbase

4.4 Install the binaries

Go to the conan website and download the installer for your platform!

Execute the installer. You don't need to install python.

4.5 Initial configuration

Check if Conan is installed correctly. Run the following command in your console:

```
$ conan
```

The response should be similar to:

```
Consumer commands
  install    Installs the requirements specified in a recipe (conanfile.py or conanfile.
↪txt).
  config     Manages Conan configuration.
  get        Gets a file or list a directory of a given reference or package.
  info       Gets information about the dependency graph of a recipe.
  ...
```

Tip: If you are using Bash, there is a bash autocompletion project created by the community for Conan commands: https://gitlab.com/akim.saidani/conan-bashcompletion

4.6 Install from source

You can run Conan directly from source code. First, you need to install Python and pip.

Clone (or download and unzip) the git repository and install it with:

```
# clone folder name matters, to avoid imports issues
$ git clone https://github.com/conan-io/conan.git conan_src
$ cd conan_src
$ python -m pip install -e .
```

Test your `conan` installation.

```
$ conan
```

You should see the Conan commands help.

4.7 Update

If installed via `pip`, Conan can be easily updated:

```
$ pip install conan --upgrade  # Might need sudo or --user
```

If installed via the installers (*.exe*, *.deb*), download the new installer and execute it.

The default *<userhome>/.conan/settings.yml* file, containing the definition of compiler versions, etc., will be upgraded if Conan does not detect local changes, otherwise it will create a *settings.yml.new* with the new settings. If you want to regenerate the settings, you can remove the *settings.yml* file manually and it will be created with the new information the first time it is required.

The upgrade shouldn't affect the installed packages or cache information. If the cache becomes inconsistent somehow, you may want to remove its content by deleting it (*<userhome>/.conan*).

4.8 Python 2 Removal Notice

From version 1.49, Conan will not work with Python 2. This is because security vulnerabilities of Conan dependencies that haven't been addressed in Python 2, so the only alternative moving forward is to finally remove Python 2 suport.

Python 2 was officially declared End Of Life 2 years and a half now, and Conan 1.22 already declared Python 2 as not supported. Extra blockers have been added in previous Conan releases to make everyone aware. Now the security vulnerabilities that are out of our scope, makes impossible to move forward support for Python 2. Please upgrade to Python>=3.6 to continue using Conan>=1.49.

If you have any issue installing Conan, please report in the Conan issue tracker or write us to info@conan.io.

CHAPTER

FIVE

GETTING STARTED

Let's get started with an example: We are going to create an MD5 hash calculator app that uses one of the most popular C++ libraries: Poco.

We'll use CMake as build system in this case but keep in mind that Conan **works with any build system** and is not limited to using CMake.

Make sure you are running the latest Conan version. Read the *Conan update* section to get more information.

5.1 An MD5 hash calculator using the Poco Libraries

Note: The source files to recreate this project are available in the example repository in GitHub. You can skip the manual creation of the folder and sources with this command:

```
$ git clone https://github.com/conan-io/examples.git && cd examples/libraries/poco/md5
```

1. Create the following source file inside a folder. This will be the source file of our application:

Listing 1: **md5.cpp**

```
#include "Poco/MD5Engine.h"
#include "Poco/DigestStream.h"

#include <iostream>

int main(int argc, char** argv){
    Poco::MD5Engine md5;
    Poco::DigestOutputStream ds(md5);
    ds << "abcdefghijklmnopqrstuvwxyz";
    ds.close();
    std::cout << Poco::DigestEngine::digestToHex(md5.digest()) <<␣
↪std::endl;
    return 0;
}
```

2. We know that our application relies on the Poco libraries. Let's look for it in the ConanCenter remote, going to https://conan.io/center, and typing "poco" in the search box. We will see that there are some different versions available:

```
poco/1.8.1
poco/1.9.3
poco/1.9.4
...
```

Note: The Conan client contains a command to search in remote repositories, and we could try **$ conan search poco --remote=conancenter**. You can perfectly use this command to search in your own repositories, but note that at the moment this might timeout in ConanCenter. The infrastructure is being improved to support this command too, but meanwhile using the ConanCenter UI is recommended.

3. We got some interesting references for Poco. Let's inspect the metadata of the 1.9.4 version:

```
$ conan inspect poco/1.9.4
name: poco
version: 1.9.4
url: https://github.com/conan-io/conan-center-index
homepage: https://pocoproject.org
license: BSL-1.0
author: None
description: Modern, powerful open source C++ class libraries for building␣
↪network- and internet-based applications that run on desktop, server,␣
↪mobile and embedded systems.
topics: ('conan', 'poco', 'building', 'networking', 'server', 'mobile',
↪'embedded')
generators: cmake
exports: None
exports_sources: CMakeLists.txt
short_paths: False
apply_env: True
build_policy: None
revision_mode: hash
settings: ('os', 'arch', 'compiler', 'build_type')
options:
    cxx_14: [True, False]
    enable_apacheconnector: [True, False]
    enable_cppparser: [True, False]
    enable_crypto: [True, False]
    [...]
default_options:
    cxx_14: False
    enable_apacheconnector: False
    enable_cppparser: False
    enable_crypto: True
    [...]
```

4. Let's use this `poco/1.9.4` version for our MD5 calculator app, creating a *conanfile.txt* inside our project's folder with the following content:

Listing 2: **conanfile.txt**

```
[requires]
poco/1.9.4
```

(continues on next page)

(continued from previous page)

```
[generators]
cmake
```

In this example we are using CMake to build the project, which is why the `cmake` generator is specified. This generator creates a *conanbuildinfo.cmake* file that defines CMake variables including paths and library names that can be used in our build. Read more about *Generators*.

5. Next step: We are going to install the required dependencies and generate the information for the build system:

Important: If you are using **GCC compiler >= 5.1**, Conan will set the `compiler.libcxx` to the old ABI for backwards compatibility. In the context of this getting started example, this is a bad choice though: Recent gcc versions will compile the example by default with the new ABI and linking will fail without further customization of your cmake configuration. You can avoid this with the following commands:

```
$ conan profile new default --detect  # Generates default profile detecting␣
↪GCC and sets old ABI
$ conan profile update settings.compiler.libcxx=libstdc++11 default  # Sets␣
↪libcxx to C++11 ABI
```

You will find more information in *How to manage the GCC >= 5 ABI*.

```
$ mkdir build && cd build
$ conan install ..
...
Requirements
    bzip2/1.0.8 from 'conancenter' - Downloaded
    expat/2.2.9 from 'conancenter' - Downloaded
    openssl/1.1.1g from 'conancenter' - Downloaded
    pcre/8.41 from 'conancenter' - Downloaded
    poco/1.9.4 from 'conancenter' - Cache
    sqlite3/3.31.1 from 'conancenter' - Downloaded
    zlib/1.2.11 from 'conancenter' - Downloaded
Packages
    bzip2/1.0.8:5be2b7a2110ec8acdbf9a1cea9de5d60747edb34 - Download
    expat/2.2.9:6cc50b139b9c3d27b3e9042d5f5372d327b3a9f7 - Download
    openssl/1.1.1g:6cc50b139b9c3d27b3e9042d5f5372d327b3a9f7 - Download
    pcre/8.41:20fc3dfce989c458ac2372442673140ea8028c06 - Download
    poco/1.9.4:73e83a21ea6817fa9ef0f7d1a86ea923190b0205 - Download
    sqlite3/3.31.1:4559c5d4f09161e1edf374b033b1d6464826db16 - Download
    zlib/1.2.11:6cc50b139b9c3d27b3e9042d5f5372d327b3a9f7 - Download

zlib/1.2.11: Retrieving package f74366f76f700cc6e991285892ad7a23c30e6d47␣
↪from remote 'conancenter'
Downloading conanmanifest.txt completed [0.25k]
Downloading conaninfo.txt completed [0.44k]
Downloading conan_package.tgz completed [83.15k]
Decompressing conan_package.tgz completed [0.00k]
zlib/1.2.11: Package installed f74366f76f700cc6e991285892ad7a23c30e6d47
zlib/1.2.11: Downloaded package revision 0
```

(continues on next page)

(continued from previous page)

```
...
poco/1.9.4: Retrieving package 645aaff0a79e6036c7780360 1e44677556109dd9␣
↪from remote 'conancenter'
Downloading conanmanifest.txt completed [48.75k]
Downloading conaninfo.txt completed [2.44k]
Downloading conan_package.tgz completed [5128.39k]
Decompressing conan_package.tgz completed [0.00k]
poco/1.9.4: Package installed 645aaff0a79e6036c7780360 1e44677556109dd9
poco/1.9.4: Downloaded package revision 0
conanfile.txt: Generator cmake created conanbuildinfo.cmake
conanfile.txt: Generator txt created conanbuildinfo.txt
conanfile.txt: Generated conaninfo.txt
conanfile.txt: Generated graphinfo
```

Conan installed our Poco dependency but also the **transitive dependencies** for it: OpenSSL, zlib, sqlite and others. It has also generated a *conanbuildinfo.cmake* file for our build system.

Warning: There are prebuilt binaries for several mainstream compilers and versions available in Conan Center repository, such as Visual Studio 14, 15, Linux GCC 4.9 and Apple Clang 3.5. Up to >130 different binaries for different configurations can be available in ConanCenter. But if your current configuration is not pre-built in ConanCenter, Conan will raise a "BinaryMissing" error. Please read carefully the error messages. You can build the binary package from sources using **conan install .. --build=missing**, it will succeed if your configuration is supported by the recipe (it is possible that some ConanCenter recipes fail to build for some platforms). You will find more info in the *Building with other configurations* section.

6. Now let's create our build file. To inject the Conan information, include the generated *conanbuildinfo.cmake* file like this:

Listing 3: **CMakeLists.txt**

```
cmake_minimum_required(VERSION 2.8.12)
project(MD5Encrypter)

add_definitions("-std=c++11")

include(${CMAKE_BINARY_DIR}/conanbuildinfo.cmake)
conan_basic_setup()

add_executable(md5 md5.cpp)
target_link_libraries(md5 ${CONAN_LIBS})
```

Note: There are other integrations with CMake, like the `cmake_find_package` generators, that will use the `find_package()` CMake syntax (see *CMake* section).

7. Now we are ready to build and run our MD5 app:

```
(win)
$ cmake .. -G "Visual Studio 16"
$ cmake --build . --config Release
```

(continues on next page)

(continued from previous page)

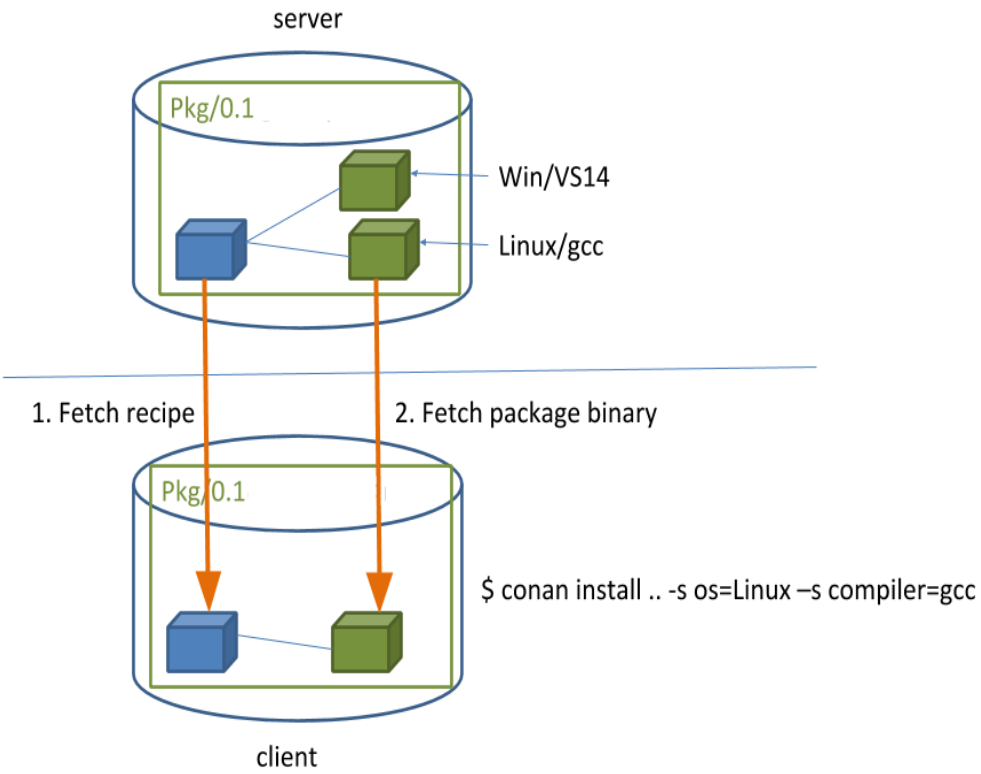
```
(linux, mac)
$ cmake .. -G "Unix Makefiles" -DCMAKE_BUILD_TYPE=Release
$ cmake --build .
...
[100%] Built target md5
$ ./bin/md5
c3fcd3d76192e4007dfb496cca67e13b
```

5.2 Installing Dependencies

The **conan install** command downloads the binary package required for your configuration (detected the first time you ran the command), **together with other (transitively required by Poco) libraries, like OpenSSL and Zlib**. It will also create the *conanbuildinfo.cmake* file in the current directory, in which you can see the CMake variables, and a *conaninfo.txt* in which the settings, requirements and optional information is saved.

Note: Conan generates a *default profile* with your detected settings (OS, compiler, architecture...) and that configuration is printed at the top of every **conan install** command. However, it is strongly recommended to review it and adjust the settings to accurately describe your system as shown in the *Building with other configurations* section.

It is very important to understand the installation process. When the **conan install** command runs, settings specified on the command line or taken from the defaults in *<userhome>/.conan/profiles/default* file are applied.



For example, the command **conan install .. --settings os="Linux" --settings compiler="gcc"**, performs these steps:

- Checks if the package recipe (for `poco/1.9.4` package) exists in the local cache. If we are just starting, the cache is empty.
- Looks for the package recipe in the defined remotes. Conan comes with `conancenter` remote as the default, but can be changed.
- If the recipe exists, the Conan client fetches and stores it in your local Conan cache.
- With the package recipe and the input settings (Linux, GCC), Conan looks for the corresponding binary in the local cache.
- As the binary is not found in the cache, Conan looks for it in the remote and fetches it.
- Finally, it generates an appropriate file for the build system specified in the `[generators]` section.

5.3 Inspecting Dependencies

The retrieved packages are installed to your local user cache (typically *.conan/data*), and can be reused from this location for other projects. This allows to clean your current project and continue working even without network connection. To search for packages in the local cache run:

```
$ conan search "*"
Existing package recipes:

openssl/1.0.2t
poco/1.9.4
zlib/1.2.11
...
```

To inspect the different binary packages of a reference run:

```
$ conan search poco/1.9.4@
Existing packages for recipe poco/1.9.4:

Package_ID: 645aaff0a79e6036c77803601e44677556109dd9
    [options]
        cxx_14: False
        enable_apacheconnector: False
        enable_cppparser: False
        enable_crypto: True
        enable_data: True
...
```

The `@` symbol at the end of the package name is important to search for a specific package. If you don't add the `@`, Conan will interpret the argument as a pattern search and return all the packages that match the `poco/1.9.4` pattern and may have different *user and channel*.

To inspect all your current project's dependencies use the **`conan info`** command by pointing it to the location of the *conanfile.txt* folder:

```
$ conan info ..
conanfile.txt
    ID: db91af4811b080e02ebe5a626f1d256bb90d5223
    BuildID: None
    Requires:
```

(continues on next page)

(continued from previous page)

```
        poco/1.9.4
openssl/1.0.2t
    ID: eb50d18a5a5d59bd0c332464a4c348ab65e353bf
    BuildID: None
    Context: host
    Remote: conancenter=https://center.conan.io
    URL: https://github.com/conan-io/conan-center-index
    Homepage: https://github.com/openssl/openssl
    License: OpenSSL
    Description: A toolkit for the Transport Layer Security (TLS) and Secure Sockets␣
↪Layer (SSL) protocols
    Topics: conan, openssl, ssl, tls, encryption, security
    Recipe: Cache
    Binary: Cache
    Binary remote: conancenter
    Creation date: 2019-11-13 23:14:37
    Required by:
        poco/1.9.4
    Requires:
        zlib/1.2.11
poco/1.9.4
    ID: 645aaff0a79e6036c77803601e44677556109dd9
    BuildID: None
    Context: host
    Remote: conancenter=https://center.conan.io
    URL: https://github.com/conan-io/conan-center-index
    Homepage: https://pocoproject.org
    License: BSL-1.0
    Description: Modern, powerful open source C++ class libraries for building network-␣
↪and internet-based applications that run on desktop, server, mobile and embedded␣
↪systems.
    Topics: conan, poco, building, networking, server, mobile, embedded
    Recipe: Cache
    Binary: Cache
    Binary remote: conancenter
    Creation date: 2020-01-07 17:29:24
    Required by:
        conanfile.txt
    Requires:
        openssl/1.0.2t
zlib/1.2.11
    ID: f74366f76f700cc6e991285892ad7a23c30e6d47
    BuildID: None
    Context: host
    Remote: conancenter=https://center.conan.io
    URL: https://github.com/conan-io/conan-center-index
    Homepage: https://zlib.net
    License: Zlib
    Description: A Massively Spiffy Yet Delicately Unobtrusive Compression Library (Also␣
↪Free, Not to Mention Unencumbered by Patents)
    Recipe: Cache
    Binary: Cache
```

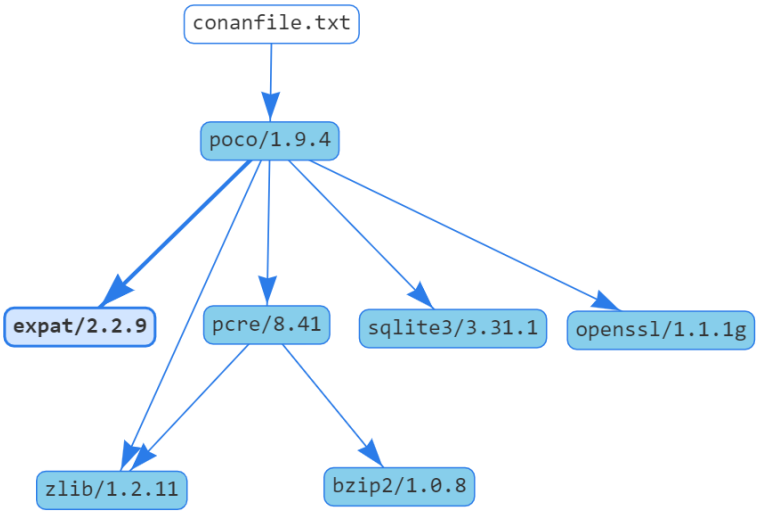
(continues on next page)

(continued from previous page)

```
    Binary remote: conancenter
    Creation date: 2020-01-07 17:01:29
    Required by:
        openssl/1.0.2t
```

Or generate a graph of your dependencies using Dot or HTML formats:

```
$ conan info .. --graph=file.html
$ file.html # or open the file, double-click
```



5.4 Searching Packages

The remote repository where packages are installed from is configured by default in Conan. It is called Conan Center (configured as *conancenter* remote).

If we search for something like `open` in ConanCenter we could find different packages like:

```
openal/1.18.2@bincrafters/stable
openal/1.19.1
opencv/2.4.13.5@conan/stable
opencv/3.4.3@conan/stable
opencv/4.1.1@conan/stable
openexr/2.3.0
openexr/2.3.0@conan/stable
openexr/2.4.0
openjpeg/2.3.0@bincrafters/stable
openjpeg/2.3.1
openjpeg/2.3.1@bincrafters/stable
openssl/1.0.2s
...
```

As you can see, some of the libraries end with a `@` symbol followed by two strings separated by a slash. These fields are the *user and channel* for the Conan package, and they are useful if you want to make specific changes and disambiguate

your modified recipe from the one in the Conan Center or any other remote. These are legacy packages, and the ones without user and channel are the ones strongly recommended to use from ConanCenter.

ConanCenter is the central public repository for Conan packages. You can contribute packages to it in the conan-center-index Github repository. If you want to store your own private packages, you can download the free Artifactory Community Edition (CE) directly from the Conan downloads page.

5.5 Building with other configurations

In this example, we have built our project using the default configuration detected by Conan. This configuration is known as the *default profile*.

A profile needs to be available prior to running commands such as **conan install**. When running the command, your settings are automatically detected (compiler, architecture...) and stored as the default profile. You can edit these settings *~/.conan/profiles/default* or create new profiles with your desired configuration.

For example, if we have a profile with a 32-bit GCC configuration in a file called *gcc_x86*, we can run the following:

```
$ conan install .. --profile=gcc_x86
```

Tip: We strongly recommend using *Profiles* and managing them with *conan config install*.

However, the user can always override the profile settings in the **conan install** command using the **--settings** parameter. As an exercise, try building the 32-bit version of the hash calculator project like this:

```
$ conan install .. --settings arch=x86
```

The above command installs a different package, using the **--settings arch=x86** instead of the one of the default profile used previously. Note you might need to install extra compilers or toolchains in some platforms, as for example, Linux distributions no longer install 32bits toolchains by default.

To use the 32-bit binaries, you will also have to change your project build:

- In Windows, change the CMake invocation to `Visual Studio 14`.
- In Linux, you have to add the `-m32` flag to your `CMakeLists.txt` by running `SET(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -m32")`, and the same applies to `CMAKE_C_FLAGS`, `CMAKE_SHARED_LINK_FLAGS` and `CMAKE_EXE_LINKER_FLAGS`. This can also be done more easily, by automatically using Conan, as we'll show later.
- In macOS, you need to add the definition `-DCMAKE_OSX_ARCHITECTURES=i386`.

Got any doubts? Check our *FAQ*, or join the community in Cpplang Slack `#conan` channel!

CHAPTER

SIX

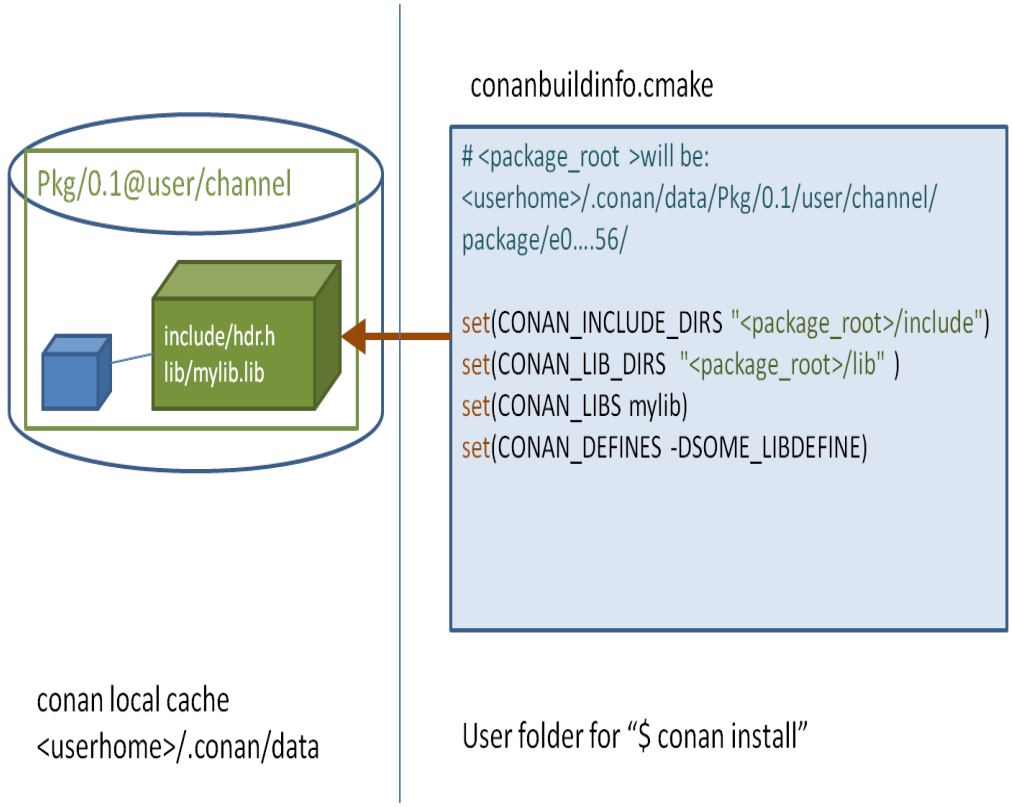
USING PACKAGES

This section shows how to setup your project and manage dependencies (i.e., install existing packages) with Conan.

6.1 Installing dependencies

In *Getting started* we used the **conan install** command to download the **Poco** library and build an example.

If you inspect the `conanbuildinfo.cmake` file that was created when running **conan install**, you can see there that there are many CMake variables declared. For example `CONAN_INCLUDE_DIRS_ZLIB`, that defines the include path to the zlib headers, and `CONAN_INCLUDE_DIRS` that defines include paths for all dependencies headers.



If you check the full path that each of these variables defines, you will see that it points to a folder under your `<userhome>` folder. Together, these folders are the **local cache**. This is where package recipes and binary packages are stored and cached, so they don't have to be retrieved again. You can inspect the **local cache** with **conan search**, and remove packages from it with **conan remove** command.

If you navigate to the folders referenced in `conanbuildinfo.cmake` you will find the headers and libraries for each package.

If you execute a **`conan install poco/1.9.4@`** command in your shell, Conan will download the Poco package and its dependencies (*openssl/1.0.2t* and *zlib/1.2.11*) to your local cache and print information about the folder where they are installed. While you can install each of your dependencies individually like that, the recommended approach for handling dependencies is to use a `conanfile.txt` file. The structure of `conanfile.txt` is described below.

6.1.1 Requires

The required dependencies should be specified in the **[requires]** section. Here is an example:

```
[requires]
mypackage/1.0.0@company/stable
```

Where:

- `mypackage` is the name of the package which is usually the same as the project/library.
- `1.0.0` is the version which usually matches that of the packaged project/library. This can be any string; it does not have to be a number, so, for example, it could indicate if this is a "develop" or "master" version. Packages can be overwritten, so it is also OK to have packages like "nightly" or "weekly", that are regenerated periodically.
- `company` is the owner of this package. It is basically a namespace that allows different users to have their own packages for the same library with the same name.
- `stable` is the channel. Channels provide another way to have different variants of packages for the same library and use them interchangeably. They usually denote the maturity of the package as an arbitrary string such as "stable" or "testing", but they can be used for any purpose such as package revisions (e.g., the library version has not changed, but the package recipe has evolved).

Optional user/channel

> **Warning:** This is an **experimental** feature subject to breaking changes in future releases.

If the package was created and uploaded without specifying the `user` and `channel` you can omit the `user/channel` when specifying a reference:

```
[requires]
packagename/1.2.0
```

Overriding requirements

You can specify multiple requirements and **override** transitive "require's requirements". In our example, Conan installed the Poco package and all its requirements transitively:

- **openssl/1.0.2t**
- **zlib/1.2.11**

> **Tip:** This is a good example of overriding requirements given the importance of keeping the OpenSSL library updated.

Consider that a new release of the OpenSSL library has been released, and a new corresponding Conan package is available. In our example, we do not need to wait until pocoproject (the author) generates a new package of POCO that includes the new OpenSSL library.

We can simply enter the new version in the **[requires]** section:

```
[requires]
poco/1.9.4
openssl/1.0.2u
```

The second line will override the openssl/1.0.2t required by POCO with the currently non-existent **openssl/1.0.2u**.

Another example in which we may want to try some new zlib alpha features: we could replace the zlib requirement with one from another user or channel.

```
[requires]
poco/1.9.4
openssl/1.0.2u
zlib/1.2.11@otheruser/alpha
```

Note: You can use environment variable *CONAN_ERROR_ON_OVERRIDE* to raise an error for every overriden requirement not marked explicitly with the `override` keyword.

6.1.2 Generators

Conan reads the **[generators]** section from `conanfile.txt` and creates files for each generator with all the information needed to link your program with the specified requirements. The generated files are usually temporary, created in build folders and not committed to version control, as they have paths to local folders that will not exist in another machine. Moreover, it is very important to highlight that generated files match the given configuration (Debug/Release, x86/x86_64, etc) specified when running **`conan install`**. If the configuration changes, the files will change accordingly.

For a full list of generators, please refer to the complete *generators* reference.

6.1.3 Options

We have already seen that there are some **settings** that can be specified during installation. For example, **`conan install .. -s build_type=Debug`**. These settings are typically a project-wide configuration defined by the client machine, so they cannot have a default value in the recipe. For example, it doesn't make sense for a package recipe to declare "Visual Studio" as a default compiler because that is something defined by the end consumer, and unlikely to make sense if they are working in Linux.

On the other hand, **options** are intended for package specific configuration that can be set to a default value in the recipe. For example, one package can define that its default linkage is static, and this is the linkage that should be used if consumers don't specify otherwise.

Note: You can see the available options for a package by inspecting the recipe with **`conan get <reference>`** command:

```
$ conan get poco/1.9.4@
```

To see only specific fields of the recipe you can use the **`conan inspect`** command instead:

```
$ conan inspect poco/1.9.4@ -a=options
$ conan inspect poco/1.9.4@ -a=default_options
```

For example, we can modify the previous example to use dynamic linkage instead of the default one, which was static, by editing the **[options]** section in `conanfile.txt`:

```
[requires]
poco/1.9.4

[generators]
cmake

[options]
poco:shared=True # PACKAGE:OPTION=VALUE
openssl:shared=True
```

Install the requirements and compile from the build folder (change the CMake generator if not in Windows):

```
$ conan install ..
$ cmake .. -G "Visual Studio 14 Win64"
$ cmake --build . --config Release
```

As an alternative to defining options in the `conanfile.txt` file, you can specify them directly in the command line:

```
$ conan install .. -o poco:shared=True -o openssl:shared=True
# or even with wildcards, to apply to many packages
$ conan install .. -o *:shared=True
```

Conan will install the binaries of the shared library packages, and the example will link with them. You can again inspect the different binaries installed. For example, **`conan search zlib/1.2.11@`**.

Finally, launch the executable:

```
$ ./bin/md5
```

What happened? It fails because it can't find the shared libraries in the path. Remember that shared libraries are used at runtime, so the operating system, which is running the application, must be able to locate them.

We could inspect the generated executable, and see that it is using the shared libraries. For example, in Linux, we could use the *objdump* tool and see the *Dynamic section*:

```
$ cd bin
$ objdump -p md5
...
Dynamic Section:
 NEEDED               libPocoUtil.so.31
 NEEDED               libPocoXML.so.31
 NEEDED               libPocoJSON.so.31
 NEEDED               libPocoMongoDB.so.31
 NEEDED               libPocoNet.so.31
 NEEDED               libPocoCrypto.so.31
 NEEDED               libPocoData.so.31
 NEEDED               libPocoDataSQLite.so.31
 NEEDED               libPocoZip.so.31
```

(continues on next page)

(continued from previous page)

```
NEEDED               libPocoFoundation.so.31
NEEDED               libpthread.so.0
NEEDED               libdl.so.2
NEEDED               librt.so.1
NEEDED               libssl.so.1.0.0
NEEDED               libcrypto.so.1.0.0
NEEDED               libstdc++.so.6
NEEDED               libm.so.6
NEEDED               libgcc_s.so.1
NEEDED               libc.so.6
```

6.1.4 Imports

There are some differences between shared libraries on Linux (*.so), Windows (*.dll) and MacOS (*.dylib). The shared libraries must be located in a folder where they can be found, either by the linker, or by the OS runtime.

You can add the libraries' folders to the path (LD_LIBRARY_PATH environment variable in Linux, DYLD_LIBRARY_PATH in OSX, or system PATH in Windows), or copy those shared libraries to some system folder where they can be found by the OS. But these operations are only related to the deployment or installation of apps; they are not relevant during development. Conan is intended for developers, so it avoids such manipulation of the OS environment.

In Windows and OSX, the simplest approach is to copy the shared libraries to the executable folder, so they are found by the executable, without having to modify the path.

This is done using the **[imports]** section in `conanfile.txt`.

To demonstrate this, edit the `conanfile.txt` file and paste the following **[imports]** section:

```
[requires]
poco/1.9.4

[generators]
cmake

[options]
poco:shared=True
openssl:shared=True

[imports]
bin, *.dll -> ./bin # Copies all dll files from packages bin folder to my "bin" folder
lib, *.dylib* -> ./bin # Copies all dylib files from packages lib folder to my "bin"␣
↪folder
```

Note: You can explore the package folder in your local cache (~/.conan/data) and see where the shared libraries are. It is common that ***.dll** are copied to **/bin**. The rest of the libraries should be found in the **/lib** folder, however, this is just a convention, and different layouts are possible.

Install the requirements (from the `build` folder), and run the binary again:

```
$ conan install ..
$ ./bin/md5
```

Now look at the `build/bin` folder and verify that the required shared libraries are there.

As you can see, the **[imports]** section is a very generic way to import files from your requirements to your project.

This method can be used for packaging applications and copying the resulting executables to your bin folder, or for copying assets, images, sounds, test static files, etc. Conan is a generic solution for package management, not only for (but focused on) C/C++ libraries.

See also:

To learn more about working with shared libraries, please refer to *Howtos/Manage shared libraries*.

6.2 Using profiles

So far, we have used the default settings stored in `~/.conan/profiles/default` and defined custom values for some of them as command line arguments.

However, in large projects, configurations can get complex, settings can be very different, and we need an easy way to switch between different configurations with different settings, options etc. An easy way to switch between configurations is by using profiles.

A profile file contains a predefined set of `settings`, `options`, `environment variables`, and `tool_requires` specified in the following structure:

```
[settings]
setting=value

[options]
MyLib:shared=True

[env]
env_var=value

[tool_requires]
tool1/0.1@user/channel
tool2/0.1@user/channel, tool3/0.1@user/channel
*: tool4/0.1@user/channel
```

Options allow the use of wildcards letting you apply the same option value to many packages. For example:

```
[options]
*:shared=True
```

Here is an example of a configuration that a profile file may contain:

Listing 1: *clang_3.5*

```
 [settings]
 os=Macos
 arch=x86_64
 compiler=clang
 compiler.version=3.5
 compiler.libcxx=libstdc++11
 build_type=Release
```

(continues on next page)

(continued from previous page)

```
[env]
CC=/usr/bin/clang-3.5
CXX=/usr/bin/clang++-3.5
```

A profile file can be stored in the default profile folder, or anywhere else in your project file structure. To use the configuration specified in a profile file, pass in the file as a command line argument as shown in the example below:

```
$ conan create . demo/testing -pr=clang_3.5
```

Continuing with the example of Poco, instead of passing in a long list of command line arguments, we can define a handy profile that defines them all and pass that to the command line when installing the project dependencies.

A profile to install dependencies as **shared** and in **debug** mode would look like this:

Listing 2: *debug_shared*

```
include(default)

[settings]
build_type=Debug

[options]
poco:shared=True
poco:enable_apacheconnector=False
openssl:shared=True
```

To install dependencies using the profile file, we would use:

```
$ conan install .. -pr=debug_shared
```

We could also create a new profile to use a different compiler version and store that in our project directory. For example:

Listing 3: *poco_clang_3.5*

```
include(clang_3.5)

[options]
poco:shared=True
poco:enable_apacheconnector=False
openssl:shared=True
```

To install dependencies using this new profile, we would use:

```
$ conan install .. -pr=../poco_clang_3.5
```

You can specify multiple profiles in the command line. The applied configuration will be the composition of all the profiles applied in the order they are specified:

```
$ conan install .. -pr=../poco_clang_3.5 -pr=my_build_tool1 -pr=my_build_tool2
```

See also:

Read more about *Profiles* for full reference. There is a Conan command, *conan profile*, that can help inspecting and managing profiles. Profiles can be also shared and installed with the *conan config install* command.

6.3 Workflows

This section summarizes some possible layouts and workflows when using Conan together with other tools as an end-user for installing and consuming existing packages. To create your own packages, please refer to *Creating Packages*.

Whether you are working on a single configuration or a multi configuration project, in both cases, the recommended approach is to have a conanfile (either .py or .txt) at the root of your project.

6.3.1 Single configuration

When working with a single configuration, your conanfile will be quite simple as shown in the examples and tutorials we have used so far in this user guide. For example, in *Getting started*, we showed how you can run the **conan install ..** command inside the *build* folder resulting in the *conaninfo.txt* and *conanbuildinfo.cmake* files being generated there too. Note that the build folder is temporary, so you should exclude it from version control to exclude these temporary files.

Out-of-source builds are also supported. Let's look at a simple example:

```
$ git clone https://github.com/conan-io/examples.git
$ cd libraries/poco
$ conan install ./md5 --install-folder=md5_build
```

This will result in the following layout:

```
md5_build
    conaninfo.txt
    conanbuildinfo.txt
    conanbuildinfo.cmake
md5
    CMakeLists.txt  # If using cmake, but can be Makefile, sln...
    README.md
    conanfile.txt
    md5.cpp
```

Now you are ready to build:

```
$ cd md5_build
$ cmake ../md5 -G "Visual Studio 15 Win64"  # or other generator
$ cmake --build . --config Release
$ ./bin/md5
> c3fcd3d76192e4007dfb496cca67e13b
```

We have created a separate build configuration of the project without affecting the original source directory in any way. The benefit is that we can freely experiment with the configuration: We can clear the build folder and build another. For example, changing the build type to Debug:

```
$ rm -rf *
$ conan install ../md5 -s build_type=Debug
$ cmake ../md5 -G "Visual Studio 15 Win64"
$ cmake --build . --config Debug
$ ./bin/md5
> c3fcd3d76192e4007dfb496cca67e13b
```

6.3.2 Multi configuration

You can also manage different configurations, whether in-source or out of source, and switch between them without having to re-issue the **conan install** command (Note however, that even if you did have to run **conan install** again, since subsequent runs use the same parameters, they would be very fast since packages would already have been installed in the local cache rather than in the project)

```
$ git clone git@github.com:conan-io/examples
$ cd libraries/poco
$ conan install md5 -s build_type=Debug -if md5_build_debug
$ conan install md5 -s build_type=Release -if md5_build_release

$ cd md5_build_debug && cmake ../md5 -G "Visual Studio 15 Win64" && cd ../..
$ cd md5_build_release && cmake ../md5 -G "Visual Studio 15 Win64" && cd ../..
```

Note: You can either use the `--install-folder` or `-if` flags to specify where to generate the output files, or manually create the output directory and navigate to it before executing the **conan install** command.

So the layout will be:

```
md5_build_debug
    conaninfo.txt
    conanbuildinfo.txt
    conanbuildinfo.cmake
    CMakeCache.txt # and other cmake files
md5_build_release
    conaninfo.txt
    conanbuildinfo.txt
    conanbuildinfo.cmake
    CMakeCache.txt # and other cmake files
example-poco-timer
    CMakeLists.txt  # If using cmake, but can be Makefile, sln...
    README.md
    conanfile.txt
    md5.cpp
```

Now you can switch between your build configurations in exactly the same way you do for CMake or other build systems, by moving to the folder in which the build configuration is located, because the Conan configuration files for that build configuration will also be there.

```
$ cd md5_build_debug && cmake --build . --config Debug && cd ../..
$ cd md5_build_release && cmake --build . --config Release && cd ../..
```

Note that the CMake `include()` of your project must be prefixed with the current cmake binary directory, otherwise it will not find the necessary file:

```
include(${CMAKE_BINARY_DIR}/conanbuildinfo.cmake)
conan_basic_setup()
```

See also:

There are two generators, `cmake_multi` and `visual_studio_multi` that could help to avoid the context switch and using Debug and Release configurations simultaneously. Read more about them in *cmake_multi* and *visual_studio_multi*

6.4 Debugging packages

In order to run a debug session and step into the source code, the debugger needs to find the source files (or pdb files ones for Visual Studio), for Mac and Unix system the location of these files is stored inside the library itself.

Usually Conan packages don't include these files and if they do, the path to the local cache might be different: in a typical scenario the packages are generated in a CI machine and the debug session will take place in the developers one, so the path to the sources won't be the same.

The only **rule of thumb is to compile the library we want to debug in the developer machine**, and thanks to Conan this is straightforward:

```
conan install <reference> --build <name> --profile <debug_profile>
```

This command will trigger the build of the library locally in the developer's machine, so the binaries will point to the sources where they are actually located and the debugger will find them.

Note: Keep updated as we are investigating more integrated solutions using *hooks* and for the major IDEs, *Visual Studio* and *CLion*.

CHAPTER

SEVEN

CREATING PACKAGES

This section shows how to create, build and test your packages.

7.1 Getting started

This section introduces how to create your own Conan packages, explain *conanfile.py* recipes and the commands to build packages from sources in your computer.

Important: This is a **tutorial** section. You are encouraged to execute these commands. For this concrete example, you will need **CMake** installed in your path. It is not strictly required by Conan to create packages, you can use other build systems (as VS, Meson, Autotools and even your own) to do that, without any dependency to CMake. Some of the features used in this section are **experimental**, like `CMakeToolchain` or `cmake_layout()`, and they might change in future releases. There are other alternative tools that are stable, please check the *reference section* for more information.

Using the **`conan new`** command will create a "Hello World" C++ library example project for us:

```
$ mkdir hellopkg && cd hellopkg
$ conan new hello/0.1 --template=cmake_lib
File saved: conanfile.py
File saved: CMakeLists.txt
File saved: src/hello.cpp
File saved: src/hello.h
File saved: test_package/conanfile.py
File saved: test_package/CMakeLists.txt
File saved: test_package/src/example.cpp
```

The generated files are:

- **conanfile.py**: On the root folder, there is a *conanfile.py* which is the main recipe file, responsible for defining how the package is built and consumed.
- **CMakeLists.txt**: A simple generic *CMakeLists.txt*, with nothing specific about Conan in it.
- **src** folder: the *src* folder that contains the simple C++ "hello" library.
- (optional) **test_package** folder: contains an *example* application that will require and link with the created package. It is not mandatory, but it is useful to check that our package is correctly created.

Let's have a look at the package recipe *conanfile.py*:

```
from conans import ConanFile
from conan.tools.cmake import CMakeToolchain, CMake, cmake_layout

class HelloConan(ConanFile):
    name = "hello"
    version = "0.1"

    # Binary configuration
    settings = "os", "compiler", "build_type", "arch"
    options = {"shared": [True, False], "fPIC": [True, False]}
    default_options = {"shared": False, "fPIC": True}

    # Sources are located in the same place as this recipe, copy them to the recipe
    exports_sources = "CMakeLists.txt", "src/*"

    def config_options(self):
        if self.settings.os == "Windows":
            del self.options.fPIC

    def layout(self):
        cmake_layout(self)

    def generate(self):
        tc = CMakeToolchain(self)
        tc.generate()

    def build(self):
        cmake = CMake(self)
        cmake.configure()
        cmake.build()

    def package(self):
        cmake = CMake(self)
        cmake.install()

    def package_info(self):
        self.cpp_info.libs = ["hello"]
```

Let's explain a little bit about this recipe:

- The binary configuration is composed by `settings` and `options`. See more in *this section*. When something changes in the configuration, the resulting binary built and packaged will be different:
  - `settings` are project wide configuration, that cannot be defaulted in recipes, like the OS or the architecture.
  - `options` are package specific configuration and can be defaulted in recipes, in this case we have the option of creating the package as a shared or static library, being static the default.
- The `exports_sources` attribute defines which sources are exported together with the recipe, these sources become part of the package recipe (there are other mechanisms that don't do this, will be explained later.
- The `config_options()` method (together with `configure()` one) allows to fine tune the binary configuration model, for example, in Windows there is no `fPIC` option, so it can be removed.
- The `generate()` method prepares the build of the package from source. In this case, it could be simplified to an attribute `generators = "CMakeToolchain"`, but it is left to show this important method. In this case,

the execution of `CMakeToolchain` `generate()` method will create a *conan_toolchain.cmake* file that maps the Conan `settings` and `options` to CMake syntax.

- The `build()` method uses the `CMake` wrapper to call CMake commands, it is a thin layer that will manage to pass in this case the `-DCMAKE_TOOLCHAIN_FILE=conan_toolchain.cmake` argument, plus other possible arguments, like `-DCMAKE_BUILD_TYPE=<config>` if necessary. It will configure the project and build it from source. The actual arguments that will be used are obtained from a generated `CMakePresets.json` file.
- The `package()` method copies artifacts (headers, libs) from the build folder to the final package folder. It can be done with bare "copy" commands, but in this case it is leveraging the already existing CMake install functionality (if the CMakeLists.txt didn't implement it, it is easy to write `self.copy()` commands in this `package()` method.
- Finally, the `package_info()` method defines that consumers must link with a "hello" library when using this package. Other information as include or lib paths can be defined as well. This information is used for files created by generators (as `CMakeDeps`) to be used by consumers. Although this method implies some potential duplication with the build system output (CMake could generate xxx-config.cmake files), it is important to define this, as Conan packages can be consumed by any other build system, not only CMake.

The contents of the `test_package` folder is not critical now for understanding how packages are created, the important bits are:

- `test_package` folder is different from unit or integration tests. These tests are "package" tests, and validate that the package is properly created, and that the package consumers will be able to link against it and reuse it.
- It is a small Conan project itself, it contains its own `conanfile.py`, and its source code including build scripts, that depends on the package being created, and builds and execute a small application that requires the library in the package.
- It doesn't belong to the package. It only exist in the source repository, not in the package.

Let's build the package from sources with the current default configuration (default profile), and then let the `test_package` folder test the package:

```
$ conan create . demo/testing
...
hello/0.1: Hello World Release!
  hello/0.1: _M_X64 defined
  ...
```

If "Hello world Release!" is displayed, it worked. This is what has happened:

- The *conanfile.py* together with the contents of the *src* folder have been copied (exported in Conan terms) to the local Conan cache.
- A new build from source for the `hello/0.1@demo/testing` package starts, calling the `generate()`, `build()` and `package()` methods. This creates the binary package in the Conan cache.
- Moves to the *test_package* folder and executes a `conan install` + `conan build` + `test()` method, to check if the package was correctly created.

We can now validate that the recipe and the package binary are in the cache:

```
$ conan search
Existing package recipes:

hello/0.1@demo/testing

$ conan search hello/0.1@demo/testing
Existing packages for recipe hello/0.1@demo/testing:
```

(continues on next page)

(continued from previous page)

```
Package_ID: 3fb49604f9c2f729b85ba3115852006824e72cab
    [options]
        shared: False
    [settings]
        arch: x86_64
        build_type: Release
        ...
```

The **conan create** command receives the same command line parameters as **conan install** so you can pass to it the same settings and options. If we execute the following lines, we will create new package binaries for those configurations:

```
$ conan create . demo/testing -s build_type=Debug
...
hello/0.1: Hello World Debug!

$ conan create . demo/testing -o hello:shared=True
...
hello/0.1: Hello World Release!
```

These new package binaries will be also stored in the Conan cache, ready to be used by any project in this computer, we can see them with:

```
$ conan search hello/0.1@demo/testing
Existing packages for recipe hello/0.1@demo/testing:

    Package_ID: 127af201a4cdf8111e2e08540525c245c9b3b99e
        [options]
            shared: True
        [settings]
            arch: x86_64
            build_type: Release
            ...
    Package_ID: 3fb49604f9c2f729b85ba3115852006824e72cab
        [options]
            shared: False
        [settings]
            arch: x86_64
            build_type: Release
            ...

    Package_ID: d057732059ea44a47760900cb5e4855d2bea8714
        [options]
            shared: False
        [settings]
            arch: x86_64
            build_type: Debug
            ...
```

Any doubts? Please check out our *FAQ section* or open a Github issue

7.2 Recipe and Sources in a Different Repo

In the previous section, we fetched the sources of our library from an external repository. It is a typical workflow for packaging third party libraries.

There are two different ways to fetch the sources from an external repository:

1. Using the `source()` method as we displayed in the previous section:

```
from conans import ConanFile, CMake, tools

class HelloConan(ConanFile):
    ...

    def source(self):
        self.run("git clone https://github.com/conan-io/hello.git")
        ...
```

You can also use the *tools.Git* class:

```
from conans import ConanFile, CMake, tools

class HelloConan(ConanFile):
    ...

    def source(self):
        git = tools.Git(folder="hello")
        git.clone("https://github.com/conan-io/hello.git", "master")
        ...
```

2. Using the *scm attribute* of the ConanFile:

Warning: This is an **experimental** feature subject to breaking changes in future releases.

```
from conans import ConanFile, CMake, tools

class HelloConan(ConanFile):
     scm = {
        "type": "git",
        "subfolder": "hello",
        "url": "https://github.com/conan-io/hello.git",
        "revision": "master"
     }
    ...
```

Conan will clone the `scm url` and will checkout the `scm revision`. Head to *creating package documentation* to know more details about SCM feature.

The `source()` method will be called after the checkout process, so you can still use it to patch something or retrieve more sources, but it is not necessary in most cases.

7.3 Recipe and Sources in the Same Repo

Sometimes it is more convenient to have the recipe and source code together in the same repository. This is true especially if you are developing and packaging your own library, and not one from a third-party.

There are two different approaches:

- **Using the *exports sources attribute* of the conanfile to** export the source code together with the recipe. This way the recipe is self-contained and will not need to fetch the code from external origins when building from sources. It can be considered a "snapshot" of the source code.
- Using the *scm attribute* of the conanfile to capture the remote and commit of your repository automatically.

7.3.1 Exporting the Sources with the Recipe: `exports_sources`

This could be an appropriate approach if we want the package recipe to live in the same repository as the source code it is packaging.

First, let's get the initial source code and create the basic package recipe:

```
$ conan new hello/0.1 -t -s
```

A *src* folder will be created with the same "hello" source code as in the previous example. You can have a look at it and see that the code is straightforward.

Now let's have a look at *conanfile.py*:

```
from conans import ConanFile, CMake

class HelloConan(ConanFile):
    name = "hello"
    version = "0.1"
    license = "<Put the package license here>"
    url = "<Package recipe repository url here, for issues about the package>"
    description = "<Description of hello here>"
    settings = "os", "compiler", "build_type", "arch"
    options = {"shared": [True, False]}
    default_options = {"shared": False}
    generators = "cmake"
    exports_sources = "src/*"

    def build(self):
        cmake = CMake(self)
        cmake.configure(source_folder="src")
        cmake.build()

        # Explicit way:
        # self.run('cmake "%s/src" %s' % (self.source_folder, cmake.command_line))
        # self.run("cmake --build . %s" % cmake.build_config)

    def package(self):
        self.copy("*.h", dst="include", src="src")
        self.copy("*.lib", dst="lib", keep_path=False)
        self.copy("*.dll", dst="bin", keep_path=False)
```

(continues on next page)

(continued from previous page)

```
        self.copy("*.dylib*", dst="lib", keep_path=False)
        self.copy("*.so", dst="lib", keep_path=False)
        self.copy("*.a", dst="lib", keep_path=False)

    def package_info(self):
        self.cpp_info.libs = ["hello"]
```

There are two important changes:

- Added the `exports_sources` field, indicating to Conan to copy all the files from the local *src* folder into the package recipe.
- Removed the `source()` method, since it is no longer necessary to retrieve external sources.

Also, you can notice the two CMake lines:

```
include(${CMAKE_BINARY_DIR}/conanbuildinfo.cmake)
conan_basic_setup()
```

They are not added in the package recipe, as they can be directly added to the *src/CMakeLists.txt* file.

And simply create the package for user and channel **demo/testing** as described previously:

```
$ conan create . demo/testing
...
hello/0.1@demo/testing test package: Running test()
Hello world Release!
```

7.3.2 Capturing the Remote and Commit: `scm`

> **Warning:** This is an **experimental** feature subject to breaking changes in future releases. Although this is an experimental feature, the use of the feature using `scm_to_conandata` is considered stable.

You can use the *scm attribute* with the `url` and `revision` field set to `auto`. When you export the recipe (or when **`conan create`** is called) the exported recipe will capture the remote and commit of the local repository:

```
import os
from conans import ConanFile, CMake, tools

class HelloConan(ConanFile):
    scm = {
        "type": "git",  # Use "type": "svn", if local repo is managed using SVN
        "subfolder": "hello",
        "url": "auto",
        "revision": "auto",
        "password": os.environ.get("SECRET", None)
    }
    ...
```

You can commit and push the *conanfile.py* to your origin repository, which will always preserve the `auto` values. When the file is exported to the Conan local cache (except you have uncommitted changes, read below), these data will be

stored in the *conanfile.py* itself (Conan will modify the file) or in a special file *conandata.yml* that will be stored together with the recipe, depending on the value of the configuration parameter *scm_to_conandata*.

- If the `scm_to_conandata` is not activated (default behavior in Conan v1.x) Conan will store a modified version of the *conanfile.py* with the values of the fields in plain text:

```
    import os
    from conans import ConanFile, CMake, tools

    class HelloConan(ConanFile):
        scm = {
            "type": "git",
            "subfolder": "hello",
            "url": "https://github.com/conan-io/hello.git",
            "revision": "437676e15da7090a1368255097f51b1a470905a0",
            "password": "MY_SECRET"
        }
        ...
```

  So when you *upload the recipe* to a Conan remote, the recipe will contain the "resolved" URL and commit.

- If `scm_to_conandata` is activated, the value of these fields (except `username` and `password`) will be stored in the *conandata.yml* file that will be automatically exported with the recipe.

Whichever option you choose, the data resolved will be asigned by Conan to the corresponding field when the recipe file is loaded, and they will be available for all the methods defined in the recipe. Also, if building the package from sources, Conan will fetch the code in the captured url/commit before running the method `source()` in the recipe (if defined).

As SCM attributes are evaluated in the local directory context (see *scm attribute*), you can write more complex functions to retrieve the proper values, this source *conanfile.py* will be valid too:

```
import os
from conans import ConanFile, CMake, tools

def get_remote_url():
     """ Get remote url regardless of the cloned directory """
     here = os.path.dirname(__file__)
     svn = tools.SVN(here)
     return svn.get_remote_url()

class HelloConan(ConanFile):
     scm = {
        "type": "svn",
        "subfolder": "hello",
        "url": get_remote_url(),
        "revision": "auto"
     }
    ...
```

Tip: When doing a **`conan create`** or **`conan export`**, Conan will capture the sources of the local scm project folder in the local cache.

This allows building packages making changes to the source code without the need of committing them and pushing them to the remote repository. This convenient to speed up the development of your packages when cloning from a local repository.

So, if you are using the `scm` feature, with some `auto` field for *url* and/or *revision* and you have uncommitted changes in your repository a warning message will be printed:

```
$ conan export . hello/0.1@demo/testing

 hello/0.1@demo/testing: WARN: There are uncommitted changes, skipping the replacement␣
↪of 'scm.url'
 and 'scm.revision' auto fields. Use --ignore-dirty to force it.
 The 'conan upload' command will prevent uploading recipes with 'auto' values in these␣
↪fields.
```

As the warning message explains, the `auto` fields won't be replaced unless you specify `--ignore-dirty`, and by default, the **conan upload** will block the upload of the recipe. This prevents recipes to be uploaded with incorrect scm values exported. You can use **conan upload --force** to force uploading the recipe with the `auto` values unreplaced.

7.4 Packaging Existing Binaries

There are specific scenarios in which it is necessary to create packages from existing binaries, for example from 3rd parties or binaries previously built by another process or team that are not using Conan. Under these circumstances building from sources is not what you want. You should package the local files in the following situations:

- When you cannot build the packages from sources (when only pre-built binaries are available).
- When you are developing your package locally and you want to export the built artifacts to the local cache. As you don't want to rebuild again (clean copy) your artifacts, you don't want to call **conan create**. This method will keep your build cache if you are using an IDE or calling locally to the **conan build** command.

7.4.1 Packaging Pre-built Binaries

Running the `build()` method, when the files you want to package are local, results in no added value as the files copied from the user folder cannot be reproduced. For this scenario, run **conan export-pkg** command directly.

A Conan recipe is still required, but is very simple and will only include the package meta information. A basic recipe can be created with the **conan new** command:

```
$ conan new hello/0.1 --bare
```

This will create and store the following package recipe in the local cache:

```
class HelloConan(ConanFile):
    name = "hello"
    version = "0.1"
    settings = "os", "compiler", "build_type", "arch"

    def package(self):
        self.copy("*")

    def package_info(self):
        self.cpp_info.libs = self.collect_libs()
```

The provided `package_info()` method scans the package files to provide end-users with the name of the libraries to link to. This method can be further customized to provide additional build flags (typically dependent on the settings).

The default `package_info()` applies as follows: it defines headers in the "include" folder, libraries in the "lib" folder, and binaries in the "bin" folder. A different package layout can be defined in the `package_info()` method.

This package recipe can be also extended to provide support for more configurations (for example, adding options: shared/static, or using different settings), adding dependencies (`requires`), and more.

Based on the above, We can assume that our current directory contains a *lib* folder with a number binaries for this "hello" library *libhello.a*, compatible for example with Windows MinGW (gcc) version 4.9:

```
$ conan export-pkg . hello/0.1@myuser/testing  -s os=Windows -s compiler=gcc -s compiler.
↪version=4.9 ...
```

Having a *test_package* folder is still highly recommended for testing the package locally before upload. As we don't want to build the package from the sources, the flow would be:

```
$ conan new hello/0.1 --bare --test
# customize test_package project
# customize package recipe if necessary
$ cd my/path/to/binaries
$ conan export-pkg PATH/TO/conanfile.py hello/0.1@myuser/testing  -s os=Windows -s␣
↪compiler=gcc -s compiler.version=4.9 ...
$ conan test PATH/TO/test_package/conanfile.py hello/0.1@myuser/testing -s os=Windows -s␣
↪compiler=gcc -s ...
```

The last two steps can be repeated for any number of configurations.

7.4.2 Downloading and Packaging Pre-built Binaries

In this scenario, creating a complete Conan recipe, with the detailed retrieval of the binaries could be the preferred method, because it is reproducible, and the original binaries might be traced. Follow our sample recipe for this purpose:

```
class HelloConan(ConanFile):
    name = "hello"
    version = "0.1"
    settings = "os", "compiler", "build_type", "arch"

    def build(self):
        if self.settings.os == "Windows" and self.settings.compiler == "Visual Studio":
            url = ("https://<someurl>/downloads/hello_binary%s_%s.zip"
                   % (str(self.settings.compiler.version), str(self.settings.build_
↪type)))
        elif ...:
            url = ...
        else:
            raise Exception("Binary does not exist for these settings")
        tools.get(url)

    def package(self):
        self.copy("*") # assume package as-is, but you can also copy specific files or␣
↪rearrange

    def package_info(self):  # still very useful for package consumers
        self.cpp_info.libs = ["hello"]
```

Typically, pre-compiled binaries come for different configurations, so the only task that the `build()` method has to implement is to map the `settings` to the different URLs.

Note:

- This is a standard Conan package even if the binaries are being retrieved from elsewhere. The **recommended approach** is to use **`conan create`**, and include a small consuming project in addition to the above recipe, to test locally and then proceed to upload the Conan package with the binaries to the Conan remote with **`conan upload`**.
- The same building policies apply. Having a recipe fails if no Conan packages are created, and the **`--build`** argument is not defined. A typical approach for this kind of packages could be to define a **`build_policy="missing"`**, especially if the URLs are also under the team control. If they are external (on the internet), it could be better to create the packages and store them on your own Conan server, so that the builds do not rely on third party URL being available.

7.5 Understanding Packaging

7.5.1 Creating and Testing Packages Manually

The previous **create** approach using *test_package* subfolder, is not strictly necessary, though **very strongly recommended**. If we didn't want to use the *test_package* functionality, we could just write our recipe ourselves or use the **`conan new`** command without the **`-t`**. command line argument.

```
$ mkdir mypkg && cd mypkg
$ conan new hello/0.1
```

This will create just the *conanfile.py* recipe file. Now we can create our package:

```
$ conan create . demo/testing
```

This is equivalent to:

```
$ conan export . demo/testing
$ conan install hello/0.1@demo/testing --build=hello
```

Once the package is created, it can be consumed like any other package, by adding `hello/0.1@demo/testing` to a project *conanfile.txt* or *conanfile.py* requirements and running:

```
$ conan install .
# build and run your project to ensure the package works
```

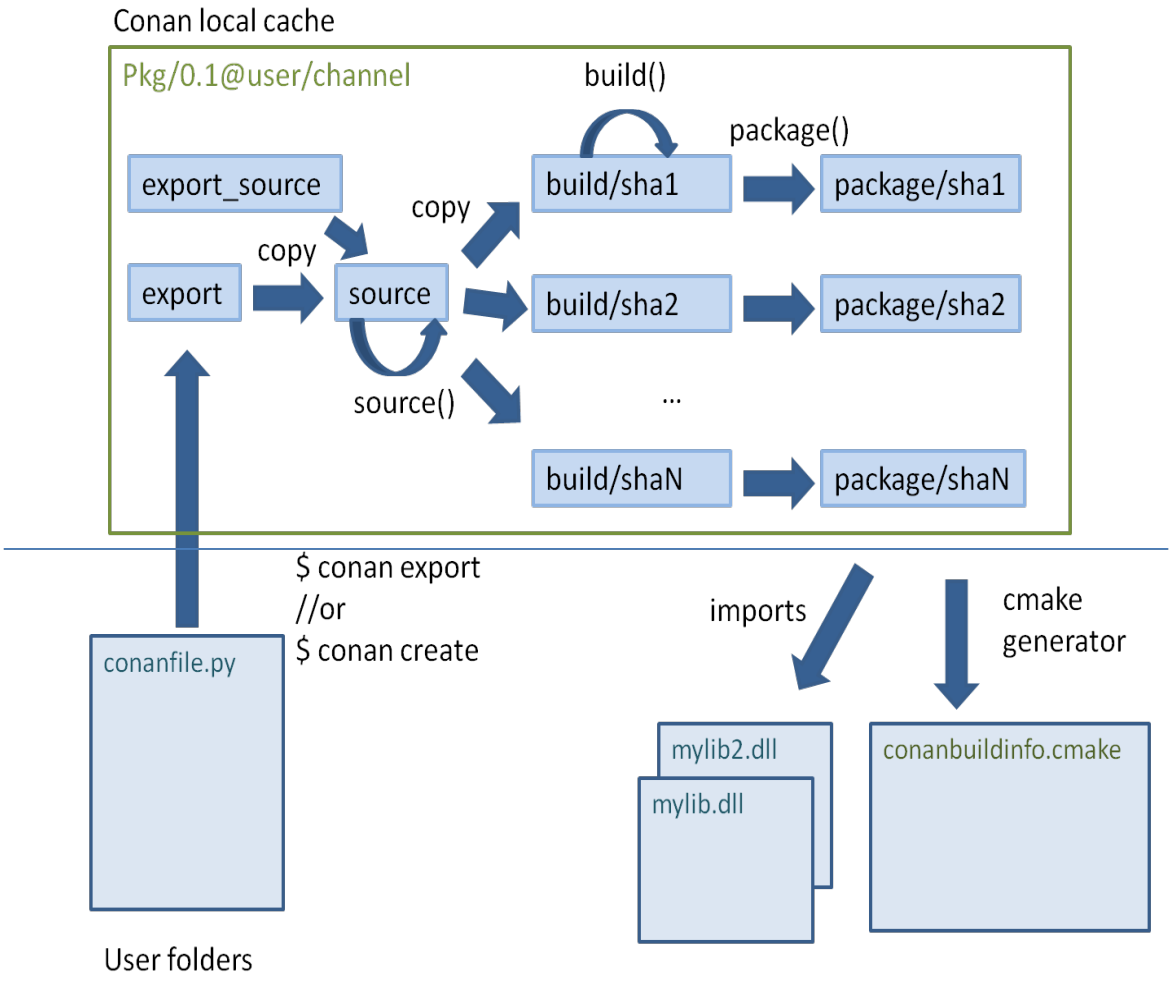
7.5.2 Package Creation Process

It is very useful for package creators and Conan users in general to understand the flow for creating a package inside the conan local cache, and all about its layout.

Each package recipe contains five important folders in the **local cache**:

- **export**: The folder in which the package recipe is stored.
- **export_source**: The folder in which code copied with the recipe `exports_sources` attribute is stored.
- **source**: The folder in which the source code for building from sources is stored.
- **build**: The folder in which the actual compilation of sources is done. There will typically be one subfolder for each different binary configuration
- **package**: The folder in which the final package artifacts are stored. There will be one subfolder for each different binary configuration

The *source* and *build* folders only exist when the packages have been built from sources.



The process starts when a package is "exported", via the **`conan export`** command or more typically, with the **`conan create`** command. The *conanfile.py* and files specified by the `exports_sources` field are copied from the user space to the **local cache**.

The *export* and *export_source* files are copied to the *source* folder, and then the `source()` method is executed (if it

exists). Note that there is only one source folder for all the binary packages. If when generating the code, there is source code that varies for the different configurations, it cannot be generated using the `source()` method, but rather needs to be generated using the `build()` method.

Then, for each different configuration of settings and options, a package ID will be computed in the form of a SHA-1 hash for this configuration. Sources will be copied to the *build/hashXXX* folder, and the `build()` method will be triggered.

After that, the `package()` method will be called to copy artifacts from the *build/hashXXX* folder to the *package/hashXXX* folder.

Finally, the `package_info()` methods of all dependencies will be called and gathered so you can generate files for the consumer build system, as the *conanbuildinfo.cmake* for the `cmake` generator. Also the `imports` feature will copy artifacts from the local cache into user space if specified.

Any doubts? Please check out our *FAQ section* or .

7.6 Defining Package ABI Compatibility

Each package recipe can generate *N* binary packages from it, depending on these three items: `settings`, `options` and `requires`.

When any of the *settings* of a package recipe changes, it will reference a different binary:

```
class MyLibConanPackage(ConanFile):
    name = "mylib"
    version = "1.0"
    settings = "os", "arch", "compiler", "build_type"
```

When this package is installed by a *conanfile.txt*, another package *conanfile.py*, or directly:

```
$ conan install mylib/1.0@user/channel -s arch=x86_64 -s ...
```

The process is:

1. Conan gets the user input settings and options. Those settings and options can come from the command line, profiles or from the values cached in the latest **`conan install`** execution.
2. Conan retrieves the `mylib/1.0@user/channel` recipe, reads the `settings` attribute, and assigns the necessary values.
3. With the current package values for `settings` (also `options` and `requires`), it will compute a SHA1 hash that will serve as the binary package ID, e.g., `c6d75a933080ca17eb7f076813e7fb21aaa740f2`.
4. Conan will try to find the `c6d75...` binary package. If it exists, it will be retrieved. If it cannot be found, it will fail and indicate that it can be built from sources using **`conan install --build`**.

If the package is installed again using different settings, for example, on a 32-bit architecture:

```
$ conan install mylib/1.0@user/channel -s arch=x86 -s ...
```

The process will be repeated with a different generated package ID, because the `arch` setting will have a different value. The same applies to different compilers, compiler versions, build types. When generating multiple binaries - a separate ID is generated for each configuration.

When developers using the package use the same settings as one of those uploaded binaries, the computed package ID will be identical causing the binary to be retrieved and reused without the need of rebuilding it from the sources.

The `options` behavior is very similar. The main difference is that `options` can be more easily defined at the package level and they can be defaulted. Check the *options* reference.

Note this simple scenario of a **header-only** library. The package does not need to be built, and it will not have any ABI issues at all. The recipe for such a package will be to generate a single binary package, no more. This is easily achieved by not declaring `settings` nor `options` in the recipe as follows:

```
class MyLibConanPackage(ConanFile):
    name = "mylib"
    version = "1.0"
    # no settings defined!
```

No matter the settings are defined by the users, including the compiler or version, the package settings and options will always be the same (left empty) and they will hash to the same binary package ID. That package will typically contain just the header files.

What happens if we have a library that can be built with GCC 4.8 and will preserve the ABI compatibility with GCC 4.9? (This kind of compatibility is easier to achieve for example for pure C libraries).

Although it could be argued that it is worth rebuilding with 4.9 too -to get fixes and performance improvements-. Let's suppose that we don't want to create 2 different binaries, but just a single built with GCC 4.8 which also needs to be compatible for GCC 4.9 installations.

7.6.1 Defining a Custom package_id()

The default `package_id()` uses the `settings` and `options` directly as defined, and assumes the semantic versioning for dependencies is defined in `requires`.

This `package_id()` method can be overridden to control the package ID generation. Within the `package_id()`, we have access to the `self.info` object, which is hashed to compute the binary ID and contains:

- **self.info.settings**: Contains all the declared settings, always as string values. We can access/modify the settings, e.g., `self.info.settings.compiler.version`.
- **self.info.options**: Contains all the declared options, always as string values too, e.g., `self.info.options.shared`.

Initially this `info` object contains the original settings and options, but they can be changed without constraints to any other string value.

For example, if you are sure your package ABI compatibility is fine for GCC versions > 4.5 and < 5.0, you could do the following:

```
from conans import ConanFile, CMake, tools
from conans.model.version import Version

class PkgConan(ConanFile):
    name = "pkg"
    version = "1.0"
    settings = "compiler", "build_type"

    def package_id(self):
        v = Version(str(self.settings.compiler.version))
        if self.settings.compiler == "gcc" and (v >= "4.5" and v < "5.0"):
            self.info.settings.compiler.version = "GCC version between 4.5 and 5.0"
```

We have set the `self.info.settings.compiler.version` with an arbitrary string, the value of which is not important (could be any string). The only important thing is that it is the same for any GCC version between 4.5 and 5.0. For all those versions, the compiler version will always be hashed to the same ID.

Let's try and check that it works properly when installing the package for GCC 4.5:

```
$ conan create . pkg/1.0@myuser/mychannel -s compiler=gcc -s compiler.version=4.5 ...

Requirements
    pkg/1.0@myuser/mychannel from local
Packages
    pkg/1.0@myuser/mychannel:af044f9619574eceb8e1cca737a64bdad88246ad
...
```

We can see that the computed package ID is `af04...46ad` (not real). What happens if we specify GCC 4.6?

```
$ conan install pkg/1.0@myuser/mychannel -s compiler=gcc -s compiler.version=4.6 ...

Requirements
    pkg/1.0@myuser/mychannel from local
Packages
    pkg/1.0@myuser/mychannel:af044f9619574eceb8e1cca737a64bdad88246ad
```

The required package has the same result again `af04...46ad`. Now we can try using GCC 4.4 (< 4.5):

```
$ conan install pkg/1.0@myuser/mychannel -s compiler=gcc -s compiler.version=4.4 ...

Requirements
    pkg/1.0@myuser/mychannel from local
Packages
    pkg/1.0@myuser/mychannel:7d02dc01581029782b59dcc8c9783a73ab3c22dd
```

The computed package ID is different which means that we need a different binary package for GCC 4.4.

The same way we have adjusted the `self.info.settings`, we could set the `self.info.options` values if needed. If you want to make packages independent on `build_type` removing the `build_type` from the package settings in the `package_id()` will work for OSX and Linux. However when building with Visual studio the `compiler.runtime` field will change based on the `build_type` value so in that case you will also want to delete the compiler runtime field like so:

```
def package_id(self):
    if self.settings.os in ["Windows","WindowsStore"] and self.settings.compiler ==
↪"Visual Studio":
        del self.info.settings.build_type
        del self.info.settings.compiler.runtime
```

See also:

Check *package_id()* to see the available helper methods and change its behavior for things like:

- Recipes packaging **header only** libraries.
- Adjusting **Visual Studio toolsets** compatibility.

7.6.2 Compatible packages

Warning: This is an **experimental** feature subject to breaking changes in future releases.

The above approach defined 1 package ID for different input configurations. For example, all `gcc` versions in the range `(v >= "4.5" and v < "5.0")` will have exactly the same package ID, no matter what was the gcc version used to build it. It worked like an information erasure, once the binary is built, it is not possible to know which gcc was used to build it.

But it is possible to define compatible binaries that have different package IDs. For instance, it is possible to have a different binary for each `gcc` version, so the `gcc 4.8` package will be a different one with a different package ID than the `gcc 4.9` one, and still define that you can use the `gcc 4.8` package when building with `gcc 4.9`.

We can define an ordered list of compatible packages, that will be checked in order if the package ID that our profile defines is not available. Let's see it with an example:

Lets say that we are building with a profile of `gcc 4.9`. But for a given package we want to fallback to binaries built with `gcc 4.8` or `gcc 4.7` if we cannot find a binary built with `gcc 4.9`. That can be defined as:

```
from conans import ConanFile

class Pkg(ConanFile):
    settings = "os", "compiler", "arch", "build_type"

    def package_id(self):
        if self.settings.compiler == "gcc" and self.settings.compiler.version == "4.9":
            for version in ("4.8", "4.7"):
                compatible_pkg = self.info.clone()
                compatible_pkg.settings.compiler.version = version
                self.compatible_packages.append(compatible_pkg)
```

Note that if the input configuration is `gcc 4.8`, it will not try to fallback to binaries of `gcc 4.7` as the condition is not met.

The `self.info.clone()` method copies the values of `settings`, `options` and `requires` from the current instance of the recipe so they can be modified to model the compatibility.

It is the responsibility of the developer to guarantee that such binaries are indeed compatible. For example in:

```
from conans import ConanFile
class Pkg(ConanFile):
    options = {"optimized": [1, 2, 3]}
    default_options = {"optimized": 1}
    def package_id(self):
        for optimized in range(int(self.options.optimized), 0, -1):
            compatible_pkg = self.info.clone()
            compatible_pkg.options.optimized = optimized
            self.compatible_packages.append(compatible_pkg)
```

This recipe defines that the binaries are compatible with binaries of itself built with a lower optimization value. It can have up to 3 different binaries, one for each different value of `optimized` option. The `package_id()` defines that a binary built with `optimized=1` can be perfectly linked and will run even if someone defines `optimized=2`, or `optimized=3` in their configuration. But a binary built with `optimized=2` will not be considered if the requested one is `optimized=1`.

The binary should be interchangeable at all effects. This also applies to other usages of that configuration. If this example used the `optimized` option to conditionally require different dependencies, that will not be taken into account. The `package_id()` step is processed after the whole dependency graph has been built, so it is not possible to define how dependencies are resolved based on this compatibility model, it only applies to use-cases where the binaries can be *interchanged*.

Note: Compatible packages are a match for a binary in the dependency graph. When a compatible package is found, the **`--build=missing`** build policy will **not** build from sources that package.

Check the *Compatible Compilers* section to see another example of how to take benefit of compatible packages.

7.6.3 Compatible Compilers

Some compilers make use of a base compiler to operate, for example, the `intel` compiler uses the `Visual Studio` compiler in Windows environments and `gcc` in Linux environments.

The `intel` compiler is declared this way in the *settings.yml*:

```
intel:
    version: ["11", "12", "13", "14", "15", "16", "17", "18", "19"]
    base:
        gcc:
            <<: *gcc
            threads: [None]
            exception: [None]
        Visual Studio:
            <<: *visual_studio
```

Remember, you can *extend Conan* to support other compilers.

You can use the `package_id()` method to define the compatibility between the packages generated by the `base` compiler and the `parent` one. You can use the following helpers together with the *compatible packages* feature to:

- Consume native `Visual Studio` packages when the input compiler in the profile is `intel` (if no `intel` package is available).
- The opposite, consume an `intel` compiler package when a consumer profile specifies `Visual Studio` as the input compiler (if no `Visual Studio` package is available).
- `base_compatible()`: This function will transform the settings used to calculate the package ID into the "base" compiler.

  ```
  def package_id(self):

      if self.settings.compiler == "intel":
          p = self.info.clone()
          p.base_compatible()
          self.compatible_packages.append(p)
  ```

  Using the above `package_id()` method, if a consumer specifies a profile with a intel profile (**-s compiler=="intel"**) and there is no binary available, it will resolve to a Visual Studio package ID corresponding to the base compiler.

- `parent_compatible(compiler="compiler", version="version")`: This function transforms the settings of a compiler into the settings of a parent one using the specified one as the base compiler. As the details

of the "parent" compatible cannot be guessed, you have to provide them as **keyword args** to the function. The "compiler" argument is mandatory, the rest of keyword arguments will be used to initialize the `info.settings.compiler.XXX` objects to calculate the correct package ID.

```
def package_id(self):

   if self.settings.compiler == "Visual Studio":
      compatible_pkg = self.info.clone()
      compatible_pkg.parent_compatible(compiler="intel", version=16)
      self.compatible_packages.append(compatible_pkg)
```

In this case, for a consumer specifying Visual Studio compiler, if no package is found, it will search for an "intel" package for the version 16.

Take into account that you can use also these helpers without the "compatible packages" feature:

```
def package_id(self):

   if self.settings.compiler == "Visual Studio":
      self.info.parent_compatible(compiler="intel", version=16)
```

In the above example, we will transform the package ID of the `Visual Studio` package to be the same as the `intel 16`, but you won't be able to differentiate the packages built with `intel` with the ones built by `Visual Studio` because both will have the same package ID, and that is not always desirable.

7.6.4 Dependency Issues

Let's define a simple scenario whereby there are two packages: `my_other_lib/2.0` and `my_lib/1.0` which depends on `my_other_lib/2.0`. Let's assume that their recipes and binaries have already been created and uploaded to a Conan remote.

Now, a new release for `my_other_lib/2.1` is released with an improved recipe and new binaries. The `my_lib/1.0` is modified and is required to be upgraded to `my_other_lib/2.1`.

Note: This scenario will be the same in the case that a consuming project of `my_lib/1.0` defines a dependency to `my_other_lib/2.1`, which takes precedence over the existing project in `my_lib/1.0`.

The question is: **Is it necessary to build new `my_lib/1.0` binary packages?** or are the existing packages still valid?

The answer: **It depends**.

Let's assume that both packages are compiled as static libraries and that the API exposed by `my_other_lib` to `my_lib/1.0` through the public headers, has not changed at all. In this case, it is not required to build new binaries for `my_lib/1.0` because the final consumer will link against both `my_lib/1.0` and `my_other_lib/2.1`.

On the other hand, it could happen that the API exposed by **my_other_lib** in the public headers has changed, but without affecting the `my_lib/1.0` binary for any reason (like changes consisting on new functions not used by **my_lib**). The same reasoning would apply if **MyOtherLib** was only the header.

But what if a header file of `my_other_lib` -named *myadd.h*- has changed from `2.0` to `2.1`:

Listing 1: *myadd.h* header file in version 2.0

```
int addition (int a, int b) { return a - b; }
```

Listing 2: *myadd.h* header file in version 2.1

```
int addition (int a, int b) { return a + b; }
```

And the `addition()` function is called from the compiled *.cpp* files of `my_lib/1.0`?

Then, **a new binary for my_lib/1.0 is required to be built for the new dependency version**. Otherwise it will maintain the old, buggy `addition()` version. Even in the case that `my_lib/1.0` doesn't have any change in its code lines neither in the recipe, the resulting binary rebuilding `my_lib` requires `my_other_lib/2.1` and the package to be different.

7.6.5 Using package_id() for Package Dependencies

The `self.info` object has also a `requires` object. It is a dictionary containing the necessary information for each requirement, all direct and transitive dependencies. For example, `self.info.requires["my_other_lib"]` is a `RequirementInfo` object.

- Each `RequirementInfo` has the following *read only* reference fields:
  - `full_name`: Full require's name, e.g., **my_other_lib**
  - `full_version`: Full require's version, e.g., **1.2**
  - `full_user`: Full require's user, e.g., **my_user**
  - `full_channel`: Full require's channel, e.g., **stable**
  - `full_package_id`: Full require's package ID, e.g., **c6d75a...**
- The following fields are used in the `package_id()` evaluation:
  - `name`: By default same value as full_name, e.g., **my_other_lib**.
  - `version`: By default the major version representation of the `full_version`. E.g., **1.Y** for a **1.2** `full_version` field and **1.Y.Z** for a **1.2.3** `full_version` field.
  - `user`: By default `None` (doesn't affect the package ID).
  - `channel`: By default `None` (doesn't affect the package ID).
  - `package_id`: By default `None` (doesn't affect the package ID).

When defining a package ID for model dependencies, it is necessary to take into account two factors:

- The versioning schema followed by our requirements (semver?, custom?).
- The type of library being built or reused (shared (*.so*, *.dll*, *.dylib*), static).

Versioning Schema

By default Conan assumes semver compatibility. For example, if a version changes from minor **2.0** to **2.1**, Conan will assume that the API is compatible (headers not changing), and that it is not necessary to build a new binary for it. This also applies to patches, whereby changing from **2.1.10** to **2.1.11** doesn't require a re-build.

If it is necessary to change the default behavior, the applied versioning schema can be customized within the `package_id()` method:

```
from conans import ConanFile, CMake, tools
from conans.model.version import Version

class PkgConan(ConanFile):
    name = "my_lib"
    version = "1.0"
    settings = "os", "compiler", "build_type", "arch"
    requires = "my_other_lib/2.0@lasote/stable"

    def package_id(self):
        myotherlib = self.info.requires["my_other_lib"]

        # Any change in the MyOtherLib version will change current Package ID
        myotherlib.version = myotherlib.full_version

        # Changes in major and minor versions will change the Package ID but
        # only a MyOtherLib patch won't. E.g., from 1.2.3 to 1.2.89 won't change.
        myotherlib.version = myotherlib.full_version.minor()
```

Besides `version`, there are additional helpers that can be used to determine whether the **channel** and **user** of one dependency also affects the binary package, or even the required package ID can change your own package ID.

You can determine if the following variables within any requirement change the ID of your binary package using the following modes:

Modes / Variables	name	version	user	channel	package_id	RREV	PREV
`semver_direct_mode()`	Yes	Yes, only > 1.0.0 (e.g., **1**.2.Z+b102)	No	No	No	No	No
`semver_mode()`	Yes	Yes, only > 1.0.0 (e.g., **1**.2.Z+b102)	No	No	No	No	No
`major_mode()`	Yes	Yes (e.g., **1**.2.Z+b102)	No	No	No	No	No
`minor_mode()`	Yes	Yes (e.g., **1.2**.Z+b102)	No	No	No	No	No
`patch_mode()`	Yes	Yes (e.g., **1.2.3**+b102)	No	No	No	No	No
`base_mode()`	Yes	Yes (e.g., **1.7**+b102)	No	No	No	No	No
`full_version_mode()`	Yes	Yes (e.g., **1.2.3+b102**)	No	No	No	No	No
`full_recipe_mode()`	Yes	Yes (e.g., **1.2.3+b102**)	Yes	Yes	No	No	No
`full_package_mode()`	Yes	Yes (e.g., **1.2.3+b102**)	Yes	Yes	Yes	No	No
`unrelated_mode()`	No	No	No	No	No	No	No
`recipe_revision_mode`	Yes	Yes	Yes	Yes	Yes	Yes	No
`package_revision_mod`	Yes	Yes	Yes	Yes	Yes	Yes	Yes

All the modes can be applied to all dependencies, or to individual ones:

```
def package_id(self):
    # apply semver_mode for all the dependencies of the package
    self.info.requires.semver_mode()
    # use semver_mode just for MyOtherLib
    self.info.requires["MyOtherLib"].semver_mode()
```

- `semver_direct_mode()`: This is the default mode. It uses `semver_mode()` for direct dependencies (first level dependencies, directly declared by the package) and `unrelated_mode()` for indirect, transitive dependencies of the package. It assumes that the binary will be affected by the direct dependencies, which they will already

encode how their transitive dependencies affect them. This might not always be true, as explained above, and that is the reason it is possible to customize it.

In this mode, if the package depends on "MyLib", which transitively depends on "MyOtherLib", the mode means:

```
my_lib/1.2.3@user/testing       => my_lib/1.Y.Z
my_other_lib/2.3.4@user/testing  =>
```

So the direct dependencies are mapped to the major version only. Changing its channel, or using version `my_lib/1.4.5` will still produce `my_lib/1.Y.Z` and thus the same package-id. The indirect, transitive dependency doesn't affect the package-id at all.

Important: **Known-bug**: Package ID mode `semver_direct_mode` takes into account the options of transitive requirements. It means that modifying the options of any transitive requirement will modify the computed package ID, and also adding/removing a transitive requirement will modify the computed package ID (this happens even if the added/removed requirement doesn't have any option).

- `semver_mode()`: In this mode, only a major release version (starting from **1.0.0**) changes the package ID. Every version change prior to 1.0.0 changes the package ID, but only major changes after 1.0.0 will be applied.

```
def package_id(self):
  self.info.requires["my_other_lib"].semver_mode()
```

This results in:

```
my_lib/1.2.3@user/testing       => my_lib/1.Y.Z
my_other_lib/2.3.4@user/testing  => my_other_lib/2.Y.Z
```

In this mode, versions starting with `0` are considered unstable and mapped to the full version:

```
my_lib/0.2.3@user/testing       => my_lib/0.2.3
my_other_lib/0.3.4@user/testing  => my_other_lib/0.3.4
```

- `major_mode()`: Any change in the major release version (starting from **0.0.0**) changes the package ID.

```
def package_id(self):
  self.info.requires["MyOtherLib"].major_mode()
```

This mode is basically the same as `semver_mode`, but the only difference is that major versions `0.Y.Z`, which are considered unstable by semver, are still mapped to only the major, dropping the minor and patch parts.

- `minor_mode()`: Any change in major or minor (not patch nor build) version of the required dependency changes the package ID.

```
def package_id(self):
    self.info.requires["my_other_lib"].minor_mode()
```

- `patch_mode()`: Any changes to major, minor or patch (not build) versions of the required dependency change the package ID.

```
def package_id(self):
    self.info.requires["my_other_lib"].patch_mode()
```

- `base_mode()`: Any changes to the base of the version (not build) of the required dependency changes the package ID. Note that in the case of semver notation this may produce the same result as `patch_mode()`, but it is actually intended to dismiss the build part of the version even without strict semver.

```
def package_id(self):
    self.info.requires["my_other_lib"].base_mode()
```

- `full_version_mode()`: Any changes to the version of the required dependency changes the package ID.

```
def package_id(self):
  self.info.requires["my_other_lib"].full_version_mode()
```

```
my_other_lib/1.3.4-a4+b3@user/testing  => my_other_lib/1.3.4-a4+b3
```

- `full_recipe_mode()`: Any change in the reference of the requirement (user & channel too) changes the package ID.

```
def package_id(self):
    self.info.requires["my_other_lib"].full_recipe_mode()
```

  This keeps the whole dependency reference, except the package-id of the dependency.

```
my_other_lib/1.3.4-a4+b3@user/testing  => my_other_lib/1.3.4-a4+b3@user/testing
```

- `full_package_mode()`: Any change in the required version, user, channel or package ID changes the package ID.

```
def package_id(self):
    self.info.requires["my_other_lib"].full_package_mode()
```

  Any change to the dependency, including its binary package-id, will in turn produce a new package-id for the consumer package.

```
MyOtherLib/1.3.4-a4+b3@user/testing:73b..fa56  => MyOtherLib/1.3.4-a4+b3@user/
↪testing:73b..fa56
```

- `unrelated_mode()`: Requirements do not change the package ID.

```
def package_id(self):
    self.info.requires["MyOtherLib"].unrelated_mode()
```

- `recipe_revision_mode()`: The full reference and the package ID of the dependencies, *pkg/version@user/channel#RREV:pkg_id* (including the recipe revision), will be taken into account to compute the consumer package ID

```
mypkg/1.3.4@user/testing#RREV1:73b..fa56#PREV1  => mypkg/1.3.4-a4+b3@user/testing
↪#RREV1
```

```
def package_id(self):
    self.info.requires["mypkg"].recipe_revision_mode()
```

- `package_revision_mode()`: The full package reference *pkg/version@user/channel#RREV:ID#PREV* of the dependencies, including the recipe revision, the binary package ID and the package revision will be taken into account to compute the consumer package ID

  This is the most strict mode. Any change in the upstream will produce new consumers package IDs, becoming a fully deterministic binary model.

```
# The full reference of the dependency package binary will be used as-is
mypkg/1.3.4@user/testing#RREV1:73b..fa56#PREV1  => mypkg/1.3.4@user/testing
↪#RREV1:73b..fa56#PREV1
```

```
def package_id(self):
    self.info.requires["mypkg"].package_revision_mode()
```

Note: Version ranges are not used to calculate the `package_id` only the resolved version in the graph is used

You can also adjust the individual properties manually:

```
def package_id(self):
    myotherlib = self.info.requires["MyOtherLib"]

    # Same as myotherlib.semver_mode()
    myotherlib.name = myotherlib.full_name
    myotherlib.version = myotherlib.full_version.stable()  # major(), minor(), patch(),␣
↪base, build
    myotherlib.user = myotherlib.channel = myotherlib.package_id = None

    # Only the channel (and the name) matters
    myotherlib.name = myotherlib.full_name
    myotherlib.user = myotherlib.package_id = myotherlib.version = None
    myotherlib.channel = myotherlib.full_channel
```

The result of the `package_id()` is the package ID hash, but the details can be checked in the generated *conaninfo.txt* file. The `[requires]`, `[options]` and `[settings]` are taken into account when generating the SHA1 hash for the package ID, while the `[full_xxxx]` fields show the complete reference information.

The default behavior produces a *conaninfo.txt* that looks like:

```
[requires]
MyOtherLib/2.Y.Z

[full_requires]
MyOtherLib/2.2@demo/testing:73bce3fd7eb82b2eabc19fe11317d37da81afa56
```

Changing the default package-id mode

It is possible to change the default `semver_direct_mode` package-id mode, in the *conan.conf* file:

Listing 3: *conan.conf* configuration file

```
[general]
default_package_id_mode=full_package_mode
```

Possible values are the names of the above methods: `full_recipe_mode`, `semver_mode`, etc.

Note: The `default_package_id_mode` is a global configuration. It will change how all the package-ids are computed, for all packages. It is impossible to mix different `default_package_id_mode` values. The same

`default_package_id_mode` must be used in all clients, servers, CI, etc., and it cannot be changed without rebuilding all packages.

Note that the default package-id mode is the mode that is used when the package is initialized and **before** `package_id()` method is called. You can still define `full_package_mode` as default in *conan.conf*, but if a recipe declare that it is header-only, with:

```
def package_id(self):
  self.info.header_only() # clears requires, but also settings if existing
  # or if there are no settings/options, this would be equivalent
  self.info.requires.clear() # or self.info.requires.unrelated_mode()
```

That would still be executed, changing the "default" behavior, and leading to a package that only generates 1 package-id for all possible configurations and versions of dependencies.

Remember that *conan.conf* can be shared and installed with *conan config install*.

Take into account that you can combine the *compatible packages* with the package-id modes.

For example, if you are generating binary packages with the default `recipe_revision_mode`, but you want these packages to be consumed from a client with a different mode activated, you can create a compatible package transforming the mode to `recipe_revision_mode` so the package generated with the `recipe_revision_mode` can be resolved if no package for the default mode is found:

```
from conans import ConanFile

class Pkg(ConanFile):
    ...

    def package_id(self):
        p = self.info.clone()
        p.requires.recipe_revision_mode()
        self.compatible_packages.append(p)
```

Enabling full transitivity in package_id modes

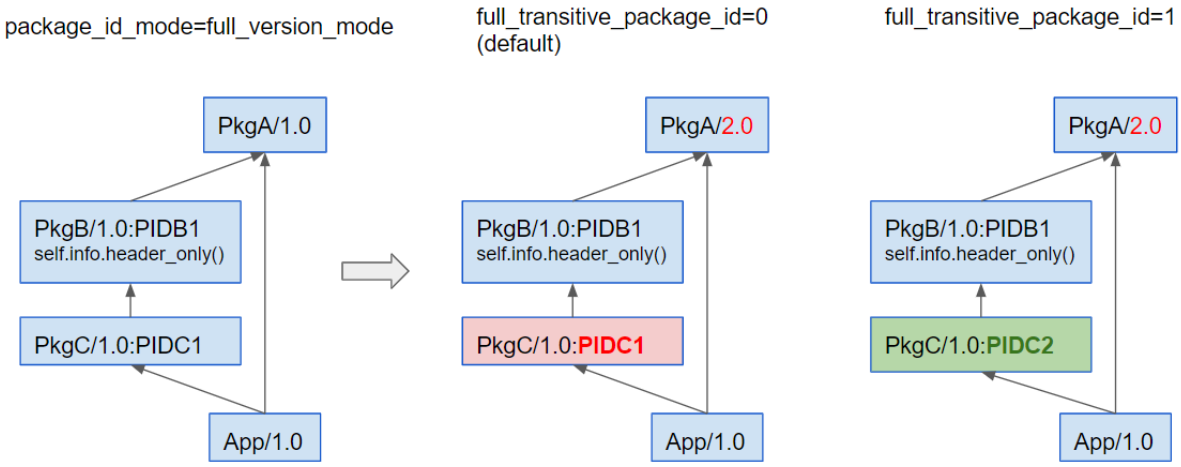
> **Warning:** This will become the default behavior in the future (Conan 2.0). It is recommended to activate it when possible (it might require rebuilding some packages, as their package IDs will change)

When a package declares in its `package_id()` method that it is not affected by its dependencies, that will propagate down to the indirect consumers of that package. There are several ways this can be done, `self.info.header_only()`, `self.info.requires.clear()`, `self.info.requires.remove["dep"]` and `self.info.requires.unrelated_mode()`, for example.

Let's assume for the discussion that it is a header only library, using the `self.info.header_only()` helper. This header only package has a single dependency, which is a static library. Then, downstream consumers of the header only library that uses a package mode different from the default, should be also affected by the upstream transitivity dependency. Lets say that we have the following scenario:

- `app/1.0` depends on `pkgc/1.0` and `pkga/1.0`
- `pkgc/1.0` depends only on `pkgb/1.0`
- `pkgb/1.0` depends on `pkga/1.0`, and defines `self.info.header_only()` in its `package_id()`

- We are using `full_version_mode`
- Now we create a new `pkga/2.0` that has some changes in its header, that would require to rebuild `pkgc/1.0` against it.
- `app/1.0` now depends on `pkgc/1.0` and `pkga/2.0`



With the default behavior, the header only `pkgb` is isolating `pkgc` from the upstream changes effects. The package-id `PIDC1` we get for `pkgc/1.0` is exactly the same when depending on `pkga/1.0` and `pkga/2.0`.

If we want to have the `full_version_mode` to be fully transitive, irrespective of the local package-id modes of the packages, we can configure it in the *conan.conf* section. To summarize, you can activate the `general.full_transitive_package_id` configuration (`$ conan config set general.full_transitive_package_id=1`).

If we do this, then `pkgc/1.0` will compute 2 different package-ids, one for `pkga/1.0` (`PIDC1`) and the other to link with `pkga/2.0` (`PIDC2`).

Library Types: Shared, Static, Header-only

Let's see some examples, corresponding to common scenarios:

- `my_lib/1.0` is a shared library that links with a static library `my_other_lib/2.0` package. When a new `my_other_lib/2.1` version is released: Do I need to create a new binary for `my_lib/1.0` to link with it?

  Yes, always, as the implementation is embedded in the `my_lib/1.0` shared library. If we always want to rebuild our library, even if the channel changes (we assume a channel change could mean a source code change):

```
def package_id(self):
    # Any change in the my_other_lib version, user or
    # channel or Package ID will affect our package ID
    self.info.requires["my_other_lib"].full_package_mode()
```

- `my_lib/1.0` is a shared library, requiring another shared library `my_other_lib/2.0` package. When a new `my_other_lib/2.1` version is released: Do I need to create a new binary for `my_lib/1.0` to link with it?

  It depends. If the public headers have not changed at all, it is not necessary. Actually it might be necessary to consider transitive dependencies that are shared among the public headers, how they are linked and if they cross the frontiers of the API, it might also lead to incompatibilities. If the public headers have changed, it would

depend on what changes and how are they used in `my_lib/1.0`. Adding new methods to the public headers will have no impact, but changing the implementation of some functions that will be inlined when compiled from `my_lib/1.0` will definitely require re-building. For this case, it could make sense to have this configuration:

```
def package_id(self):
    # Any change in the my_other_lib version, user or channel
    # or Package ID will affect our package ID
    self.info.requires["my_other_lib"].full_package_mode()

    # Or any change in the my_other_lib version, user or
    # channel will affect our package ID
    self.info.requires["my_other_lib"].full_recipe_mode()
```

- `my_lib/1.0` is a header-only library, linking with any kind (header, static, shared) of library in `my_other_lib/2.0` package. When a new `my_other_lib/2.1` version is released: Do I need to create a new binary for `my_lib/1.0` to link with it?

  Never. The package should always be the same as there are no settings, no options, and in any way a dependency can affect a binary, because there is no such binary. The default behavior should be changed to:

```
def package_id(self):
    self.info.requires.clear()
```

- `my_lib/1.0` is a static library linking to a header only library in `my_other_lib/2.0` package. When a new `my_other_lib/2.1` version is released: Do I need to create a new binary for `my_lib/1.0` to link with it? It could happen that the `my_other_lib` headers are strictly used in some `my_lib` headers, which are not compiled, but transitively included. But in general, it is more likely that `my_other_lib` headers are used in `MyLib` implementation files, so every change in them should imply a new binary to be built. If we know that changes in the channel never imply a source code change, as set in our workflow/lifecycle, we could write:

```
def package_id(self):
    self.info.requires["my_other_lib"].full_package()
    self.info.requires["my_other_lib"].channel = None # Channel doesn't change out␣
↪package ID
```

7.7 Define the package information

When creating a recipe to package a library, it is important to define the information about the package so consumers can get the information correctly. Conan achieves this by decoupling the information of the package from the format needed using *Generators*, that translate the generic information into the appropriate format file.

This generic information is defined inside the recipe, using the *package_info()* method. There you can declare package information like the location of the header files, library names, defines, flags...

```
from conans import ConanFile

class MyConan(ConanFile):
    name = "cool_library"
    ...

    def package_info(self):
        self.cpp_info.includedirs = ["include/cool"]
```

(continues on next page)

(continued from previous page)

```
        self.cpp_info.libs = ["libcool"]
        self.cpp_info.defines= ["DEFINE_COOL=1"]
```

The package information is done using the attributes of the *cpp_info* object. This information will be aggregated by Conan and exposed via `self.deps_cpp_info` to consumers and generators.

Important: This information is important as it describes the package contents in a generic way with a pretty straightforward syntax that can later be translated to a suitable format. The advantage of having this information here, is that the package could be consumed from a different build system that the one used to compile the library. For example, a library that builds using Autotools can be consumed later in CMake with this information using any of the CMake generators.

See also:

Read *package_info()* to learn more about this method.

7.7.1 Using Components

If your package contains more than one library or you want to define separated components so consumers can have more granular information, you can use components in your *package_info()* method.

> **Warning:** This is a **experimental** feature subject to breaking changes in future releases.

When you are creating a Conan package, it is recommended to have only one library (*.lib*, *.a*, *.so*, *.dll*...) per package. However, especially with third-party projects like Boost, Poco or OpenSSL, they would contain several libraries inside.

Usually those libraries inside the same package depend on each other and modelling the relationship among them is required.

With **components**, you can model libraries and executables inside the same package and how one depends on the other. Each library or executable will be one component inside `cpp_info` like this:

```
def package_info(self):
    self.cpp_info.names["cmake_find_package"] = "OpenSSL"
    self.cpp_info.names["cmake_find_package_multi"] = "OpenSSL"
    self.cpp_info.components["crypto"].names["cmake_find_package"] = "Crypto"
    self.cpp_info.components["crypto"].libs = ["libcrypto"]
    self.cpp_info.components["crypto"].defines = ["DEFINE_CRYPTO=1"]
    self.cpp_info.components["ssl"].names["cmake"] = "SSL"
    self.cpp_info.components["ssl"].includedirs = ["include/headers_ssl"]
    self.cpp_info.components["ssl"].libs = ["libssl"]
    self.cpp_info.components["ssl"].requires = ["crypto"]
```

You can define dependencies among different components using the `requires` attribute and the name of the component. The dependency graph for components will be calculated and values will be aggregated in the correct order for each field.

```
def package_info(self):
    self.cpp_info.components["LibA"].libs = ["liba"]      # Name of the library for the
↪'LibA' component
    self.cpp_info.components["LibA"].requires = ["LibB"]  # Requires point to the name␣
```

(continues on next page)

(continued from previous page)

```
↪of the component

    self.cpp_info.components["LibB"].libs = ["libb"]

    self.cpp_info.components["LibC"].libs = ["libc"]
    self.cpp_info.components["LibC"].requires = ["LibA"]

    self.cpp_info.components["LibD"].libs = ["libd"]
    self.cpp_info.components["LibD"].requires = ["LibA"]

    self.cpp_info.components["LibE"].libs = ["libe"]
    self.cpp_info.components["LibE"].requires = ["LibB"]

    self.cpp_info.components["LibF"].libs = ["libf"]
    self.cpp_info.components["LibF"].requires = ["LibD", "LibE"]
```

For consumers and generators, the order of the libraries from this components graph will be:

```
self.deps_cpp_info.libs == ["libf", "libe", "libd", "libc", "liba", "libb"]
```

Declaration of requires from other packages is also allowed:

```
class MyConan(ConanFile):
    ...
    requires = "zlib/1.2.11", "openssl/1.1.1g"

    def package_info(self):
        self.cpp_info.components["comp1"].requires = ["zlib::zlib"]  # Depends on all␣
↪components in zlib package
        self.cpp_info.components["comp2"].requires = ["comp1", "openssl::ssl"]  #␣
↪Depends on ssl component in openssl package
```

By default, components **won't link against any other package required by the recipe**. The requires list has to be **populated explicitly** with the list of components from other packages to use: it can be the full requirement (`zlib::zlib`) or a single component (`openssl::ssl`).

Important: The information of components is aggregated to the *global* `cpp_info` scope and the usage of components should be transparent.

Consumers can get this information via `self.deps_cpp_info` as usual and use it in the `build()` method of any dependent recipe:

```
class PocoTimerConan(ConanFile):
    ...
    requires = "zlib/1.2.11", "openssl/1.0.2u"
    ...

    def build(self):
        # Get the include directories of the SSL component of openssl package
        self.deps_cpp_info["openssl"].components["ssl"].include_paths
```

Recipes that require packages that declare components can also take advantage of this granularity, they can declare in

the `cpp_info.requires` attribute the list of components from the requirements they want to link with:

```
class Library(ConanFile):
    name = 'library'
    requires = "openssl/1.0.2u"

    def package_info(self):
        self.cpp_info.requires = ['openssl::ssl']
```

In the previous example, the 'library' package and transitively all its consumers will link only with the component `ssl` from the `openssl` package.

See also:

Read *components reference* for more information.

7.8 Toolchains

Toolchains are the new, experimental way to integrate with build systems in Conan. Recipes can define a `generate()` method that will return an object which can generate files from the current configuration that can be used by the build systems. Conan *generators* provide information about dependencies, while toolchains provide a "translation" from the Conan settings and options, and the recipe defined configuration to something that the build system can understand. A recipe that does not have dependencies does not need a generator, but can still use a toolchain.

A toolchain can be defined, among the built-ins toolchains, with an attribute with the name of the toolchain class to use.

```
generators = "<ToolChainClassName>"
```

For example, for using the CMake toolchain this should be declared in the recipe:

```
generators = "CMakeToolchain"
```

Note: At the moment (Conan 1.32), the available built-in toolchains are `CMakeToolchain`, `MSBuildToolchain` and `MesonToolchain`.

But in the more general case, and if it needs any specific configuration beyond the default one:

```
from conan.tools.cmake import CMakeToolchain

def generate(self):
    tc = CMakeToolchain(self)
    # customize toolchain "tc"
    tc.generate()
```

It is possible to use the `generate()` method to create your own files, which will typically be deduced from the current configuration of `self.settings` and `self.options`.

```
from conans.tools import save

def generate(self):
    # Based on the self.settings, self.options, the user
```

(continues on next page)

(continued from previous page)

```
    # can generate their own files:
    save("mytoolchain.tool", "my own toolchain contents, deduced from the settings and␣
↪options")
    # The "mytoolchain.tool" file can be used by the build system to
    # define the build
```

And as usual, you can create your own toolchain helpers, put them in a `python_requires` package and reuse them in all your recipes.

Toolchains have some important advantages:

- They execute at **`conan install`** time. They generate files, not command line arguments, providing better reproducibility and debugging of builds.
- They provide a better developer experience. The command line used by developers locally, like `cmake ...` will achieve the same build, with the same flags, as the **`conan build`** or the build that is done in the cache with a **`conan create`**.
- They are more extensible and configurable.

The toolchains implement most of the build system logic, leaving the build helpers, like `CMake()`, doing less work, and acting basically as a high level wrapper of the build system. Many of the existing arguments, attributes or methds of those build helpers will not be available. Check the documentation of each toolchain to check the associated build helper available functionality.

```
from conan.tools.cmake import CMakeToolchain, CMake

def generate(self):
    tc = CMakeToolchain(self)
    # customize toolchain "tc"
    tc.generate()

def build(self):
    # NOTE: This is a simplified helper
    # Not all arguments attributes and methods might be available
    cmake = CMake(self)
```

To learn more about existing built-in toolchains, read the reference in *tools*.

7.9 Inspecting Packages

You can inspect the uploaded packages and also the packages in the local cache by running the **`conan get`** command.

- List the files of a local recipe folder:

```
$ conan get zlib/1.2.11@ .

Listing directory '.':
 conandata.yml
 conanfile.py
 conanmanifest.txt
```

- Print the *conaninfo.txt* file of a binary package:

```
$ conan get zlib/1.2.11@:2144f833c251030c3cfd61c4354ae0e38607a909
```

- Print the *conanfile.py* from a remote package:

```
$ conan get zlib/1.2.11@ -r conancenter
```

```
import os
import stat
from conans import ConanFile, tools, CMake, AutoToolsBuildEnvironment
from conans.errors import ConanException


class ZlibConan(ConanFile):
    name = "zlib"
    version = "1.2.11"
    url = "https://github.com/conan-io/conan-center-index"
    homepage = "https://zlib.net"


    #...
```

Check the *conan get command* command reference and more examples.

7.10 Packaging Approaches

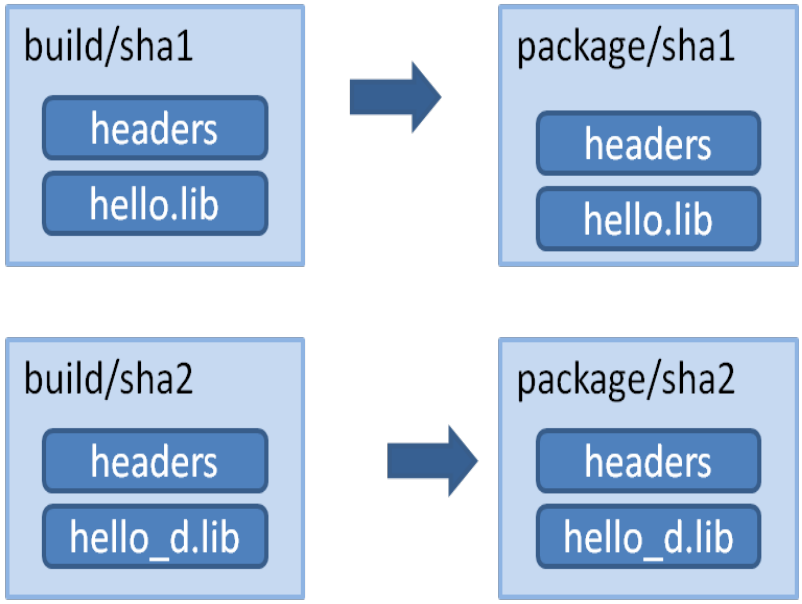
Package recipes have three methods for controlling the package's binary compatibility and for implementing different packaging approaches: *package_id()*, *build_id()* and *package_info()*.

These methods let package creators select the method most suitable for each library.

7.10.1 1 config (1 build) -> 1 package

A typical approach is to have one configuration for each package containing the artifacts. Using this approach, for example, the debug pre-compiled libraries will be in a different package than the release pre-compiled libraries.

So if there is a package recipe that builds a "hello" library, there will be one package containing the release version of the "hello.lib" library and a different package containing a debug version of that library (in the figure denoted as "hello_d.lib", to make it clear, it is not necessary to use different names).



Using this approach, the `package_info()` method, allows you to set the appropriate values for consumers, letting them know about the package library names, necessary definitions and compile flags.

```
class HelloConan(ConanFile):

    settings = "os", "compiler", "build_type", "arch"

    def package_info(self):
        self.cpp_info.libs = ["mylib"]
```

It is very important to note that it is declaring the `build_type` as a setting. This means that a different package will be generated for each different value of such setting.

The values declared by the packages (the *include*, *lib* and *bin* subfolders are already defined by default, so they define the include and library path to the package) are translated to variables of the respective build system by the used generators. That is, running the `cmake` generator will translate the above definition in the *conanbuildinfo.cmake* to something like:

```
set(CONAN_LIBS_MYPKG mylib)
# ...
set(CONAN_LIBS mylib ${CONAN_LIBS})
```

Those variables, will be used in the `conan_basic_setup()` macro to actually set the relevant cmake variables.

If the developer wants to switch configuration of the dependencies, they will usually switch with:

```
$ conan install -s build_type=Release ...
# when need to debug
$ conan install -s build_type=Debug ...
```

These switches will be fast, since all the dependencies are already cached locally.

This process offers a number of advantages:

- It is quite easy to implement and maintain.
- The packages are of minimal size, so disk space and transfers are faster, and builds from sources are also kept to the necessary minimum.

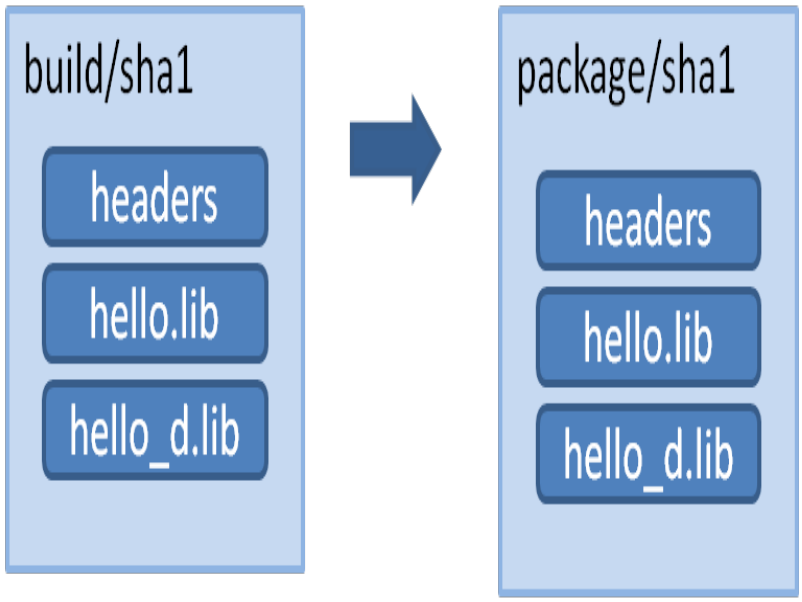
- The decoupling of configurations might help with isolating issues related to mixing different types of artifacts, and also protecting valuable information from deploy and distribution mistakes. For example, debug artifacts might contain symbols or source code, which could help or directly provide means for reverse engineering. So distributing debug artifacts by mistake could be a very risky issue.

Read more about this in *package_info()*.

7.10.2 N configs -> 1 package

Warning: This approach is discouraged. The support for defining multi-configuration packages (`self.cpp_info.release`, `self.cpp_info.debug`), will be removed in Conan 2.0, as discussed and approved by the Tribe in https://github.com/conan-io/tribe/pull/21. New generators and helpers in `conan.tools.xxxx`, like `CMakeDeps` or `MSBuildDeps` already ignore `cpp_info` multi-configuration definitions.

You may want to package both debug and release artifacts in the same package, so it can be consumed from IDEs like Visual Studio. This will change the debug/release configuration from the IDE, without having to specify it in the command line. This type of package can contain different artifacts for different configurations and can be used to include both the release and debug version of a library in the same package.



Note: A complete working example of the following code can be found in the examples repo: https://github.com/conan-io/examples

```
$ git clone https://github.com/conan-io/examples.git
$ cd features/multi_config
$ conan create . user/channel
```

Creating a multi-configuration debug/release package is simple

The first step will be to remove `build_type` from the settings. It will not be an input setting and the generated package will always contain both debug and release artifacts.

The Visual Studio runtime is different for debug and release (MDd or MD) and is set using the default runtime (MD/MDd). If this meets your needs, we recommend removing the compiler.runtime subsetting in the configure() method:

```
class HelloConan(ConanFile):
    # build_type has been omitted. It is not an input setting.
    settings = "os", "compiler", "arch"
    generators = "cmake"

    # Remove runtime and use always default (MD/MDd)
    def configure(self):
        if self.settings.compiler == "Visual Studio":
            del self.settings.compiler.runtime

    def build(self):
        cmake_release = CMake(self, build_type="Release")
        cmake_release.configure()
        cmake_release.build()

        cmake_debug = CMake(self, build_type="Debug")
        cmake_debug.configure()
        cmake_debug.build()
```

In this example, the binaries will be differentiated with a suffix in the CMake syntax, so we have to add this information to the data provided to the consumers in the *package_info* function:

```
set_target_properties(mylibrary PROPERTIES DEBUG_POSTFIX _d)
```

Such a package can define its information for consumers as:

```
def package_info(self):
    self.cpp_info.release.libs = ["mylibrary"]
    self.cpp_info.debug.libs = ["mylibrary_d"]
```

This will translate to the CMake variables:

```
set(CONAN_LIBS_MYPKG_DEBUG mylibrary_d)
set(CONAN_LIBS_MYPKG_RELEASE mylibrary)
# ...
set(CONAN_LIBS_DEBUG mylibrary_d ${CONAN_LIBS_DEBUG})
set(CONAN_LIBS_RELEASE mylibrary ${CONAN_LIBS_RELEASE})
```

And these variables will be correctly applied to each configuration by conan_basic_setup() helper.

In this case you can still use the general and not config-specific variables. For example, the include directory when set by default to *include* remains the same for both debug and release. Those general variables will be applied to all configurations.

Important: The above code assumes that the package will always use the default Visual Studio runtime (MD/MDd). To keep the package configurable for supporting static(MT)/dynamic(MD) linking with the VS runtime library, you can do the following:

- Keep the compiler.runtime setting, e.g. do not implement the configure() method removing it.
- Don't let the CMake helper define the CONAN_LINK_RUNTIME variable to define the runtime and define CONAN_LINK_RUNTIME_MULTI instead.

- In *CMakeLists.txt*, use the `CONAN_LINK_RUNTIME_MULTI` variable to correctly setup up the runtime for debug and release flags.
- Write a separate `package_id()` methods for MD/MDd and for MT/MTd defining the packages to be built.

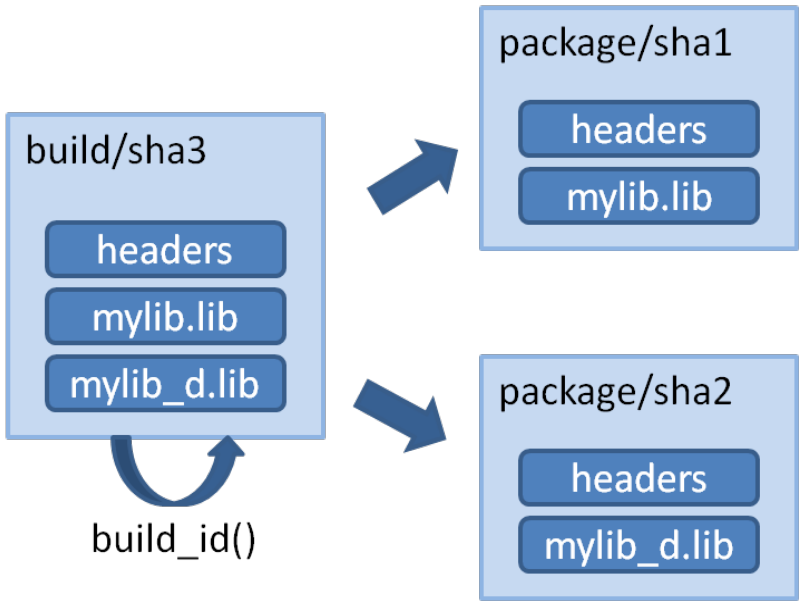
All these steps are already coded in the repo https://github.com/conan-io/examples/tree/master/features/multi_config and commented out as **"Alternative 2"**.

Note: The automatic conversion of multi-config variables to generators is currently implemented in the `cmake`, `visual_studio`, `txt`, and `cmake_find_package` generators (and also for their corresponding `_multi` implementations). If you want to have support for them in another build system, please open a GitHub issue.

7.10.3 N configs (1 build) -> N packages

It's possible that an existing build script is simultaneously building binaries for different configurations, like debug/release, or different architectures (32/64bits), or library types (shared/static). If such a build script is used in the previous "Single configuration packages" approach, it will definitely work without problems. However, we'll be wasting precious build time, as we'll be rebuilding the project for each package, then extracting the relevant artifacts for the relevant configuration, while ignoring the others.

It is more efficient to build the logic, whereby the same build can be reused to create different packages:



This can be done by defining a `build_id()` method in the package recipe that will specify the logic.

```
settings = "os", "compiler", "arch", "build_type"

def build_id(self):
    self.info_build.settings.build_type = "Any"

def package(self):
    if self.settings.build_type == "Debug":
        #package debug artifacts
```

(continues on next page)

(continued from previous page)

```
    else:
        # package release
```

Note that the `build_id()` method uses the `self.info_build` object to alter the build hash. If the method doesn't change it, the hash will match the package folder one. By setting `build_type="Any"`, we are forcing that for both the Debug and Release values of `build_type`, the hash will be the same (the particular string is mostly irrelevant, as long as it is the same for both configurations). Note that the build hash `sha3` will be different of both `sha1` and `sha2` package identifiers.

This does not imply that there will be strictly one build folder. There will be a build folder for every configuration (architecture, compiler version, etc). So if we just have Debug/Release build types, and we're producing N packages for N different configurations, we'll have N/2 build folders, saving half of the build time.

Read more about this in *build_id()*.

7.11 Package Creator Tools

Using Python (or just pure shell or bash) scripting, allows you to easily automate the whole package creation and testing process, for many different configurations. For example you could put the following script in the package root folder. Name it *build.py*:

```
import os, sys
import platform

def system(command):
    retcode = os.system(command)
    if retcode != 0:
        raise Exception("Error while executing:\n\t %s" % command)

if __name__ == "__main__":
    params = " ".join(sys.argv[1:])

    if platform.system() == "Windows":
        system('conan create . demo/testing -s compiler="Visual Studio" -s compiler.
↪version=14 %s' % params)
        system('conan create . demo/testing -s compiler="Visual Studio" -s compiler.
↪version=12 %s' % params)
        system('conan create . demo/testing -s compiler="gcc" -s compiler.version=4.8 %s
↪' % params)
    else:
        pass
```

This is a pure Python script, not related to Conan, and should be run as such:

```
$ python build.py
```

We have developed another FOSS tool for package creators, the **Conan Package Tools** to help you generate multiple binary packages from a package recipe. It offers a simple way to define the different configurations and to call **`conan test`**. In addition to offering CI integration like **Travis CI, Appveyor and Bamboo**, for cloud-based automated binary package creation, testing, and uploading.

This tool enables the creation of hundreds of binary packages in the cloud with a simple `$ git push` and supports:

- Easy **generation of multiple Conan packages** with different configurations.

- Automated/remote package generation in **Travis/Appveyor** server with distributed builds in CI jobs for big/slow builds.
- **Docker**: Automatic generation of packages for several versions of `gcc` and `clang` in Linux, and in Travis CI.
- Automatic creation of OSX packages with apple-clang, and in Travis-CI.
- **Visual Studio**: Automatic configuration of the command line environment with detected settings.

It's available in pypi:

```
$ pip install conan_package_tools
```

For more information, read the README.md in the Conan Package Tools repository.

CHAPTER

EIGHT

UPLOADING PACKAGES

This section shows how to upload packages using remotes and specifies the different binary repositories you can use.

8.1 Remotes

In the previous sections, we built several packages on our computer that were stored in the local cache, typically under *~/.conan/data*. Now, you might want to upload them to a Conan server for later use on another machine, project, or for sharing purposes.

Conan packages can be uploaded to different remotes previously configured with a name and a URL. The remotes are just servers used as binary repositories that store packages by reference.

There are several possibilities when uploading packages to a server:

For private development:

- **Artifactory Community Edition for C/C++**: Artifactory Community Edition (CE) for C/C++ is a completely free Artifactory server that implements both Conan and generic repositories. It is the recommended server for companies and teams wanting to host their own private repository. It has a web UI, advanced authentication and permissions, very good performance and scalability, a REST API, and can host generic artifacts (tarballs, zips, etc). Check *Artifactory Community Edition for C/C++* for more information.
- **Artifactory Pro**: Artifactory is the binary repository manager for all major packaging formats. It is the recommended remote type for enterprise and professional package management. Check the Artifactory documentation for more information. For a comparison between Artifactory editions, check the Artifactory Comparison Matrix.
- **Conan server**: Simple, free and open source, MIT licensed server that comes bundled with the Conan client. Check *Running conan_server* for more information.

For distribution:

- **Artifactory Cloud-hosted instance**: Artifactory Cloud, where JFrog manages, maintains and scales the infrastructure and provides automated server backups with free updates and guaranteed uptime. It's offered with a free tier designed for individual with reduced usage. Check *Uploading to Artifactory Cloud Instance* for more information.

8.1.1 conancenter

ConanCenter (https://conan.io/center) is the main official repository for open source Conan packages. It is configured as the default remote in the Conan client, but if you want to add it manually:

```
$ conan remote add conancenter https://center.conan.io
```

It contains **packages without "user/channel"** that can be used directly as *pkg/version* (*zlib/1.2.11*): These packages are created automatically from the central GitHub repository conan-center-index, with an automated build service: C3I (ConanCenter Continuous Integration).

To contribute packages to ConanCenter, read the *ConanCenter guide* for more information.

8.1.2 conan-center [deprecated]

conan-center was the official repository but is **no longer a default remote** in the Conan client and **its usage is completely discouraged**. This documentation is kept here only for reference purposes.

```
$ conan-center: https://conan.bintray.com [Verify SSL: True]
```

It contains all the packages that were uploaded to ConanCenter before July 1st (new packages are no longer uploaded to this remote), as well as **legacy packages with full reference** (*zlib/1.2.11@conan/stable*). These package binaries were created by users in their own Bintray repositories and included in this main repository. This flow of contributing packages to ConanCenter is no longer available and packages are **not recommended** and should be considered as **legacy**.

Important: This remote contains packages that are no longer maintained. We strongly encourage users to use *conan-center* and swift to the official package references without **user/channel** (*zlib/1.2.11@conan/stable -> zlib/1.2.11*).

8.2 Uploading Packages to Remotes

First, check if the remote you want to upload to is already in your current remote list:

```
$ conan remote list
```

You can easily add any remote. To run a remote on your machine:

```
$ conan remote add my_local_server http://localhost:9300
```

You can search any remote in the same way you search your computer. Actually, many Conan commands can specify a specific remote.

```
$ conan search -r=my_local_server
```

Now, upload the package recipe and all the packages to your remote. In this example, we are using our `my_local_server` remote, but you could use any other.

```
$ conan upload hello/0.1@demo/testing --all -r=my_local_server
```

You might be prompted for a username and password. The default Conan server remote has a **demo/demo** account we can use for testing.

The `--all` option will upload the package recipe plus all the binary packages. Omitting the `--all` option will upload the package recipe *only*. For fine-grained control over which binary packages are upload to the server, consider using the `--packages/-p` or `--query/-q` flags. `--packages` allows you to explicitly declare which package gets uploaded to the server by specifying the package ID. `--query` accepts a query parameter, e.g. `arch=armv8 and os=Linux`, and only uploads binary packages which match this query. When using the `--query` flag, ensure that your query string is enclosed in quotes to make the parameter explicit to your shell. For example, `conan upload <package> -q 'arch=x86_64 and os=Linux' ...` is appropriate use of the `--query` flag.

Now try again to read the information from the remote. We refer to it as remote, even if it is running on your local machine, as it could be running on another server in your LAN:

```
$ conan search hello/0.1@demo/testing -r=my_local_server
```

Note: If package upload fails, you can try to upload it again. Conan keeps track of the upload integrity and will only upload missing files.

Now we can check if we can download and use them in a project. For that purpose, we first have to **remove the local copies**, otherwise the remote packages will not be downloaded. Since we have just uploaded them, they are identical to the local ones.

```
$ conan remove "hello*"
$ conan search
```

Since we have our test setup from the previous section, we can just use it for our test. Go to your package folder and run the tests again, now saying that we don't want to build the sources again. We just want to check if we can download the binaries and use them:

```
$ conan create . demo/testing --not-export --build=never
```

You will see that the test is built, but the packages are not. The binaries are simply downloaded from your local server. You can check their existence on your local computer again with:

```
$ conan search
```

8.3 Using Artifactory

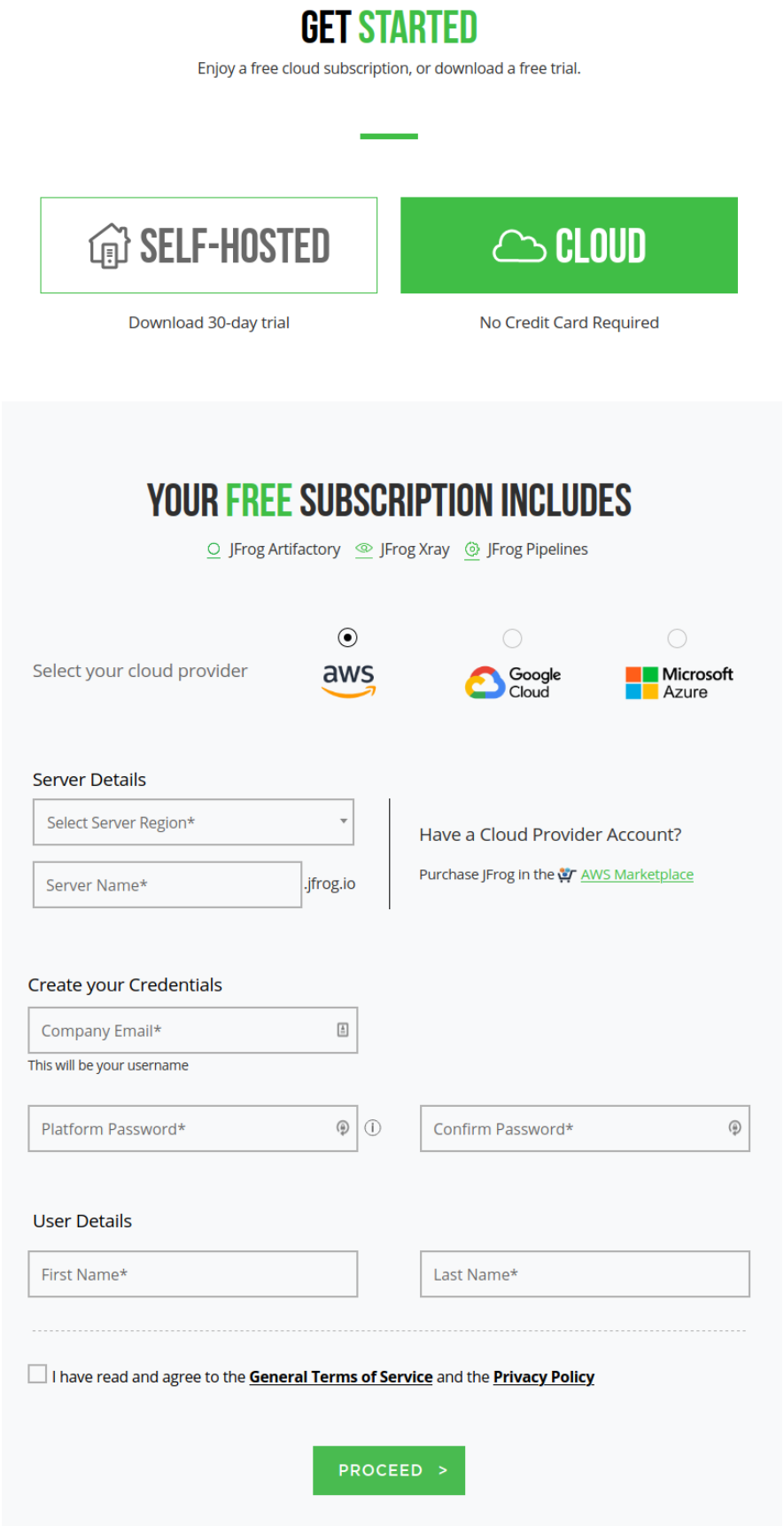
In Artifactory, you can create and manage as many free, personal Conan repositories as you like. On an Free-Tier account, all packages you upload are public, and anyone can use them by simply adding your repository to their Conan remotes. You still can create private repositories using *Artifatory CE* or Artifactory Cloud Premium account.

8.3.1 Uploading to Artifactory Cloud Instance

Conan packages can be uploaded to Artifactory under your own users or organizations. To create a repository follow these steps:

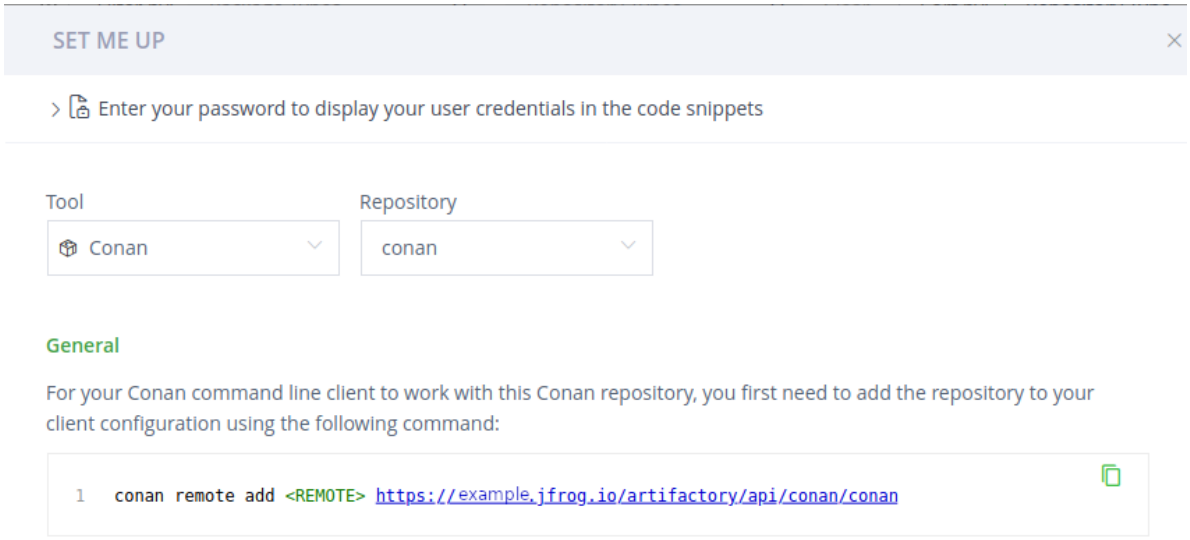
1. **Create an Artifactory Free-Tier account**

   Browse to https://jfrog.com/artifactory/start-free/?isConan=true and submit the form to create your account. Note that you don't have to use the same username that you use for your Conan account.


GET STARTED
Enjoy a free cloud subscription, or download a free trial.
SELF-HOSTED
CLOUD
Download 30-day trial
No Credit Card Required
YOUR FREE SUBSCRIPTION INCLUDES
JFrog Artifactory
JFrog Xray
JFrog Pipelines
Select your cloud provider
aws
Google Cloud
Microsoft Azure
Server Details
Select Server Region*
Server Name*
.jfrog.io
Have a Cloud Provider Account?
Purchase JFrog in the AWS Marketplace
Create your Credentials
Company Email*
This will be your username
Platform Password*
Confirm Password*
User Details
First Name*
Last Name*
I have read and agree to the General Terms of Service and the Privacy Policy
PROCEED >

2. **Add your Artifactory repository**

   To get the correct address, click on `Application -> Artifactory -> Artifacts -> Set Me Up`:

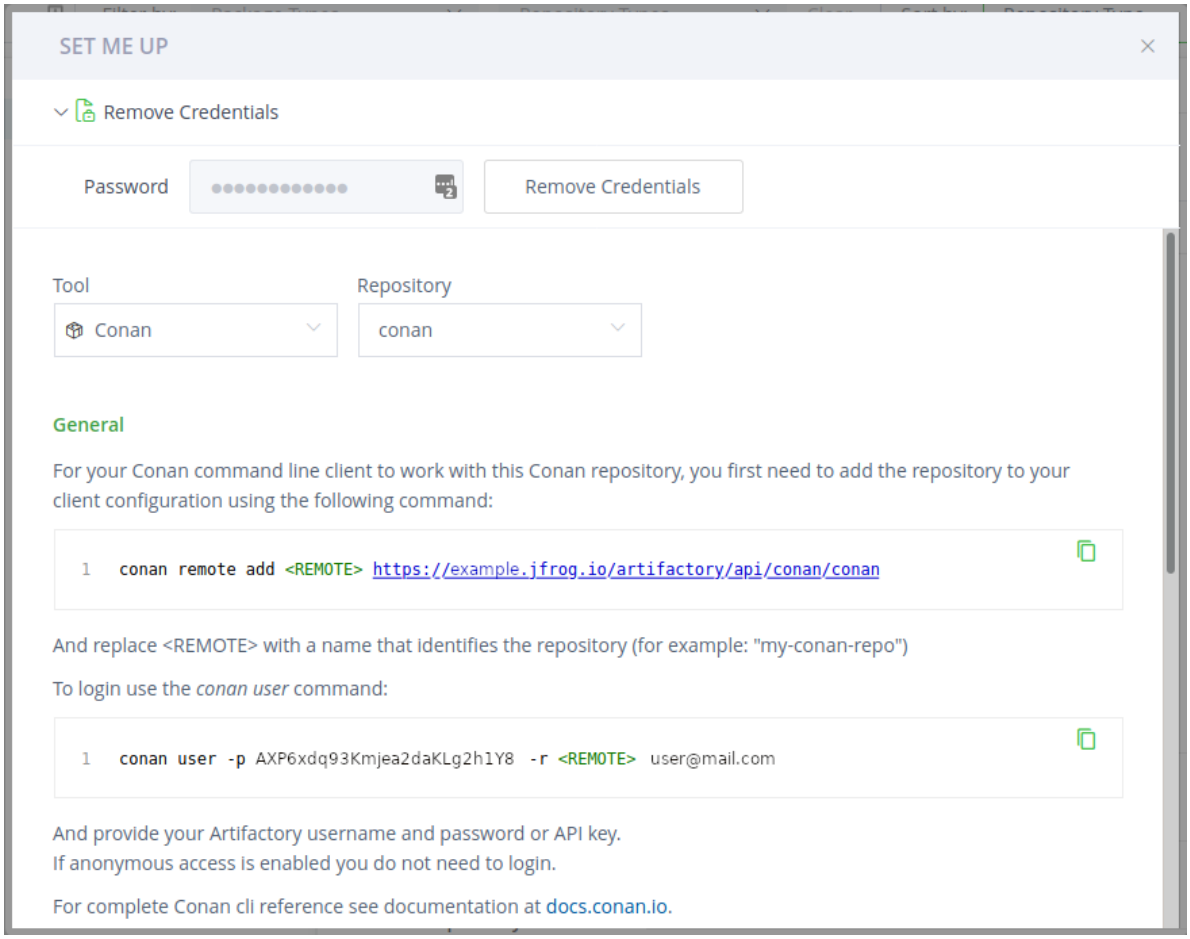


   Add a Conan remote in your Conan client pointing to your Artifactory repository.

```
$ conan remote add <REMOTE> <YOUR_ARTIFACTORY_REPO_URL>
```

4. **Get your API key**

   Your API key is the "password" used to authenticate the Conan client to Artifactory, NOT your Artifatory password. To get your API key, go to "Set Me Up" and enter your account password. Your API key will appear on conan user command line listed on Set Me Up box:



5. **Set your user credentials**

   Add your Conan user with the API Key, your remote and your Artifatory user name:

```
$ conan user -p <APIKEY> -r <REMOTE> <USEREMAIL>
```

Setting the remotes in this way will cause your Conan client to resolve packages and install them from repositories in the following order of priority:

1. conancenter
2. Your own repository

If you want to have your own repository first, please use the `--insert` command line option when adding it:

```
$ conan remote add <your_remote> <your_url> --insert 0
$ conan remote list
  <your remote>: <your_url> [Verify SSL: True]
  conancenter: https://center.conan.io [Verify SSL: True]
```

Tip: Check the full reference of *$ conan remote* command.

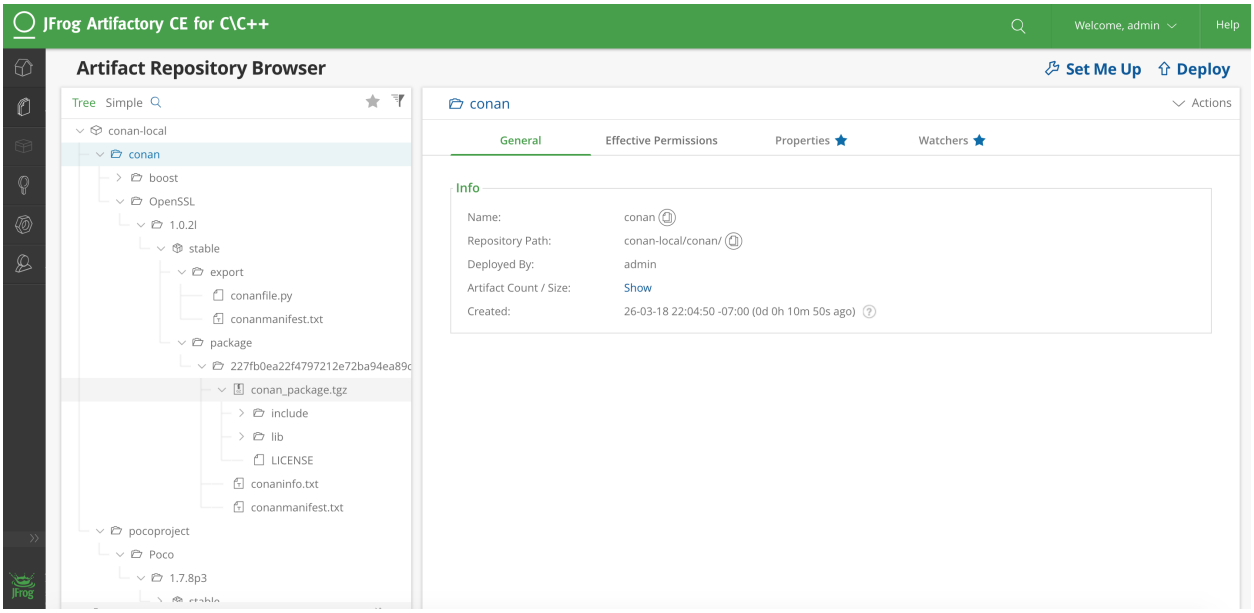
8.3.2 Artifactory Community Edition for C/C++

Artifactory Community Edition (CE) for C/C++ is the recommended server for development and hosting private packages for a team or company. It is completely free, and it features a WebUI, advanced authentication and permissions, great performance and scalability, a REST API, a generic CLI tool and generic repositories to host any kind of source or binary artifact.

This is a very brief introduction to Artifactory CE. For the complete Artifactory CE documentation, visit Artifactory docs.

Running Artifactory CE

There are several ways to download and run Artifactory CE. The simplest one might be to download and unzip the designated zip file, though other installers, including also installing from a Docker image. When the file is unzipped, launch Artifactory by double clicking the .bat or .sh script in the *bin* subfolder, depending on the OS. Java 8 update 45 or later runtime is required. If you don't have it, please install it first (newer Java versions preferred).



Once Artifactory has started, navigate to the default URL *http://localhost:8081*, where the Web UI should be running. The default user and password are `admin:password`.

Creating and Using a Conan Repo

Navigate to Admin -> Repositories -> Local, then click on the "New" button. A dialog for selecting the package type will appear, select Conan, then type a "Repository Key" (the name of the repository you are about to create), for example "conan-local". You can create multiple repositories to serve different flows, teams, or projects.

Now, it is necessary to configure the client. Go to Artifacts, and click on the created repository. The "Set Me Up" button in the top right corner provides instructions on how to configure the remote in the Conan client:

```
$ conan remote add artifactory http://localhost:8081/artifactory/api/conan/conan-local
```

From now, you can upload, download, search, etc. the remote repos similarly to the other repo types.

```
$ conan upload "*" --all -r=artifactory
$ conan search "*" -r=artifactory
```

Migrating from Other Servers

If you are already running another server, for example, the open source *conan_server*, it is easy to migrate your packages, using the Conan client to download the packages and re-upload them to the new server.

This Python script might be helpful, given that it already defines the respective `local` and `artifactory` remotes:

```
import os
import subprocess

def run(cmd):
    ret = os.system(cmd)
    if ret != 0:
        raise Exception("Command failed: %s" % cmd)

# Assuming local = conan_server and artifactory remotes
output = subprocess.check_output("conan search -r=local --raw")
packages = output.splitlines()

for package in packages:
    print("Downloading %s" % package)
    run("conan download %s -r=local" % package)

run("conan upload \"*\" --all --confirm -r=artifactory")
```

8.3.3 Contributing Packages to ConanCenter

Contribution of packages to ConanCenter is done via pull requests to the Github repository in https://github.com/conan-io/conan-center-index. The C3I (ConanCenter Continuous Integration) service will build binaries automatically from those pull requests, and once merged, will upload them to ConanCenter package repository.

Read more about how to submit a pull request to conan-center-index source repository.

8.4 Running conan_server

The *conan_server* is a free and open source server that implements Conan remote repositories. It is a very simple application, bundled with the regular Conan client installation. In most cases, it is recommended to use the free Artifactory Community Edition for C/C++ server, check *Artifactory Community Edition for C/C++* for more information.

`conan_server` needs Python>=3.6 for running.

Running the simple open source *conan_server* that comes with the Conan installers (or pip packages) is simple. Just open a terminal and type:

```
$ conan_server
```

Note: On Windows, you may experience problems with the server if you run it under bash/msys. It is better to launch it in a regular `cmd` window.

This server is mainly used for testing (though it might work fine for small teams). If you need a more stable, responsive and robust server, you should run it from source:

8.4.1 Running from Source (linux)

The Conan installer includes a simple executable **conan_server** for a server quick start. But you can use the **conan server** through the WSGI application, which means that you can use gunicorn to run the app, for example.

First, clone the Conan repository from source and install the requirements:

```
$ git clone https://github.com/conan-io/conan.git
$ cd conan
$ pip install -r conans/requirements.txt
$ pip install -r conans/requirements_server.txt
$ pip install gunicorn
```

Run the server application with `gunicorn`. In the following example, we run the server on port 9300 with four workers and a timeout of 5 minutes (300 seconds, for large uploads/downloads, you can also decrease it if you don't have very large binaries):

```
$ gunicorn -b 0.0.0.0:9300 -w 4 -t 300 conans.server.server_launcher:app
```

Note: Please note the timeout of `-t 300` seconds, resulting in a 5 minute parameter. If your transfers are very large or on a slow network, you might need to increase that value.

You can also bind to an IPv6 address or specify both IPv4 and IPv6 addresses:

```
$ gunicorn -b 0.0.0.0:9300 -b [::1]:9300 -w 4 -t 300 conans.server.server_launcher:app
```

8.4.2 Server Configuration

By default your server configuration is saved under `~/.conan_server/server.conf`, however you can modify this behaviour by either setting the `CONAN_SERVER_HOME` environment variable or launching the server with `-d` or `--server_dir` command line argument followed by desired path. In case you use one of the options your configuration file will be stored under `server_directory/server.conf` Please note that command line argument will override the environment variable. You can change configuration values in `server.conf`, prior to launching the server. Note that the server does not support hot-reload, and thus in order to see configuration changes you will have to manually relaunch the server.

The server configuration file is by default:

```
[server]
jwt_secret: MnpuzsExftskYGOMgaTYDKfw
jwt_expire_minutes: 120

ssl_enabled: False
port: 9300
```

(continues on next page)

(continued from previous page)

```
public_port:
host_name: localhost

store_adapter: disk
authorize_timeout: 1800

# Just for disk storage adapter
disk_storage_path: ~/.conan_server/data
disk_authorize_timeout: 1800

updown_secret: NyiSWNWnwumTVpGpoANuyyhR


[write_permissions]
# "opencv/2.3.4@lasote/testing": default_user,default_user2

[read_permissions]
# opencv/1.2.3@lasote/testing: default_user default_user2
# By default all users can read all blocks
*/*@*/*: *

[users]
demo: demo
```

Server Parameters

- `port`: Port where **conan_server** will run.
- The client server authorization is done with JWT. `jwt_secret` is a random string used to generate authentication tokens. You can change it safely anytime (in fact it is a good practice). The change will just force users to log in again. `jwt_expire_minutes` is the amount of time that users remain logged-in within the client without having to introduce their credentials again.

Other parameters (not recommended from Conan 1.1, but necessary for previous versions):

- `host_name`: If you set `host_name`, you must use the machine's IP where you are running your server (or domain name), something like **host_name: 192.168.1.100**. This IP (or domain name) has to be visible (and resolved) by the Conan client, so take it into account if your server has multiple network interfaces.
- `public_port`: Might be needed when running virtualized, Docker or any other kind of port redirection. File uploads/downloads are served with their own URLs, generated by the system, so the file storage backend is independent. Those URLs need the public port they have to communicate from the outside. If you leave it blank, the `port` value is used.

  Example: Use conan_server in a Docker container that internally runs in the 9300 port but exposes the 9999 port (where the clients will connect to):

  ```
  docker run ... -p9300:9999 ... # Check Docker docs for that
  ```

  server.conf

  ```
  [server]

  ssl_enabled: False
  ```

(continues on next page)

(continued from previous page)

```
port: 9300
public_port: 9999
host_name: localhost
```

- `ssl_enabled` Conan doesn't handle the SSL traffic by itself, but you can use a proxy like Nginx to redirect the SSL traffic to your Conan server. If your Conan clients are connecting with "https", set *ssl_enabled* to True. This way the conan_server will generate the upload/download urls with "https" instead of "http".

Note: **Important**: The Conan client, by default, will validate the server SSL certificates and won't connect if it's invalid. If you have self signed certificates you have two options:

1. Use the **`conan remote`** command to disable the SSL certificate checks. E.g., *conan remote add/update myremote https://somedir False*
2. Append the server *.crt* file contents to *~/.conan/cacert.pem* file.

To learn more, see *How to manage SSL (TLS) certificates*.

Conan has implemented an extensible storage backend based on the abstract class `StorageAdapter`. Currently, the server only supports storage on `disk`. The folder in which the uploaded packages are stored (i.e., the folder you would want to backup) is defined in the `disk_storage_path`.

The storage backend might use a different channel, and uploads/downloads are authorized up to a maximum of `authorize_timeout` seconds. The value should sufficient so that large downloads/uploads are not rejected, but not too big to prevent hanging up the file transfers. The value `disk_authorize_timeout` is not currently used. File transfers are authorized with their own tokens, generated with the secret `updown_secret`. This value should be different from the above `jwt_secret`.

Running the Conan Server with SSL using Nginx

server.conf

```
[server]
port: 9300
```

nginx conf file

```
server {
    listen 443;
    server_name myservername.mydomain.com;

    location / {
      proxy_pass http://0.0.0.0:9300;
    }
    ssl on;
    ssl_certificate /etc/nginx/ssl/server.crt;
    ssl_certificate_key /etc/nginx/ssl/server.key;
}
```

remote configuration in Conan client

```
$ conan remote add myremote https://myservername.mydomain.com
```

Running the Conan Server with SSL using Nginx in a Subdirectory

server.conf

```
[server]
port: 9300
```

nginx conf file

```
server {

       listen 443;
       ssl on;
       ssl_certificate /usr/local/etc/nginx/ssl/server.crt;
       ssl_certificate_key /usr/local/etc/nginx/ssl/server.key;
       server_name myservername.mydomain.com;

       location /subdir/ {
          proxy_pass http://0.0.0.0:9300/;
       }
  }
```

remote configuration in Conan client

```
$ conan remote add myremote https://myservername.mydomain.com/subdir/
```

Running Conan Server using Apache

You need to install `mod_wsgi`. If you want to use Conan installed from `pip`, the conf file should be similar to the following example:

Apache conf file (e.g., /etc/apache2/sites-available/0_conan.conf)

```
<VirtualHost *:80>
    WSGIScriptAlias / /usr/local/lib/python3.6/dist-packages/conans/server/
↪server_launcher.py
    WSGICallableObject app
    WSGIPassAuthorization On

    <Directory /usr/local/lib/python3.6/dist-packages/conans>
        Require all granted
    </Directory>
</VirtualHost>
```

If you want to use Conan checked out from source in, for example in */srv/conan*, the conf file should be as follows:

Apache conf file (e.g., /etc/apache2/sites-available/0_conan.conf)

```
<VirtualHost *:80>
    WSGIScriptAlias / /srv/conan/conans/server/server_launcher.py
    WSGICallableObject app
    WSGIPassAuthorization On
```

(continues on next page)

(continued from previous page)

```
    <Directory /srv/conan/conans>
        Require all granted
    </Directory>
</VirtualHost>
```

The directive `WSGIPassAuthorization On` is needed to pass the HTTP basic authentication to Conan.

Also take into account that the server config files are located in the home of the configured Apache user, e.g., var/www/.conan_server, so remember to use that directory to configure your Conan server.

Permissions Parameters

By default, the server configuration when set to Read can be done anonymous, but uploading requires you to be registered users. Users can easily be registered in the `[users]` section, by defining a pair of `login: password` for each one. Plain text passwords are used at the moment, but as the server is on-premises (behind firewall), you just need to trust your sysadmin :)

If you want to restrict read/write access to specific packages, configure the `[read_permissions]` and `[write_permissions]` sections. These sections specify the sequence of patterns and authorized users, in the form:

```
# use a comma-separated, no-spaces list of users
package/version@user/channel: allowed_user1,allowed_user2
```

E.g.:

```
*/*@*/*: * # allow all users to all packages
PackageA/*@*/*: john,peter # allow john and peter access to any PackageA
*/*@project/*: john # Allow john to access any package from the "project" user
```

The rules are evaluated in order. If the left side of the pattern matches, the rule is applied and it will not continue searching for matches.

Authentication

By default, Conan provides a simple `user: password` users list in the `server.conf` file.

There is also a plugin mechanism for setting other authentication methods. The process to install any of them is a simple two-step process:

1. Copy the authenticator source file into the `.conan_server/plugins/authenticator` folder.
2. Add `custom_authenticator: authenticator_name` to the `server.conf` [server] section.

This is a list of available authenticators, visit their URLs to retrieve them, but also to report issues and collaborate:

- **htpasswd**: Use your server Apache htpasswd file to authenticate users. Get it: https://github.com/d-schiffner/conan-htpasswd
- **LDAP**: Use your LDAP server to authenticate users. Get it: https://github.com/uilianries/conan-ldap-authentication

Create Your Own Custom Authenticator

If you want to create your own Authenticator, create a Python module in `~/.conan_server/plugins/authenticator/my_authenticator.py`

Example:

```
def get_class():
    return MyAuthenticator()


class MyAuthenticator(object):
    def valid_user(self, username, plain_password):
        return username == "foo" and plain_password == "bar"
```

The module has to implement:

- A factory function `get_class()` that returns a class with a `valid_user()` method instance.
- The class containing the `valid_user()` that has to return True if the user and password are valid or False otherwise.

Got any doubts? Please check out our *FAQ section* or .

CHAPTER

NINE

DEVELOPING PACKAGES

This section shows how to work on packages with source code continuously being modified.

9.1 Package development flow

In the previous examples, we used the **`conan create`** command to create a package of our library. Every time it is run, Conan performs the following costly operations:

1. Copy the sources to a new and clean build folder.
2. Build the entire library from scratch.
3. Package the library once it is built.
4. Build the `test_package` example and test if it works.

But sometimes, especially with big libraries, while we are developing the recipe, **we cannot afford** to perform these operations every time.

The following section describes the local development flow, based on the Bincrafters community blog.

The local workflow encourages users to perform trial-and-error in a local sub-directory relative to their recipe, much like how developers typically test building their projects with other build tools. The strategy is to test the *conanfile.py* methods individually during this phase.

We will use this conan flow example to follow the steps in the order below.

9.1.1 conan source

You will generally want to start off with the **`conan source`** command. The strategy here is that you're testing your source method in isolation, and downloading the files to a temporary sub-folder relative to the *conanfile.py*. This just makes it easier to get to the sources and validate them.

This method outputs the source files into the source-folder.

Input folders	Output folders
–	`source-folder`

```
$ cd example_conan_flow
$ conan source . --source-folder=tmp/source

PROJECT: Configuring sources in C:\Users\conan\example_conan_flow\tmp\source
Cloning into 'hello'...
...
```

Once you've got your source method right and it contains the files you expect, you can move on to testing the various attributes and methods related to downloading dependencies.

9.1.2 conan install

Conan has multiple methods and attributes which relate to dependencies (all the ones with the word "require" in the name). The command **conan install** activates all them.

Input folders	Output folders
–	install-folder

```
$ conan install . --install-folder=tmp/build [--profile XXXX]

PROJECT: Installing C:\Users\conan\example_conan_flow\conanfile.py
Requirements
Packages
...
```

This also generates the *conaninfo.txt* and *conanbuildinfo.xyz* files (extensions depends on the generator you've used) in the temp folder (`install-folder`), which will be needed for the next step. Once you've got this command working with no errors, you can move on to testing the `build()` method.

9.1.3 conan build

The build method takes a path to a folder that has sources and also to the install folder to get the information of the settings and dependencies. It uses a path to a folder where it will perform the build. In this case, as we are including the *conanbuildinfo.cmake* file, we will use the folder from the install step.

Input folders	Output folders
source-folder install-folder	build-folder

```
$ conan build . --source-folder=tmp/source --build-folder=tmp/build

Project: Running build()
...
Build succeeded.
    0 Warning(s)
    0 Error(s)

Time Elapsed 00:00:03.34
```

Here we can avoid the repetition of `--install-folder=tmp/build` and it will be defaulted to the `--build-folder` value.

This is pretty straightforward, but it does add a very helpful new shortcut for people who are packaging their own library. Now, developers can make changes in their normal source directory and just pass that path as the `--source-folder`.

9.1.4 conan package

Just as it sounds, this command now simply runs the `package()` method of a recipe. It needs all the information of the other folders in order to collect the needed information for the package: header files from source folder, settings and dependency information from the install folder and built artifacts from the build folder.

Input folders	Output folders

`source-folder`
`install-folder`
`build-folder`
	`package-folder`

```
$ conan package . --source-folder=tmp/source --build-folder=tmp/build --package-
↪folder=tmp/package

PROJECT: Generating the package
PROJECT: Package folder C:\Users\conan\example_conan_flow\tmp\package
PROJECT: Calling package()
PROJECT package(): Copied 1 '.h' files: hello.h
PROJECT package(): Copied 2 '.lib' files: greet.lib, hello.lib
PROJECT: Package 'package' created
```

9.1.5 conan export-pkg

When you have checked that the package is done correctly, you can generate the package in the local cache. Note that the package is generated again to make sure this step is always reproducible.

This parameters takes the same parameters as `package()`.

Input folders	Output folders

`source-folder`
`install-folder`
`build-folder`
`package-folder`
	–

There are 2 modes of operation:

- Using `source-folder` and `build-folder` will use the `package()` method to extract the artifacts from those folders and create the package, directly in the Conan local cache. Strictly speaking, it doesn't require executing a **`conan package`** before, as it packages directly from these source and build folders, though **`conan package`** is still recommended in the dev-flow to debug the `package()` method.
- Using the `package-folder` argument (incompatible with the above 2), will not use the `package()` method, it will create an exact copy of the provided folder. It assumes the package has already been created by a previous **`conan package`** command or with a **`conan build`** command with a `build()` method running a `cmake.install()`.

```
$ conan export-pkg . user/channel --source-folder=tmp/source --build-folder=tmp/build --
↪profile=myprofile

Packaging to 6cc50b139b9c3d27b3e9042d5f5372d327b3a9f7
hello/1.1@user/channel: Generating the package
hello/1.1@user/channel: Package folder C:\Users\conan\.conan\data\hello\1.1\user\channel\
↪package\6cc50b139b9c3d27b3e9042d5f5372d327b3a9f7
hello/1.1@user/channel: Calling package()
hello/1.1@user/channel package(): Copied 2 '.lib' files: greet.lib, hello.lib
hello/1.1@user/channel package(): Copied 2 '.lib' files: greet.lib, hello.lib
hello/1.1@user/channel: Package '6cc50b139b9c3d27b3e9042d5f5372d327b3a9f7' created
```

9.1.6 conan test

The final step to test the package for consumers is the test command. This step is quite straight-forward:

```
$ conan test test_package hello/1.1@user/channel

hello/1.1@user/channel (test package): Installing C:\Users\conan\repos\example_conan_
↪flow\test_package\conanfile.py
Requirements
    hello/1.1@user/channel from local
Packages
    hello/1.1@user/channel:6cc50b139b9c3d27b3e9042d5f5372d327b3a9f7

hello/1.1@user/channel: Already installed!
hello/1.1@user/channel (test package): Generator cmake created conanbuildinfo.cmake
hello/1.1@user/channel (test package): Generator txt created conanbuildinfo.txt
hello/1.1@user/channel (test package): Generated conaninfo.txt
hello/1.1@user/channel (test package): Running build()
...
```

There is often a need to repeatedly re-run the test to check the package is well generated for consumers.

As a summary, you could use the default folders and the flow would be as simple as:

```
$ git clone https://github.com/conan-io/examples.git
$ cd features/package_development_flow
$ conan source .
$ conan install . -pr=default
$ conan build .
$ conan package .
# So far, this is local. Now put the local binaries in cache
$ conan export-pkg . hello/1.1@user/testing -pr=default
# And test it, to check it is working in the local cache
$ conan test test_package hello/1.1@user/testing
...
hello/1.1@user/testing (test package): Running test()
Hello World Release!
```

9.1.7 conan create

Now we know we have all the steps of a recipe working. Thus, now is an appropriate time to try to run the recipe all the way through, and put it completely in the local cache.

The usual command for this is **conan create** and it basically performs the previous commands with **conan test** for the *test_package* folder:

```
$ conan create . user/channel
```

Even with this command, the package creator can iterate over the local cache if something does not work. This could be done with `--keep-source` and `--keep-build` flags.

If you see in the traces that the `source()` method has been properly executed but the package creation finally failed, you can skip the `source()` method the next time issue **conan create** using **--keep-source**:

```
$ conan create . user/channel --keep-source

hello/1.1@user/channel: A new conanfile.py version was exported
hello/1.1@user/channel: Folder: C:\Users\conan\.conan\data\hello\1.1\user\channel\export
hello/1.1@user/channel (test package): Installing C:\Users\conan\repos\features\package_
↪development_flow\test_package\conanfile.py
Requirements
    hello/1.1@user/channel from local
Packages
    hello/1.1@user/channel:6cc50b139b9c3d27b3e9042d5f5372d327b3a9f7

hello/1.1@user/channel: WARN: Forced build from source
hello/1.1@user/channel: Building your package in C:\Users\conan\.conan\data\hello\1.1\
↪user\channel\build\6cc50b139b9c3d27b3e9042d5f5372d327b3a9f7
hello/1.1@user/channel: Configuring sources in C:\Users\conan\.conan\data\hello\1.1\user\
↪channel\source
Cloning into 'hello'...
remote: Counting objects: 17, done.
remote: Total 17 (delta 0), reused 0 (delta 0), pack-reused 17
Unpacking objects: 100% (17/17), done.
Switched to a new branch 'static_shared'
Branch 'static_shared' set up to track remote branch 'static_shared' from 'origin'.
hello/1.1@user/channel: Copying sources to build folder
hello/1.1@user/channel: Generator cmake created conanbuildinfo.cmake
hello/1.1@user/channel: Calling build()
...
```

If you see that the library is also built correctly, you can also skip the `build()` step with the `--keep-build` flag:

```
$ conan create . user/channel --keep-build
```

9.2 Package layout

Warning: This is an **experimental** feature subject to breaking changes in future releases. The `layout()` feature will be fully functional only in the new build system integrations (*in the conan.tools space*). If you are using other integrations, they might not fully support this feature.

Available since: 1.37.0

9.2.1 Before starting

To understand correctly how the `layout()` method can help us we need to recall first how Conan works.

Let's say we are working in a project, using, for example, CMake:

```
<my_project_folder>
├── conanfile.py
└── src
    ├── CMakeLists.txt
    ├── hello.cpp
    ├── my_tool.cpp
    └── include
        └── hello.h
```

When we call `conan create`, this is a simplified description of what happens:

1. Conan exports the recipe (conanfile.py) and the declared sources (exports_sources) to the cache. The folders in the cache would be something like:

Listing 1: .conan/data/<some_cache_folder>

```
├── export
│   └── conanfile.py
└── export_source
    └── src
        ├── CMakeLists.txt
        ├── hello.cpp
        ├── my_tool.cpp
        └── include
            └── hello.h
```

2. If the method `source()` exists, it might retrieve sources from the internet. Also, the `export_source` folder is copied to the `source` folder.

Listing 2: .conan/data/<some_cache_folder>

```
├── export
│   └── conanfile.py
├── export_source
│   └── src
│       ├── CMakeLists.txt
│       ├── hello.cpp
│       ├── my_tool.cpp
```

(continues on next page)

(continued from previous page)

```
│       └── include
│           └── hello.h
└── source
    └── src
        ├── CMakeLists.txt
        ├── hello.cpp
        ├── my_tool.cpp
        └── include
            └── hello.h
```

3. Before calling the `build()` method, a build folder is created and the **sources** are copied there. Later, we call the `build()` method so the libraries and executables are built:

Listing 3: .conan/data/<some_cache_folder>

```
├── export
│   └── conanfile.py
├── export_source
│   └── src
│       ├── CMakeLists.txt
│       ├── hello.cpp
│       ├── my_tool.cpp
│       └── include
│           └── hello.h
├── source
│   └── src
│       ├── CMakeLists.txt
│       ├── hello.cpp
│       ├── my_tool.cpp
│       └── include
│           └── hello.h
└── build
    └── <build_id>
        ├── say.a
        └── bin
            └── my_app
```

4. At last, Conan calls the `package()` method to copy the built artifacts from the `source` (typically includes) and `build` folders (libraries and executables) to a **package** folder.

Listing 4: .conan/data/<some_cache_folder>

```
├── export
│   └── conanfile.py
├── export_source
│   └── src
│       ├── CMakeLists.txt
│       ├── hello.cpp
│       ├── my_tool.cpp
│       └── include
│           └── hello.h
├── source
│   └── src
```

(continues on next page)

(continued from previous page)

```
│       ├── CMakeLists.txt
│       ├── hello.cpp
│       ├── my_tool.cpp
│       └── include
│           └── hello.h
├── build
│   └── <build_id>
│       ├── say.a
│       └── bin
│           └── my_app
└── package
    └── <package_id>
        ├── lib
        │   └── say.a
        ├── bin
        │   └── my_app
        └── include
            └── hello.h
```

5. The `package_info(self)` method will describe with the `self.cpp_info` object the contents of the `package` folder, that is the one the consumers use to link against it. If we call *conan create* with different configurations the base folder in the cache is different and nothing gets messed.

Listing 5: conanfile.py

```
import os
from conans import ConanFile
from conan.tools.cmake import CMake


class SayConan(ConanFile):
    name = "say"
    version = "0.1"
    exports_sources = "src/*"
    ...
    def package_info(self):
        # These are default values and doesn't need to be adjusted
        self.cpp_info.includedirs = ["include"]
        self.cpp_info.libdirs = ["lib"]
        self.cpp_info.bindirs = ["bin"]

        # The library name
        self.cpp_info.libs = ["say"]
```

So, this workflow in the cache works flawlessly but:

- What if I'm developing the recipe in my local project and want to use the local methods (**conan source**, **conan build**) and later call **export-pkg** to create the package?

  If you call **conan build** in your working directory, without specifying a `--build-folder` argument, you will end up with a bunch of files polluting your project. Moreover, if you want to build more configurations you will need to create several build folders by hand, this is inconvenient, error-prone, and wouldn't be easy for Conan to locate the correct artifacts if you want to call **export-pkg** later.

- What if I don't even want to call **conan build** but use my CLion IDE to build the project?

By default, the CLion IDE will create the folders **cmake-build-release** and **cmake-build-debug** to put the build files there, so maybe your `package()` method is not able to locate the files in there and the **export-pkg** might fail.

- What if I want to use my project as an *editable package*?

  If you want to keep developing your package but let the consumers link with the artifacts in your project instead of the files in the Conan cache, you would need to declare a yml file describing where the headers, libraries and executables are in your application.

So, just as we describe the package folder in the `package_info()` method, we can use `layout()` to describe the `source` and `build` folders (both in a local project and in the cache):

- We can run the conan local commands (**conan source**, **conan build**, **conan export-pkg**) without taking care of specifying directories, always with the same syntax.
- If you are using an IDE, you can describe the build folder naming in the layout, so the libraries and executables are always in a known place.
- In the cache, the layout (like a build subfolder) is kept, so we can always know where the artifacts are before packaging them.
- It enables tools like the *AutoPackager* to automate the **package()** method.
- It out-of-the-box enables to use *editable packages*, because the recipe describes where the contents will be, even for different configurations, so the consumers can link with the correct built artifacts.

9.2.2 Declaring the layout

In the `layout()` method, you can set:

- **self.folders**
  - **self.folders.source**: To specify a folder where your sources are.
  - **self.folders.build**: To specify a subfolder where the files from the build are (or will be).
  - **self.folders.generators**: To specify a subfolder where to write the files from the generators and the toolchains (e.g. the *xx-config.cmake* files from the `CMakeDeps` generator).
  - **self.folders.imports**: To specify a subfolder where to write the files copied when using the `imports(self)` method in a `conanfile.py`.
  - **self.folders.root**: To specify the relative path from the `conanfile.py` to the root of the project, in case the `conanfile.py` is in a subfolder and not in the project root. If defined, all the other paths will be relative to the project root, not to the location of the `conanfile.py`.

  Check the *complete reference* of the **self.folders** attribute.
- **self.cpp.source** and **self.cpp.build**: The same you set the `self.cpp.package` to describe the package folder after calling the `package()` method, you can also describe the *source* and *build* folders.
- **self.cpp.package**: You can use it as you use the **self.cpp_info** at the `package_info(self)` method. The **self.cpp_info** object will be populated with the information declared in the `self.cpp.package` object, so you can complete it or modify it later in the `package_info(self)` method.

9.2.3 Example: Everything together

Let's say we are working in the project introduced in the section above:

```
<my_project_folder>
├── conanfile.py
└── src
    ├── CMakeLists.txt
    ├── hello.cpp
    ├── my_tool.cpp
    └── include
        └── hello.h
```

We are using the following **CMakeLists.txt**:

```
cmake_minimum_required(VERSION 3.15)
project(say CXX)

add_library(say hello.cpp)
target_include_directories(say PUBLIC "include")

add_executable(my_tool my_tool.cpp)
target_link_libraries(my_tool say)

# The executables are generated at the "bin" folder
set_target_properties(my_tool PROPERTIES RUNTIME_OUTPUT_DIRECTORY "${CMAKE_BINARY_DIR}/
↪bin")
```

Let's see how we describe our project in the `layout()` method:

Listing 6: conanfile.py

```
import os
from conans import ConanFile
from conan.tools.cmake import CMake


class SayConan(ConanFile):
    name = "say"
    version = "0.1"
    exports_sources = "src/*"
    ...
    def layout(self):
        self.folders.source = "src"
        build_type = str(self.settings.build_type).lower()
        self.folders.build = "cmake-build-{}".format(build_type)
        self.folders.generators = os.path.join(self.folders.build, "conan")

        self.cpp.package.libs = ["say"]
        self.cpp.package.includedirs = ["include"] # includedirs is already set to this␣
↪value by
                                                   # default, but declared for completion

        # this information is relative to the source folder
```

(continues on next page)

(continued from previous page)

```
        self.cpp.source.includedirs = ["include"]  # maps to ./src/include

        # this information is relative to the build folder
        self.cpp.build.libdirs = ["."]             # maps to ./cmake-build-<build_type>
        self.cpp.build.bindirs = ["bin"]           # maps to ./cmake-build-<build_type>/
↪bin

    def build(self):
        cmake = CMake(self)
        cmake.configure()
        cmake.build()
        # we can also know where the executable we are building is
        self.run(os.path.join(self.build_folder, self.cpp.build.bindirs[0], "my_tool"))
```

Let's review the `layout()` method changes:

- **self.folders**
  - As we have our sources in the `src` folder, `self.folders.source` is set to "**src**".
  - We set `self.folders.build` to be **cmake-build-release** or **cmake-build-debug** depending on the build_type.
  - The `self.folders.generators` folder is where all files generated by Conan will be stored so they don't pollute the other folders.

  Please, note that the values above are for a single-configuration CMake generator. To support multi-configuration generators, such as Visual Studio, you should make some changes to this layout. For a complete layout that supports both single-config and multi-config, please check the *cmake_layout()* in the Conan documentation.

- **self.cpp**

  We can set the information about the package that the consumers need to use by setting the conanfile's `cpp.package` attributes values:

  - Declaring `self.cpp.package.libs` inside the `layout()` method is equivalent to the "classic" `self.cpp_info.libs` declaration in the `package_info()` method.
  - Also, as you may know, `self.cpp.package.includedirs` is set to `["include"]` by default, so there's no need in declaring it but we are leaving it here for completeness.

  We can also describe the `source` and `build` folders with the `cpp.source` and `cpp.build` objects:

  - We are setting `self.cpp.source.includedirs = ["include"]`. The `self.folders.source` information will be automatically prepended to that path for consumers so, for example, when working with an editable package, Conan will try to get the include files from the **./my_project_folder/src/include** folder.
  - We set the `self.cpp.build.libdirs` to **["."]**, so we are declaring that, if we make the package `editable`, the libraries will be at the **./cmake-build-<build_type>** folder.
  - We set the `self.cpp.build.bindirs` to **["bin"]**, because the `CMakeLists.txt` file is changing the `RUNTIME_OUTPUT_DIRECTORY` to that directory.

There is also an interesting line in the `build(self)` method:

Listing 7: conanfile.py

```
def build(self):
    ...
    # we can also know where is the executable we are building
    self.run(os.path.join(self.build_folder, self.cpp.build.bindirs[0], "my_tool"))
```

We are using the `self.cpp.build.bindirs[0]` folder to locate the `my_tool`. This is a very recommended practice, especially when our layout depends on the build system. For example, when using CMake with Visual Studio, the binaries are typically built at **Release/** or **Debug/** (multiconfiguration) but on Linux or macOS, the output folder will typically be **"."**, so it is better to declare the layout `self.cpp.build.bindirs` following that logic and then just access the correct path if we need to know where the resulting files of our build are. If you check the *cmake_layout()*, you can see that the predefined `cmake_layout` is doing exactly that when using a multiconfiguration build system.

So, now we can run the conan local methods without taking much care of the directories where the files are or the build files should be, because everything is declared in the layout:

```
# This will write the toolchains and generator files from the dependencies to cmake-
↪build-debug/generators
$ conan install . -if=my_install -s build_type=Debug

# In case we needed it (not the case as we don't have a source() method), this would␣
↪fetch the sources to the ./src folder
$ conan source . -if=my_install

# This will build the project using the declared source folder and cmake-build-debug as␣
↪the build folder
$ conan build . -if=my_install
```

Note: Maybe you are wondering why the **install folder** is not parametrized and has to be specified with the `-if` argument. Currently, Conan generates several files like the `graph_info.json` and the `conanbuildinfo.txt` that are read to restore the configuration saved (settings, options, etc) to be applied in the local commands. That configuration is needed before running the `layout()` method because the folders might depend on the settings like in the previous example. It is a kind of a chicken-egg issue. In Conan 2.0, likely, the configuration won't be stored, and the local methods like **`conan build .`** will compute the graph from arguments (–profile, -s, -o...) and won't need the `--if` argument anymore, being always trivial to run.

Our current folder now looks like this:

```
<my_project_folder>
├── conanfile.py
├── src
│   ├── CMakeLists.txt
│   ├── hello.cpp
│   ├── my_tool.cpp
│   └── include
│       └── hello.h
└── cmake-build-debug
    ├── libsay.a
    └── bin
        └── my_tool
```

We could put the package in editable mode and other packages that require it would consume it in a completely trans-

parent way, even locating the correct **Release/Debug** artifacts.

```
$ conan editable add . say/0.1
```

Note: When working with editable packages, the information set in `self.cpp.source` and `self.cpp.build` will be merged with the information set in `self.cpp.package` so that we don't have to declare again something like `self.cpp.build.libs = ["say"]` that is the same for the consumers, independently of whether the package is in editable mode or not.

And of course, we can run also a `conan create` command. When the `build(self)` method is run in the conan cache, it is also able to locate the `my_tool` correctly, because it is using the same `folders.build`:

Listing 8: .conan/data/<some_cache_folder>

```
├── source
│   └── src
│       ├── CMakeLists.txt
│       ├── hello.cpp
│       ├── my_tool.cpp
│       └── include
│           └── hello.h
├── build
│   └── cmake-build-debug
│       ├── say.a
│       └── bin
│           └── my_app
└── package
    ├── lib
    │   └── say.a
    ├── bin
    │   └── my_app
    └── include
        └── hello.h
```

Warning: The `conan package` local command has been disabled (will raise an exception) when the `layout()` method is declared. If the package can be consumed "locally" in a handy way, the use case for the `conan package` method is only testing that the method is correctly coded, but that can also be done with the `conan export-pkg` method. Thus, as part of the migration to Conan 2.0, the `conan package` method will disappear.

9.2.4 Example: export_sources_folder

If we have this project, intended to create a package for a third-party library which code is located externally:

```
├── conanfile.py
├── patches
│   └── mypatch
└── CMakeLists.txt
```

The `conanfile.py` would look like this:

```
import os
from conan import ConanFile


class Pkg(ConanFile):
    name = "pkg"
    version = "0.1"
    exports_sources = "CMakeLists.txt", "patches*"

    def layout(self):
        self.folders.source = "src"

    def source(self):
        # we are inside a "src" subfolder, as defined by layout
        # download something, that will be inside the "src" subfolder
        # access to paches and CMakeLists, to apply them, replace files is done with:
        mypatch_path = os.path.join(self.export_sources_folder, "patches/mypatch")
        cmake_path = os.path.join(self.export_sources_folder, "CMakeLists.txt")
        # patching, replacing, happens here

    def build(self):
        # If necessary, the build() method also has access to the export_sources_folder
        # for example if patching happens in build() instead of source()
        cmake_path = os.path.join(self.export_sources_folder, "CMakeLists.txt")
```

We can see that the `ConanFile.export_sources_folder` can provide access to the root folder of the sources:

- Locally it will be the folder where the `conanfile.py` lives
- In the cache it will be the "source" folder, that will contain a copy of `CMakeLists.txt` and `patches`, while the "source/src" folder will contain the actual downloaded sources.

9.2.5 Example: conanfile in subfolder

If we have this project, intended to package the code that is in the same repo as the `conanfile.py`, but the `conanfile.py` is not in the root of the project:

```
├── CMakeLists.txt
└── conan
    └── conanfile.py
```

The `conanfile.py` would look like this:

```
import os
from conan import ConanFile
from conan.tools.files import load, copy


class Pkg(ConanFile):
    name = "pkg"
    version = "0.1"

    def layout(self):
```

(continues on next page)

(continued from previous page)

```
        # The root of the project is one level above
        self.folders.root = ".."
        # The source of the project (the root CMakeLists.txt) is the source folder
        self.folders.source = "."
        self.folders.build = "build"

    def export_sources(self):
        # The path of the CMakeLists.txt we want to export is one level above
        folder = os.path.join(self.recipe_folder, "..")
        copy(self, "*.txt", folder, self.export_sources_folder)

    def source(self):
        # we can see that the CMakeLists.txt is inside the source folder
        cmake = load(self, "CMakeLists.txt")

    def build(self):
        # The build() method can also access the CMakeLists.txt in the source folder
        path = os.path.join(self.source_folder, "CMakeLists.txt")
        cmake = load(self, path)
```

9.3 Packages in editable mode

Important: This is a **tutorial** section. You are encouraged to execute these commands. Some of the features used in this section are **experimental**, like `layout()` or `CMakeToolchain`, and they might change in future releases. Check the *reference section* for more information.

When working in big projects with several functionalities interconnected it is recommended to avoid the one-and-only huge project approach in favor of several libraries, each one specialized in a set of common tasks, even maintained by dedicated teams. This approach helps to isolate and reusing code helps with compiling times and reduces the likelihood of including files that not correspond to the API of the required library.

Nevertheless, in some case, it is useful to work in several libraries at the same time and see how the changes in one of them are propagated to the others. With the normal flow, for every source change, it is necessary to do `conan create` or `conan export-pkg` to put the package in the cache and make it available to consumers.

With the editable packages, you can tell Conan where to find the headers and the artifacts ready for consumption in your local working directory. There is no need to package.

Let's see this feature over a practical example, the code can be found in the examples repository:

```
$ git clone https://github.com/conan-io/examples.git
$ cd examples/features/editable/cmake
```

There are 2 folders inside this project:

- A "say" folder containing a fully fledge package, with its `conanfile.py`, its source code.
- A "hello" folder containing a simple consumer project with a `conanfile.txt` and its source code, which depends on the `say/0.1@user/testing` requirement.

The goal is to be able to build the "hello" project, without actually having the `say/0.1@user/testing` package in the cache, but directly in this project folder.

9.3.1 Put a package in editable mode

To avoid creating the package `say/0.1@user/channel` in the cache for every change, we are going to put that package in editable mode, creating **a link from the reference in the cache to the local working directory**:

```
$ conan editable add say say/0.1@user/channel
$ conan editable list
say/0.1@user/channel
    Path: ...
```

That is it. Now, every usage of `say/0.1@user/channel`, by any other Conan package or project, will be redirected to the `examples/features/editable/cmake/say` user folder instead of using the package from the conan cache.

Note that the key of editable packages is a correct definition of the `layout()` of the package. In this example, the `say conanfile.py` recipe is using the predefined `cmake_layout()` which defines the typical CMake project layout, which can be different in the different platforms. Take also into account that only using the new build system integrations like `CMakeDeps` and `CMakeToolchain` will correctly follow the layout definition.

Now the `say/0.1@user/channel` package is in editable mode, lets build it locally:

```
$ cd say

# windows, we will build 2 configurations to show multi-config
$ conan install . -s build_type=Release
$ conan install . -s build_type=Debug
$ mkdir build && cd build
$ cmake .. -DCMAKE_TOOLCHAIN_FILE=conan/conan_toolchain.cmake
$ cmake --build . --config Release
$ cmake --build . --config Debug

# Linux, we will only build 1 configuration
$ conan install .
$ mkdir cmake-build-release && cd cmake-build-release
$ cmake .. -DCMAKE_BUILD_TYPE=Release -DCMAKE_TOOLCHAIN_FILE=conan/conan_toolchain.cmake
$ cmake --build .
```

9.3.2 Using a package in editable mode

Consuming a package in editable mode is transparent from the consumer perspective. In this case we can build the `hello` application as usual:

```
$ cd ../../hello

# windows, we will build 2 configurations to show multi-config
$ conan install . -s build_type=Release
$ conan install . -s build_type=Debug
$ mkdir build && cd build
$ cmake .. -DCMAKE_TOOLCHAIN_FILE=conan_toolchain.cmake
$ cmake --build . --config Release
$ cmake --build . --config Debug
$ Release\hello.exe
say/0.1: Hello World Release!
$ Debug\hello.exe
```

(continues on next page)

(continued from previous page)

```
say/0.1: Hello World Debug!

# Linux, we will only build 1 configuration
$ conan install .
$ mkdir cmake-build-release && cd cmake-build-release
$ cmake .. -DCMAKE_BUILD_TYPE=Release -DCMAKE_TOOLCHAIN_FILE=conan_toolchain.cmake
$ cmake --build .
$ ./hello
say/0.1: Hello World Release!
```

9.3.3 Working with editable packages

Once the above steps have been done, we can basically work with our build system or IDE, no Conan involved, and do changes in the editable packages and have those changes used by the consumers directly. Lets see it, lets start by doing a change in the `say` source code:

```
$ cd ../../say
# Edit src/say.cpp and change the error message from "Hello" to "Bye"

# windows, we will build 2 configurations to show multi-config
$ cd build
$ cmake --build . --config Release
$ cmake --build . --config Debug

# Linux, we will only build 1 configuration
$ cd cmake-build-release
$ cmake --build .
```

And build and run the "hello" project:

```
$ cd ../../hello

# windows,
$ cd build
$ cmake --build . --config Release
$ cmake --build . --config Debug
$ Release\hello.exe
say/0.1: Bye World Release!
$ Debug\hello.exe
say/0.1: Bye World Debug!

# Linux
$ cd cmake-build-release
$ cmake --build .
$ ./hello
say/0.1: Bye World Release!
```

In that way, it is possible to be developing both the `say` library and the `hello` application, at the same time, without any Conan command. If you had both open in the IDE, it would be just building one after the other.

Note: When a package is in editable mode, most of the commands will not work. It is not possible to **`conan upload`**,

conan export or **conan create** when a package is in editable mode.

9.3.4 Revert the editable mode

In order to revert the editable mode just remove the link using:

```
$ conan editable remove say/0.1@user/channel
```

It will remove the link (the local directory won't be affected) and all the packages consuming this requirement will get it from the cache again.

Warning: Packages that are built consuming an editable package in its graph upstreams can generate binaries and packages incompatible with the released version of the editable package. Avoid uploading these packages without re-creating them with the in-cache version of all the libraries.

9.4 Workspaces

Warning: This is an **experimental** feature. This is actually a preview of the feature, with the main goal of receiving feedbacks and improving it. Consider the file formats, commands and flows to be unstable and subject to changes in the next releases.

Sometimes, it is necessary to work simultaneously on more than one package. In theory, each package should be a "work unit", and developers should be able to work on them in isolation. But sometimes, some changes require modifications in more than one package at the same time. The local development flow can help, but it still requires using **export-pkg** to put the artifacts in the local cache, where other packages under development will consume them.

The Conan workspaces allow to have more than one package in user folders, and have them directly use other packages from user folders without needing to put them in the local cache. Furthermore, they enable incremental builds on large projects containing multiple packages.

Lets introduce them with a practical example; the code can be found in the conan examples repository:

```
$ git clone https://github.com/conan-io/examples.git
$ cd features/workspace/cmake
```

Note that this folder contains two files *conanws_gcc.yml* and *conanws_vs.yml*, for gcc (Makefiles, single-configuration build environments) and for Visual Studio (MSBuild, multi-configuration build environment), respectively.

9.4.1 Conan workspace definition

Workspaces are defined in a yaml file, with any user defined name. Its structure is:

```
editables:
    say/0.1@user/testing:
        path: say
    hello/0.1@user/testing:
        path: hello
```

(continues on next page)

(continued from previous page)

```
    chat/0.1@user/testing:
        path: chat
layout: layout_gcc
workspace_generator: cmake
root: chat/0.1@user/testing
```

The first section `editables` defines the mapping between package references and relative paths. Each one is equivalent to a *conan editable add* command (Do NOT do this – it is not necessary. It will be automatically done later. Just to understand the behavior):

```
$ conan editable add say say/0.1@user/testing --layout=layout_gcc
$ conan editable add hello hello/0.1@user/testing --layout=layout_gcc
$ conan editable add chat chat/0.1@user/testing --layout=layout_gcc
```

The main difference is that this *Editable* state is only temporary for this workspace. It doesn't affect other projects or packages, which can still consume these say, hello, chat packages from the local cache.

Note that the `layout: layout_gcc` declaration in the workspace affects all the packages. It is also possible to define a different layout per package, as:

```
editables:
    say/0.1@user/testing:
        path: say
        layout: custom_say_layout
```

Layout files are explained in *Editable layout files* and in the *Packages in editable mode* sections.

The `workspace_generator` defines the file that will be generated for the top project. The only supported value so far is `cmake` and it will generate a *conanworkspace.cmake* file that looks like:

```
set(PACKAGE_say_SRC "<path>/examples/workspace/cmake/say/src")
set(PACKAGE_say_BUILD "<path>/examples/workspace/cmake/say/build/Debug")
set(PACKAGE_hello_SRC "<path>/examples/workspace/cmake/hello/src")
set(PACKAGE_hello_BUILD "<path>/examples/workspace/cmake/hello/build/Debug")
set(PACKAGE_chat_SRC "<path>/examples/workspace/cmake/chat/src")
set(PACKAGE_chat_BUILD "<path>/examples/workspace/cmake/chat/build/Debug")

macro(conan_workspace_subdirectories)
    add_subdirectory(${PACKAGE_say_SRC} ${PACKAGE_say_BUILD})
    add_subdirectory(${PACKAGE_hello_SRC} ${PACKAGE_hello_BUILD})
    add_subdirectory(${PACKAGE_chat_SRC} ${PACKAGE_chat_BUILD})
endmacro()
```

This file can be included in your user-defined *CMakeLists.txt* (this file is not generated). Here you can see the *CMakeLists.txt* used in this project:

```
cmake_minimum_required(VERSION 3.0)

project(WorkspaceProject)

include(${CMAKE_BINARY_DIR}/conanworkspace.cmake)
conan_workspace_subdirectories()
```

The `root: chat/0.1@user/testing` defines which is the consumer node of the graph, typically some kind of executable. You can provide a comma separated list of references, as a string, or a yaml list (abbreviated or full as

yaml items). All the root nodes will be in the same dependency graph, leading to conflicts if they depend on different versions of the same library, as in any other Conan command.

```
editables:
    say/0.1@user/testing:
        path: say
    hello/0.1@user/testing:
        path: hello
    chat/0.1@user/testing:
        path: chat

root: chat/0.1@user/testing, say/0.1@user/testing
# or
root: ["helloa/0.1@lasote/stable", "hellob/0.1@lasote/stable"]
# or
root:
    - helloa/0.1@lasote/stable
    - hellob/0.1@lasote/stable
```

9.4.2 Single configuration build environments

There are some build systems, like Make, that require the developer to manage different configurations in different build folders, and switch between folders to change configuration. The file described above is *conan_gcc.yml* file, which defines a Conan workspace that works for a CMake based project for MinGW/Unix Makefiles gcc environments (working for apple-clang or clang would be very similar, if not identical).

Lets use it to install this workspace:

```
$ mkdir build_release && cd build_release
$ conan workspace install ../conanws_gcc.yml --profile=my_profile
```

Here we assume that you have a `my_profile` profile defined which would use a single-configuration build system (like Makefiles). The example is tested with gcc in Linux, but working with apple-clang with Makefiles would be the same). You should see something like:

```
Configuration:
[settings]
...
build_type=Release
compiler=gcc
compiler.libcxx=libstdc++
compiler.version=4.9
...

Requirements
    chat/0.1@user/testing from user folder - Editable
    hello/0.1@user/testing from user folder - Editable
    say/0.1@user/testing from user folder - Editable
Packages
    chat/0.1@user/testing:df2c4f4725219597d44b7eab2ea5c8680abd57f9 - Editable
    hello/0.1@user/testing:b0e473ad8697d6069797b921517d628bba8b5901 - Editable
    say/0.1@user/testing:80faec7955dcba29246085ff8d64a765db3b414f - Editable
```

(continues on next page)

(continued from previous page)

```
say/0.1@user/testing: Generator cmake created conanbuildinfo.cmake
...
hello/0.1@user/testing: Generator cmake created conanbuildinfo.cmake
...
chat/0.1@user/testing: Generator cmake created conanbuildinfo.cmake
...
```

These *conanbuildinfo.cmake* files have been created in each package *build/Release* folder, as defined by the *layout_gcc* file:

```
# This helps to define the location of CMakeLists.txt within package
[source_folder]
src

# This defines where the conanbuildinfo.cmake will be written to
[build_folder]
build/{{settings.build_type}}
```

Now we can configure and build our project as usual:

```
$ cmake .. -G "Unix Makefiles" -DCMAKE_BUILD_TYPE=Release
$ cmake --build . # or just $ make
$ ../chat/build/Release/app
Release: Hello World!
Release: Hello World!
Release: Hello World!
```

Now, go do a change in some of the packages, for example the "say" one, and rebuild. See how it does an incremental build (fast).

Note that nothing will really be installed in the local cache, all the dependencies are resolved locally:

```
$ conan search say
There are no packages matching the 'say' pattern
```

Note: The package *conanfile.py* recipes do not contain anything special, they are standard recipes. But the packages *CMakeLists.txt* have defined the following:

```
conan_basic_setup(NO_OUTPUT_DIRS)
```

This is because the default `conan_basic_setup()` does define output directories for artifacts such as *bin*, *lib*, etc, which is not what the local project layout expects. You need to check and make sure that your build scripts and recipe matches both the expected local layout (as defined in layout files), and the recipe `package()` method logic.

Building for debug mode is done in its own folder:

```
$ cd .. && mkdir build_debug && cd build_debug
$ conan workspace install ../conanws_gcc.yml --profile=my_gcc_profile -s build_type=Debug
$ cmake .. -G "Unix Makefiles" -DCMAKE_BUILD_TYPE=Debug
$ cmake --build . # or just $ make
$ ../chat/build/Debug/app
Debug: Bye World!
```

(continues on next page)

(continued from previous page)

```
Debug: Bye World!
Debug: Bye World!
```

9.4.3 Multi configuration build environments

Some build systems, like Visual Studio (MSBuild), use "multi-configuration" environments. That is, even if the project is configured just once you can switch between different configurations (like Debug/Release) directly in the IDE and build there.

The above example uses the Conan `cmake` generator, that creates a single *conanbuildinfo.cmake* file. This is not a problem if we have our configurations built in different folders. Each one will contain its own *conanbuildinfo.cmake*. For Visual Studio that means that if we wanted to switch from Debug<->Release, we should issue a new `conan workspace install` command with the right `-s build_type` and do a clean build, in order to get the right dependencies.

Conan has the *cmake_multi generator* generator, that allows this direct switch of Debug/Release configuration in the IDE. The *conanfile.py* recipes they have defined the `cmake` generator, so the first step is to override that in our *conanws_vs.yml* file:

```
editables:
say/0.1@user/testing:
    path: say
hello/0.1@user/testing:
    path: hello
chat/0.1@user/testing:
    path: chat
layout: layout_vs
generators: cmake_multi
workspace_generator: cmake
root: chat/0.1@user/testing
```

Note the `generators: cmake_multi` line, that will define the generators to be used by our workspace packages. Also, our *CMakeLists.txt* should take into account that now we won't have a *conanbuildinfo.cmake* file, but a *conanbuildinfo_multi.cmake* file. See for example the *hello/src/CMakeLists.txt* file:

```
project(Hello)

if(EXISTS ${CMAKE_CURRENT_BINARY_DIR}/conanbuildinfo_multi.cmake)
    include(${CMAKE_CURRENT_BINARY_DIR}/conanbuildinfo_multi.cmake)
else()
    include(${CMAKE_CURRENT_BINARY_DIR}/conanbuildinfo.cmake)
endif()

conan_basic_setup(NO_OUTPUT_DIRS)

add_library(hello hello.cpp)
conan_target_link_libraries(hello)
```

The last `conan_target_link_libraries(hello)` is a helper that does the right linking with Debug/Release libraries (also works when using cmake targets).

Make sure to install both Debug and Release configurations straight ahead, if we want to later switch between them in the IDE:

```
$ mkdir build && cd build
$ conan workspace install ../conanws_vs.yml
$ conan workspace install ../conanws_vs.yml -s build_type=Debug
$ cmake .. -G "Visual Studio 15 Win64"
```

With those commands you will get a Visual Studio solution, that you can open, select the *app* executable as StartUp project, and start building, executing, debugging, switching from Debug and Release configurations freely from the IDE, without needing to issue further Conan commands.

You can check in the project folders, how the following files have been generated:

```
hello
  |- build
        | - conanbuildinfo_multi.cmake
        | - conanbuildinfo_release.cmake
        | - conanbuildinfo_debug.cmake
```

Note that they are not located in *build/Release* and *build/Debug* subfolders; that is because of the multi-config environment. To account for that the *layout_vs* define the `[build_folder]` not as `build/{settings.build_type}` but just as:

```
[build_folder]
build
```

9.4.4 Out of source builds

The above examples are using a build folder in-source of the packages in editable mode. It is possible to define out-of-source builds layouts, using relative paths and the `reference` argument. The following layout definition could be used to locate the build artifacts of an editable package in a sibling `build/<package-name>` folder:

```
[build_folder]
../build/{{reference.name}}/{{settings.build_type}}

[includedirs]
src

[libdirs]
../build/{{reference.name}}/{{settings.build_type}}/lib
```

9.4.5 Notes

Note that this way of developing packages shouldn't be used to create the final packages (you could try to use **`conan export-pkg`**), but instead, a full package creation with **`conan create`** (best in CI) is recommended.

So far, only the CMake super-project generator is implemented. A Visual Studio one is being considered, and seems feasible, but not yet available.

Important: We really want your feedback. Please submit any issues to https://github.com/conan-io/conan/issues with any suggestion, problem, idea, and using [workspaces] prefix in the issue title.

CHAPTER

TEN

PACKAGE APPS AND DEVTOOLS

With conan it is possible to package and deploy applications. It is also possible that these applications are also dev-tools, like compilers (e.g. MinGW), or build systems (e.g. CMake).

This section describes how to package and run executables, and also how to package dev-tools. Also, how to apply applications like dev-tools or even libraries (like testing frameworks) to other packages to build them from sources: *Tool requirements*

10.1 Running and deploying packages

Executables and applications including shared libraries can also be distributed, deployed and run with Conan. This might have some advantages compared to deploying with other systems:

- A unified development and distribution tool, for all systems and platforms.
- Manage any number of different deployment configurations in the same way you manage them for development.
- Use a Conan server remote to store all your applications and runtimes for all Operating Systems, platforms and targets.

There are different approaches:

10.1.1 Using virtual environments

We can create a package that contains an executable, for example from the default package template created by **conan new**:

```
$ conan new hello/0.1
```

The source code used contains an executable called `greet`, but it is not packaged by default. Let's modify the recipe `package()` method to also package the executable:

```
def package(self):
    self.copy("*greet*", src="bin", dst="bin", keep_path=False)
```

Now we create the package as usual, but if we try to run the executable it won't be found:

```
$ conan create . user/testing
...
hello/0.1@user/testing package(): Copied 1 '.h' files: hello.h
hello/0.1@user/testing package(): Copied 1 '.exe' files: greet.exe
hello/0.1@user/testing package(): Copied 1 '.lib' files: hello.lib
```

(continues on next page)

(continued from previous page)

```
$ greet
> ... not found...
```

By default, Conan does not modify the environment, it will just create the package in the local cache, and that is not in the system PATH, so the `greet` executable is not found.

The `virtualrunenv` generator generates files that add the package's default binary locations to the necessary paths:

- It adds the dependencies `lib` subfolder to the `DYLD_LIBRARY_PATH` environment variable (for OSX shared libraries)
- It adds the dependencies `lib` subfolder to the `LD_LIBRARY_PATH` environment variable (for Linux shared libraries)
- It adds the dependencies `bin` subfolder to the `PATH` environment variable (for executables)

So if we install the package, specifying such `virtualrunenv` like:

```
$ conan install hello/0.1@user/testing -g virtualrunenv
```

This will generate a few files that can be called to activate and deactivate the required environment variables

```
$ activate_run.sh # $ source activate_run.sh in Unix/Linux
$ greet
> Hello World Release!
$ deactivate_run.sh # $ source deactivate_run.sh in Unix/Linux
```

10.1.2 Imports

It is possible to define a custom conanfile (either *.txt* or *.py*), with an `imports()` section, that can retrieve from local cache the desired files. This approach requires a user conanfile.

For more details see the example below *runtime packages*.

10.1.3 Deployable packages

With the `deploy()` method, a package can specify which files and artifacts to copy to user space or to other locations in the system. Let's modify the example recipe adding the `deploy()` method:

```
def deploy(self):
    self.copy("*", dst="bin", src="bin")
```

And run **conan create**

```
$ conan create . user/testing
```

With that method in our package recipe, it will copy the executable when installed directly:

```
$ conan install hello/0.1@user/testing
...
> hello/0.1@user/testing deploy(): Copied 1 '.exe' files: greet.exe
$ bin\greet.exe
> Hello World Release!
```

The deploy will create a *deploy_manifest.txt* file with the files that have been deployed.

Sometimes it is useful to adjust the package ID of the deployable package in order to deploy it regardless of the compiler it was compiled with:

```
def package_id(self):
    del self.info.settings.compiler
```

See also:

Read more about the *deploy()* method.

10.1.4 Using the *deploy* generator

The *deploy generator* is used to have all the dependencies of an application copied into a single place. Then all the files can be repackaged into the distribution format of choice.

For instance, if the application depends on boost, we may not know that it also requires many other 3rt-party libraries, such as zlib, bzip2, lzma, zstd, iconv, etc.

```
$ conan install . -g deploy
```

This helps to collect all the dependencies into a single place, moving them out of the Conan cache.

10.1.5 Using the *json* generator

A more advanced approach is to use the *json generator*. This generator works in a similar fashion as the *deploy* one, although it doesn't copy the files to a directory. Instead, it generates a JSON file with all the information about the dependencies including the location of the files in the Conan cache.

```
$ conan install . -g json
```

The *conanbuildinfo.json* file produced, is fully machine-readable and could be used by scripts to prepare the files and recreate the appropriate format for distribution. The following code shows how to read the library and binary directories from the *conanbuildinfo.json*:

```
import os
import json

data = json.load(open("conanbuildinfo.json"))

dep_lib_dirs = dict()
dep_bin_dirs = dict()

for dep in data["dependencies"]:
    root = dep["rootpath"]
    lib_paths = dep["lib_paths"]
    bin_paths = dep["bin_paths"]

    for lib_path in lib_paths:
        if os.listdir(lib_path):
            lib_dir = os.path.relpath(lib_path, root)
            dep_lib_dirs[lib_path] = lib_dir
    for bin_path in bin_paths:
```

(continues on next page)

(continued from previous page)

```
        if os.listdir(bin_path):
            bin_dir = os.path.relpath(bin_path, root)
            dep_bin_dirs[bin_path] = bin_dir
```

While with the *deploy* generator, all the files were copied into a folder. The advantage with the *json* one is that you have fine-grained control over the files and those can be directly copied to the desired layout.

In that sense, the script above could be easily modified to apply some sort of filtering (e.g. to copy only shared libraries, and omit any static libraries or auxiliary files such as pkg-config .pc files).

Additionally, you could also write a simple startup script for your application with the extracted information like this:

```
executable = "MyApp"  # just an example
varname = "$APPDIR"

def _format_dirs(dirs):
    return ":".join(["%s/%s" % (varname, d) for d in dirs])

path = _format_dirs(set(dep_bin_dirs.values()))
ld_library_path = _format_dirs(set(dep_lib_dirs.values()))
exe = varname + "/" + executable

content = """#!/usr/bin/env bash
set -ex
export PATH=$PATH:{path}
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:{ld_library_path}
pushd $(dirname {exe})
$(basename {exe})
popd
""".format(path=path,
       ld_library_path=ld_library_path,
       exe=exe)
```

10.1.6 Running from packages

If a dependency has an executable that we want to run in the conanfile, it can be done directly in code using the `run_environment=True` argument. It internally uses a `RunEnvironment()` helper. For example, if we want to execute the **greet** app while building the `consumer` package:

```
from conans import ConanFile, tools, RunEnvironment

class ConsumerConan(ConanFile):
    name = "consumer"
    version = "0.1"
    settings = "os", "compiler", "build_type", "arch"
    requires = "hello/0.1@user/testing"

    def build(self):
        self.run("greet", run_environment=True)
```

Now run **conan install** and **conan build** for this consumer recipe:

```
$ conan install . && conan build .
...
Project: Running build()
Hello World Release!
```

Instead of using the environment, it is also possible to explicitly access the path of the dependencies:

```
def build(self):
    path = os.path.join(self.deps_cpp_info["hello"].rootpath, "bin")
    self.run(["%s/greet" % path])
```

Note that this might not be enough if shared libraries exist. Using the `run_environment=True` helper above is a more complete solution.

This example also demonstrates using a list to specify the command to run. This bypasses the system shell and works correctly even when `path` contains special characters like whitespace or quotes that would otherwise be interpreted by the shell. However, it also means that substituting environment variables or the output from other commands which are normally done by the shell won't work when using this method. Specify your command using a plain string as shown above when you require this functionality.

Finally, there is another approach: the package containing the executable can add its *bin* folder directly to the `PATH`. In this case the **Hello** package conanfile would contain:

```
def package_info(self):
    self.cpp_info.libs = ["hello"]
    self.env_info.PATH = os.path.join(self.package_folder, "bin")
```

We may also define `DYLD_LIBRARY_PATH` and `LD_LIBRARY_PATH` if they are required for the executable.

The consumer package is simple, as the `PATH` environment variable contains the `greet` executable:

```
def build(self):
    self.run("greet")
```

Read the *next section* for a more comprenhensive explanation about using packaged executables in your recipe methods.

10.1.7 Runtime packages and re-packaging

It is possible to create packages that contain only runtime binaries, getting rid of all build-time dependencies. If we want to create a package from the above "hello" one, but only containing the executable (remember that the above package also contains a library, and the headers), we could do:

```
from conans import ConanFile

class HellorunConan(ConanFile):
    name = "hello_run"
    version = "0.1"
    tool_requires = "hello/0.1@user/testing"
    keep_imports = True

    def imports(self):
        self.copy("greet*", src="bin", dst="bin")
```

(continues on next page)

(continued from previous page)

```
    def package(self):
        self.copy("*")
```

This recipe has the following characteristics:

- It includes the `hello/0.1@user/testing` package as `tool_requires`. That means that it will be used to build this *hello_run* package, but once the *hello_run* package is built, it will not be necessary to retrieve it.
- It is using `imports()` to copy from the dependencies, in this case, the executable
- It is using the `keep_imports` attribute to define that imported artifacts during the `build()` step (which is not define, then using the default empty one), are kept and not removed after build
- The `package()` method packages the imported artifacts that will be created in the build folder.

To create and upload this package to a remote:

```
$ conan create . user/testing
$ conan upload hello_run* --all -r=my-remote
```

Installing and running this package can be done using any of the methods presented above. For example:

```
$ conan install hello_run/0.1@user/testing -g virtualrunenv
# You can specify the remote with -r=my-remote
# It will not install hello/0.1@...
$ activate_run.sh # $ source activate_run.sh in Unix/Linux
$ greet
> Hello World Release!
$ deactivate_run.sh # $ source deactivate_run.sh in Unix/Linux
```

Deployment challenges

When deploying a C/C++ application there are some specific challenges that have to be solved when distributing your application. Here you will find the most usual ones and some recommendations to overcome them.

The C standard library

A common challenge for all the applications no matter if they are written in pure C or in C++ is the dependency on C standard library. The most wide-spread variant of this library is GNU C library or just glibc.

Glibc is not a just C standard library, as it provides:

- C functions (like `malloc()`, `sin()`, etc.) for various language standards, including C99.
- POSIX functions (like posix threads in the `pthread` library).
- BSD functions (like BSD sockets).
- Wrappers for OS-specific APIs (like Linux system calls)

Even if your application doesn't use directly any of these functions, they are often used by other libraries, so, in practice, it's almost always in actual use.

There are other implementations of the C standard library that present the same challenge, such as newlib or musl, used for embedded development.

To illustrate the problem, a simple hello-world application compiled in a modern Ubuntu distribution will give the following error when it is run in a Centos 6 one:

```
$ /hello
/hello: /lib64/libc.so.6: version 'GLIBC_2.14' not found (required by /hello)
```

This is because the versions of the `glibc` are different between those Linux distributions.

There are several solutions to this problem:

- LibcWrapGenerator
- glibc_version_header
- bingcc

Some people also advice to use static of glibc, but it's strongly discouraged. One of the reasons is that newer glibc might be using syscalls that are not available in the previous versions, so it will randomly fail in runtime, which is much harder to debug (the situation about system calls is described below).

It's possible to model either `glibc` version or Linux distribution name in Conan by defining custom Conan sub-setting in the *settings.yml* file (check out sections *Adding new settings* and *Adding new sub-settings*). The process will be similar to:

- Define new sub-setting, for instance *os.distro*, as explained in the section *Adding new sub-settings*.
- Define compatibility mode, as explained by sections *package_id()* and *build_id()* (e.g. you may consider some `Ubuntu` and `Debian` packages to be compatible with each other)
- Generate different packages for each distribution.
- Generate deployable artifacts for each distribution.

C++ standard library

Usually, the default C++ standard library is libstdc++, but libc++ and stlport are other well-known implementations.

Similarly to the standard C library *glibc*, running the application linked with libstdc++ in the older system may result in an error:

```
$ /hello
/hello: /usr/lib64/libstdc++.so.6: version 'GLIBCXX_3.4.21' not found (required by /
↪hello)
/hello: /usr/lib64/libstdc++.so.6: version 'GLIBCXX_3.4.26' not found (required by /
↪hello)
```

Fortunately, this is much easier to address by just adding `-static-libstdc++` compiler flag. Unlike C runtime, C++ runtime can be linked statically safely, because it doesn't use system calls directly, but instead relies on `libc` to provide required wrappers.

Compiler runtime

Besides C and C++ runtime libraries, the compiler runtime libraries are also used by applications. Those libraries usually provide lower-level functions, such as compiler intrinsics or support for exception handling. Functions from these runtime libraries are rarely referenced directly in code and are mostly implicitly inserted by the compiler itself.

```
$ ldd ./a.out
libgcc_s.so.1 => /lib/x86_64-linux-gnu/libgcc_s.so.1 (0x00007f6626aee000)
```

you can avoid this kind of dependency by the using of the `-static-libgcc` compiler flag. However, it's not always sane thing to do, as there are certain situations when applications should use shared runtime. The most common is when the application wishes to throw and catch exceptions across different shared libraries. Check out the GCC manual for the detailed information.

System API (system calls)

New system calls are often introduced with new releases of Linux kernel. If the application, or 3rd-party libraries, want to take advantage of these new features, they sometimes directly refer to such system calls (instead of using wrappers provided by `glibc`).

As a result, if the application was compiled on a machine with a newer kernel and build system used to auto-detect available system calls, it may fail to execute properly on machines with older kernels.

The solution is to either use a build machine with lowest supported kernel, or model supported operation system (just like in case of `glibc`). Check out sections *Adding new settings* and *Adding new sub-settings* to get a piece of information on how to model distribution in conan settings.

10.2 Creating conan packages to install dev tools

One of the most useful features of Conan is to package executables like compilers or build tools and distribute them in a controlled way to the team of developers. This way Conan helps not only with the graph of dependencies of the application itself, but also with all the ecosystem needed to generate the project, making it really easy to control everything involved in the deployed application.

Those tools need to run in the working machine (the `build` machine) regardless of the `host` platform where the generated binaries will run. If those platforms are different, we are cross building software.

In this section we cope with the general scenario where a library requires other tools to compile that are also packaged with Conan. Read this section first, and get more information specific to cross compiling in the dedicated section of the docs: *Cross building*.

Note: Conan v1.24 introduced a new feature to declare a full profile for the `build` and the `host` machine, it is the preferred way to deal with this scenario. Older versions should rely on the deprecated settings `os_build` and `arch_build`. There is a small section below about those settings, for a full explanation read the docs matching your Conan client.

A Conan package for a tool is like any other package with an executable. Here it is a recipe for packaging the `nasm` tool for building assembler:

```
import os
from conans import ConanFile, tools
from conans.errors import ConanInvalidConfiguration


class NasmConan(ConanFile):
    name = "nasm"
    version = "2.13.02"
    license = "BSD-2-Clause"
    url = "https://github.com/conan-io/conan-center-index"
    settings = "os", "arch"
    description="Nasm for windows. Useful as a build_require."
```

(continues on next page)

(continued from previous page)

```
    def validate(self):
        if self.settings.os != "Windows":
            raise ConanInvalidConfiguration("Only windows supported for nasm")

    @property
    def nasm_folder_name(self):
        return "nasm-%s" % self.version

    def build(self):
        suffix = "win32" if self.settings.arch == "x86" else "win64"
        nasm_zip_name = "%s-%s.zip" % (self.nasm_folder_name, suffix)
        tools.download("http://www.nasm.us/pub/nasm/releasebuilds/"
                       "%s/%s/%s" % (self.version, suffix, nasm_zip_name), nasm_zip_name)
        self.output.info("Downloading nasm: "
                         "http://www.nasm.us/pub/nasm/releasebuilds"
                         "/%s/%s/%s" % (self.version, suffix, nasm_zip_name))
        tools.unzip(nasm_zip_name)
        os.unlink(nasm_zip_name)

    def package(self):
        self.copy("*", src=self.nasm_folder_name, dst="bin", keep_path=True)
        self.copy("license*", dst="", src=self.nasm_folder_name, keep_path=False, ignore_
↪case=True)

    def package_info(self):
        self.env_info.PATH.append(os.path.join(self.package_folder, "bin"))
```

This recipe has nothing special: it doesn't declare the `compiler` and `build_type` settings because it is downloading already available binaries, and it is declaring the information for their consumers as usual in the *package_info()* method:

- The *cpp_info* is not declared, so it will take its default values: the `bindirs` will point to the `bin` folder where the `nasm.exe` executable is packaged.
- In the *env_info* attribute, it is adding the `bin` folder to the `PATH` environment variable.

This two simple declarations are enough to reuse this tool in the scenarios we are detailing below.

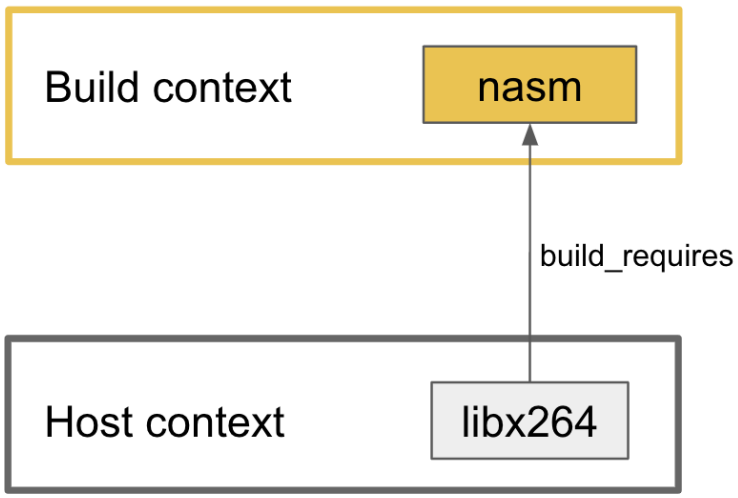
10.2.1 Using the tool packages in other recipes

Warning: This section refers to the **experimental feature** that is activated when using `--profile:build` and `--profile:host` in the command-line. It is currently under development, features can be added or removed in the following versions.

These kind of tools are not usually part of the application graph itself, they are needed only to build the library, so you should usually declare them as *tool requirements*, in the recipe itself or in a profile.

For example, there are many recipes that can take advantage of the `nasm` package we've seen above, like flac or libx264 that are already available in ConanCenter. Those recipes will take advantage of `nasm` being in the PATH to run some assembly optimizations.

```
class LibX264Conan(ConanFile):
    name = "libx264"
    ...
    tool_requires = "nasm/2.13.02"

    def build(self):
        ... # ``nasm.exe`` will be in the PATH here

    def package_info(self):
        self.cpp_info.libs = [...]
```



The consumer recipe needs only to declare the corresponding `build_require` and Conan will take care of adding the required paths to the corresponding environment variables:

```
conan create path/to/libx264 --profile:build=windows --profile:host=profile_host
```

Here we are telling Conan to create the package for the `libx264` for the `host` platform defined in the profile `profile_host` file and to use the profile `windows` for all the tool requirements that are in the `build` context. In other words: in this example we are running a Windows machine and we need a version of `nasm` compatible with this machine, so we are providing a `windows` profile for the `build` context, and we are generating the library for the `host` platform which is declared in the `profile_host` profile (read more about *tool requires context*).

Using two profiles forces Conan to make this distinction between recipes in the `build` context and those in the `host` context. It has several advantages:

- Recipes for these tools are regular recipes, no need to adapt them (before 1.24 they require special settings and some package ID customization).
- We provide a full profile for the `build` machine, so Conan is able to compile those tool requirements from sources if they are not already available.
- Conan will add to the environment not only the path to the `bin` folder, but also it will populate the `DYLD_LIBRARY_PATH` and `LD_LIBRARY_PATH` variables that are needed to find the shared libraries that tool could need during runtime.

10.2.2 Using the tool packages in your system

A different scenario is when you want to use in your system the binaries generated by Conan, to achieve this objective you can use the *virtualrunenv generator* to get your environment populated with the required variables.

For example: Working in Windows with the `nasm` package we've already defined:

1. Create a separate folder from your project, this folder will handle our global development environment.

```
$ mkdir my_cpp_environ
$ cd my_cpp_environ
```

2. Create a *conanfile.txt* file:

```
[requires]
nasm/2.13.02
# You can add more tools here

[generators]
virtualrunenv
```

3. Install them. Here it doesn't matter if you use only the `host` profile or the `build` one too because the environment that is going to be populated includes only the root of the graph and its dependencies, without any tool requirement. In any case, the `profile:host` needed is the one corresponding to the Windows machine where we are running these tests.

```
$ conan install . --profile:host=windows [--profile:build=windows]
```

4. Activate the virtual environment in your shell:

```
$ activate_run
(my_cpp_environ)$
```

5. Check that the tools are in the path:

```
(my_cpp_environ)$ nasm --version

> NASM version 2.13.02 compiled on Dec 18 2019
```

6. You can deactivate the virtual environment with the *deactivate.bat* script

```
(my_cpp_environ)$ deactivate_run
```

10.3 Tool requirements

Important: The tool requirement was formerly named "build requirement" and has been renamed to highlight that the usage of this kind of requirement must be for "tools" exclusively, not being valid for libraries to express a "private" require or other meanings.

There are some requirements that don't feel natural to add to a package recipe. For example, imagine that you had a `cmake/3.4` package in Conan. Would you add it as a requirement to the `zlib` package, so it will install cmake first in order to build `zlib`?

In short:

- There are requirements that are only needed when you need to build a package from sources, but if the binary package already exists, you don't want to install or retrieve them.
- These could be dev tools, compilers, build systems, code analyzers, testing libraries, etc.
- They can be very orthogonal to the creation of the package. It doesn't matter whether you build zlib with CMake 3.4, 3.5 or 3.6. As long as the *CMakeLists.txt* is compatible, it will produce the same final package.
- You don't want to add a lot of different versions (like those of CMake) to be able to use them to build the package. You want to easily change the requirements, without needing to edit the zlib package recipe.
- Some of them might not even be taken into account when a package like zlib is created, such as cross-compiling it to Android (in which the Android toolchain would be a tool requirement too).

Important: `tool_requires` are designed for packaging tools, utilities that only run at build-time, but are not part of the final binary code. Anything that is linked into consumer packages like all type of libraries (header only, static, shared) most likely are not `tool_requires` but regular `requires`. The only exception would be testing libraries and frameworks, as long as the tests are not included in the final package.

To address these needs Conan implements `tool_requires`.

10.3.1 Declaring tool requirements

Tool requirements can be declared in profiles, like:

Listing 1: my_profile

```
[tool_requires]
tool1/0.1@user/channel
tool2/0.1@user/channel, tool3/0.1@user/channel
*: tool4/0.1@user/channel
my_pkg*: tool5/0.1@user/channel
&: tool6/0.1@user/channel
&!: tool7/0.1@user/channel
```

Tool requirements are specified by a `pattern:`. If such pattern is not specified, it will be assumed to be `*`, i.e. to apply to all packages. Packages can be declared in different lines or by a comma separated list. In this example, `tool1`, `tool2`, `tool3` and `tool4` will be used for all packages in the dependency graph (while running **`conan install`** or **`conan create`**).

If a pattern like `my_pkg*` is specified, the declared tool requirements will only be applied to packages matching that pattern: `tool5` will not be applied to Zlib for example, but it will be applied to `my_pkg_zlib`.

The special case of a **consumer** conanfile (without name or version) it is impossible to match with a pattern, so it is handled with the special character `&`:

- `&` means apply these tool requirements to the consumer conanfile
- `&!` means apply the tool requirements to all packages except the consumer one.

Remember that the consumer conanfile is the one inside the *test_package* folder or the one referenced in the **`conan install`** command.

Tool requirements can also be specified in a package recipe, with the `tool_requires` attribute and the `build_requirements()` method:

```
class MyPkg(ConanFile):
    tool_requires = "tool_a/0.2@user/testing", "tool_b/0.2@user/testing"

    def build_requirements(self):
        # useful for example for conditional tool_requires
        # This means, if we are running on a Windows machine, require ToolWin
        if platform.system() == "Windows":
            self.tool_requires("tool_win/0.1@user/stable")
```

The above `tool_a` and `tool_b` will always be retrieved and used for building this recipe, while the `tool_win` one will only be used only in Windows.

If any tool requirement defined inside `build_requirements()` has the same package name as the one defined in the `tool_requires` attribute, the one inside the `build_requirements()` method will prevail.

As a rule of thumb, downstream defined values always override upstream dependency values. If some tool requirement is defined in the profile, it will overwrite the tool requirements defined in package recipes that have the same package name.

10.3.2 Build and Host contexts

> **Warning:** This section refers to the **experimental feature** that is activated when using `--profile:build` and `--profile:host` in the command-line. It is currently under development, features can be added or removed in the following versions.

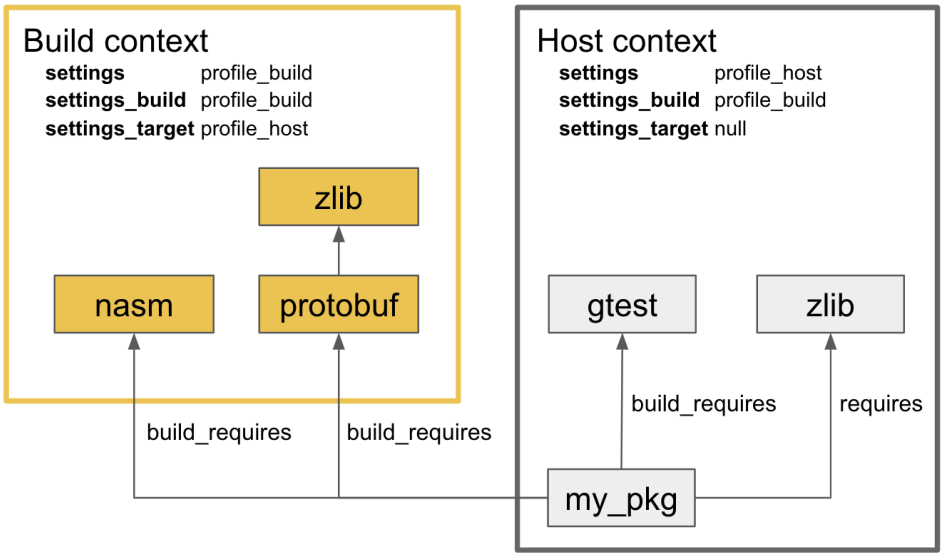
Conan v1.24 differentiates between the `build` context and the `host` context in the dependency graph (read more about the meaning of `host` and `build` platforms in the *cross building* section) **when the user supplies two profiles** to the command line using the `--profile:build` and `--profile:host` arguments:

- The **host context** is populated with the root package (the one specified in the **`conan install`** or **`conan create`** command), all its requirements and the tool requirements forced to be in the host context.
- The **build context** contains the rest of tool requirements and all of them in the profiles. This category typically includes all the *dev tools* like CMake, compilers, linkers,...

Tool requirements declared in the recipes can be forced to stay in the host context, this is needed for testing libraries that will be linked to the generated library or other executable we want to deploy to the `host` platform, for example:

```
class MyPkg(ConanFile):
    tool_requires = "nasm/2.14"  # 'build' context (nasm.exe will be available)

    def build_requirements(self):
        self.tool_requires("protobuf/3.6.1")  # 'build' context (protoc.exe will be␣
↪available)
        self.test_requires("gtest/0.1")
```

Note: The `test_requires()`, available from Conan 1.43, is equivalent to the previous `self.build_requires(, force_host_context=True)` syntax. As the later is going to dissapear in Conan 2.0, the former `test_requires()` form is recommended.



Take into account that the same package (executable or library) can appear two times in the graph, in the `host` and in the `build` context, with different package IDs. Conan will propagate the proper information to the consumers:

- Tool requirements in the `host` context will propagate like any other requirement:
  - `cpp_info`: all information will be available in the `deps_cpp_info["xxx"]` object.
  - `env_info`: won't be propagated.
  - `user_info`: will be available using the `deps_user_info["xxx"]` object.
- Tool requirements in the `build` context will propagate all the `env_info` and Conan will also populate the environment variables `DYLD_LIBRARY_PATH`, `LD_LIBRARY_PATH` and `PATH` with the corresponding information from the `cpp_info` object. All this information will be available in the `deps_env_info` object.

  Custom information declared in the `user_info` attribute will be available in the `user_info_build["xxx"]` object in the consumer *conanfile*.

Important: If no `--profile:build` is provided, all tool requirements will belong to the one and only context and they will share their dependencies with the libraries we are building. In this scenario all the tool requirements propagate `user_info`, `cpp_info` and `env_info` to the consumer's `deps_user_info`, `deps_cpp_info` and `deps_env_info`.

10.3.3 Properties of tool requirements

The behavior of `tool_requires` is the same irrespective if they are defined in the profile or if defined in the package recipe.

- They will only be retrieved and installed if some package that has to be built from sources and matches the declared pattern. Otherwise, they will not even be checked for existence.
- Options and environment variables declared in the profile as well as in the command line will affect the tool requirements for packages. In that way, you can define, for example, for the `cmake/3.16.3` package which CMake version will be installed.
- Tool requirements will be activated for matching packages, see the section above about *tool requires context* to know the information that this package will propagate to its consumers.

- Tool requirements can also be transitive. They can declare their own requirements, both normal requirements and their own build requirements. Normal logic for dependency graph resolution applies, such as conflict resolution and dependency overriding.
- Each matching pattern will produce a different dependency graph of tool requirements. These graphs are cached so that they are only computed once. If a tool requirement applies to different packages with the same configuration it will only be installed once (same behavior as normal dependencies - once they are cached locally, there is no need to retrieve or build them again).
- Tool requirements do not affect the binary package ID. If using a different tool requirement produces a different binary, you should consider adding an option or a setting to model that (if not already modeled).
- Can also use version-ranges, like `Tool/[>0.3]@user/channel`.
- Tool requirements are not listed in **conan info** nor are represented in the graph (with **conan info --graph**).

10.3.4 Example: testing framework and build tool

One example of tool requirement is a testing framework implemented as a library, another good example is a build tool used in the compile process. Let's call them `mytest_framework` and `cmake_turbo`, and imagine we already have a package available for both of them.

Tool requirements can be checked for existence (whether they've been applied) in the recipes, which can be useful for conditional logic in the recipes. In this example, we could have one recipe with the following `build()` method:

```
def build_requirements(self):
    if self.options.enable_testing:
        self.tool_requires("mytest_framework/0.1@user/channel", force_host_context=True)

def build(self):
    # Use our own 'cmake_turbo' if it is available
    use_cmake_turbo = "cmake_turbo" in self.deps_env_info.deps
    cmake_executable = "cmake_turbo" if use_cmake_turbo else None
    cmake = CMake(self, cmake_program=cmake_executable)
    cmake.configure(defs={"ENABLE_TESTING": self.options.enable_testing})
    cmake.build()
    if enable_testing:
        cmake.test()
```

And the package *CMakeLists.txt*:

```
project(PackageTest CXX)
cmake_minimum_required(VERSION 2.8.12)

include(${CMAKE_BINARY_DIR}/conanbuildinfo.cmake)
conan_basic_setup()
if(ENABLE_TESTING)
    add_executable(example test.cpp)
    target_link_libraries(example ${CONAN_LIBS})

    enable_testing()
    add_test(NAME example
             WORKING_DIRECTORY ${CMAKE_BINARY_DIR}/bin
             COMMAND example)
endif()
```

This package recipe won't retrieve the `cmake_turbo` package for normal installation:

```
$ conan install .
```

But if the following profile is defined:

Listing 2: use_cmake_turbo_profile

```
[tool_requires]
cmake_turbo/0.1@user/channel
```

then the install command will retrieve the `cmake_turbo` and use it:

```
$ conan install . --profile=use_cmake_turbo_profile
```

Although the previous line would work it is preferred to use the feature from Conan v1.24 and provide two profiles to the command line, that way the tool requirements in the `build` context won't interfer with the `host` graph if they share common requirements (see *section about dev tools*). It can also be needed if cross compiling (see *section about cross compiling*).

```
$ conan install . --profile:host=use_cmake_turbo_profile --profile:build=build_machine
```

10.3.5 Making tool_requires affect the consumers package-ID

Warning: This subsection should be considered a workaround, not a feature, and it might have other side effects, that will not be fixed as this is not recommended production code.

As discussed above, the `tool_requires` do not affect at all the package ID. As they will not be present at all when the `package_id` is computed, it cannot be part of it. It is possible that this might change in the future in Conan 2.0, but at the moment it is not. In the meantime, there is a possible workaround that might be used if this is very needed: using `python_requires` to point to the same `tool_requires` package. Something like:

```
from conans import ConanFile
class Pkg(ConanFile):
    python_requires ="tool/[>=0.0]"
    tool_requires ="tool/[>=0.0]"
```

By using this mechanism, `tool` dependency will always be used (the recipe will be fetched from servers), and the version of `tool` will be used to compute the `package_id` following the `default_python_requires_id_mode` in *conan.conf*, or the specific `self.info.python_requires.xxxx_mode()` in recipes.

10.3.6 Testing tool_requires

Warning: This is an **experimental** feature, subject to future breaking changes

Available since: 1.44.0

From Conan 1.44, it is possible to test `tool_requires` with the `test_package` functionality. In the `test_package/conanfile.py`, specify the `test_type = "explicit"` and use the variable `self.tested_reference_str` in `build_requirements()` method to explicitly require the reference as a `tool_requires` or `test_requires`:

```
from conans import ConanFile

class Pkg(ConanFile):
    test_type = "explicit"

    def build_requirements(self):
        self.test_requires(self.tested_reference_str)
```

If for some reason, it is necessary to test the same package both as a regular require and a build_require, then it is possible to specify:

```
from conans import ConanFile

class Pkg(ConanFile):
    test_type = "explicit"

    def requirements(self):
        self.requires(self.tested_reference_str)

    def build_requirements(self):
        self.test_requires(self.tested_reference_str)
```

CHAPTER

ELEVEN

VERSIONING

11.1 Introduction to versioning

11.1.1 Versioning approaches

Fixed versions

This is the standard, direct way to specify dependencies versions, with their exact version, for example in a *conanfile.py* recipe:

```
requires = "zlib/1.2.11"
```

When doing a **conan install**, it will try to fetch from the remotes exactly that *1.2.11* version.

This method is nicely explicit and deterministic, and is probably the most used one. As a possible disadvantage, it requires the consumers to explicitly modify the recipes to use updated versions, which could be tedious or difficult to scale for large projects with many dependencies, in which those dependencies are frequently modified, and it is desired to move the whole project forward to those updated dependencies.

To mitigate that issue, especially while developing the packages, you can use fixed versions with *package revisions* (see below) to resolve automatically the latest revision for a given fixed version.

Version ranges

A *conanfile* can specify a range of valid versions that could be consumed, using brackets:

```
requires = "pkg/[>1.0 <1.8]@user/stable"
```

When a **conan install** is executed, it will check in the local cache first and if not in the remotes what `pkg` versions are available and will select the latest one that satisfies the defined range.

By default, it is less deterministic, one **conan install** can resolve to `pkg/1.1` and then `pkg/1.2` is published, and a new **conan install** (by users, or CI), will automatically pick the newer 1.2 version, with different results. On the other hand it doesn't require changes to consumer recipes to upgrade to use new versions of dependencies.

It is also true that the *semver* definition that comes from other programming languages doesn't fit that well to C and C++ packages, because of different reasons, because of open source libraries that don't closely follow the semver specification, but also because of the ABI compatibility issues and compilation model that is so characteristic of C and C++ binaries.

Read more about it in *Version ranges* section.

Package alias

It is possible to define a "proxy" package that references another one, using the syntax:

```
from conans import ConanFile

class AliasConanfile(ConanFile):
    alias = "pkg/0.1@user/testing"
```

This package creation can be automatically created with the *conan alias* command, that can for example create a `pkg/latest@user/testing` alias that will be pointing to that `pkg/0.1@user/testing`. Consumers can define `requires = "pkg/latest@user/testing"` and when the graph is evaluated, it will be directly replaced by the `pkg/0.1` one. That is, the `pkg/latest` package will not appear in the dependency graph at all.

This is also less deterministic, and puts the control on the package creator side, instead of the consumer (version ranges are controlled by the consumer). Package creators can control which real versions will their consumers be using. **This is probably not the recommended way for normal dependencies versions management.**

Note: From Conan 1.39, a new **experimental** syntax for requiring alias packages has been introduced, to make explicit its usage and solve several issues with alias:

```
from conans import ConanFile

class Pkg(ConanFile):
    # Previous syntax, implicit, nothing in the reference tells it is an alias
    # requires = "pkg/latest@user/testing"
    # New experimental syntax, explicit:
    requires = "pkg/(latest)@user/testing"
```

The new `requires = "pkg/(latest)@user/testing"` comes from https://github.com/conan-io/tribe/pull/25, and is introduced in Conan 1.39 to allow getting feedback, stabilizing it, previously to make it the default in Conan 2.0 while removing the previous one.

Package revisions

Revisions are automatic internal versions to both recipes and binary packages. When revisions are enabled, when a recipe changes and it is used to create a package, a new recipe revision is generated, with the hash of the contents of the recipe. The revisioned reference of the recipe is:

```
pkg/version@user/channel#recipe_revision1
# after the change of the recipe
pkg/version@user/channel#recipe_revision2
```

A conanfile can reference a specific revision of its dependencies, but in the general case that they are not specified, it will fetch the latest revision available in the remote server:

```
[requires]
# Use the latest revision of pkg1
pkg1/version@user/channel
# use the specific revision RREV1 of pkg2
pkg2/version@user/channel#RREV1
```

Each binary package will also be revisioned. The good practice is to build each binary just once. But if for some reason, like a change in the environment, a new build of exactly the same recipe with the same code (and the same recipe revision) is fired again, a new package revision can be created. The package revision is the hash of the contents of the package (headers, libraries...), so unless deterministic builds are achieved, new package revisions will be generated.

In general revisions are not intended to be defined explicitly in conanfiles, although they can for specific purposes like debugging.

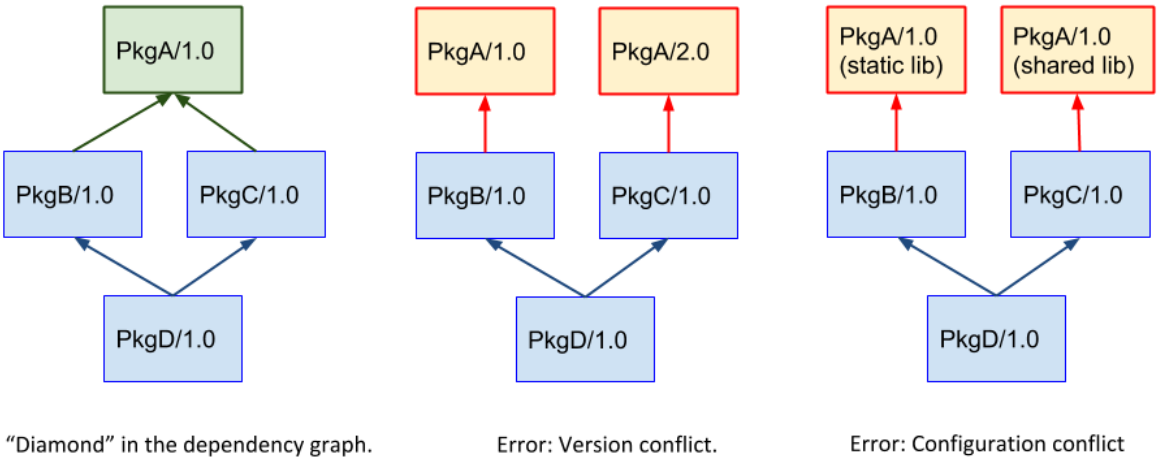
Read more about *Package Revisions*

11.1.2 Version and configuration conflicts

When two different branches of the same dependency graph require the same package, this is known as "diamonds" in the graph. If the two branches of a diamond require the same package but different versions, this is known as a conflict (a version conflict).

Lets say that we are building an executable in **pkgd/1.0**, that depends on **pkgb/1.0** and **pkgc/1.0**, which contain static libraries. In turn, **pkgb/1.0** depends on **pkga/1.0** and finally **pkgc/1.0** depends on **pkga/2.0**, which is also another static library.

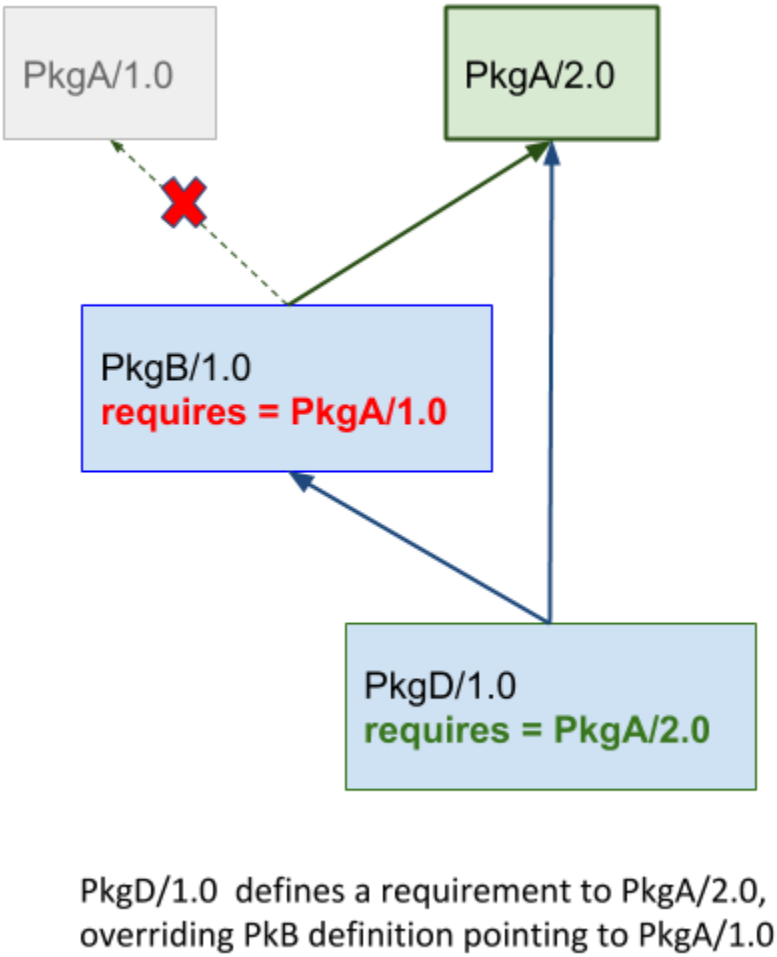
The executable in **pkgd/1.0**, cannot link with 2 different versions of the same static library in **pkgc**, and the dependency resolution algorithm raises an error to let the user decide which one.



The same situation happens if the different packages require different configurations of the same upstream package, even if the same version is used. In the example above, both **PkgB** and **PkgC** can be requiring the same version **pkga/1.0**, but one of them will try to use it as a static library and the other one will try to use it as shared library. The dependency resolution algorithm will also raise an error.

11.1.3 Dependencies overriding

The downstream consumer packages always have higher priority, so the versions they request, will be overridden upstream as the dependency graph is built, re-defining the possible requires that the packages could have. For example, **pkgb/1.0** could define in its recipe a dependency to **pkga/1.0**. But if a downstream consumer defines a requirement to **pkga/2.0**, then that version will be used in the upstream graph:



This is what enables the users to have control. Even when a package recipe upstream defines an older version, the downstream consumers can force to use an updated version. Note that this is not a diamond structure in the graph, so it is not a conflict by default. This behavior can be also restricted defining the *CONAN_ERROR_ON_OVERRIDE* environment variable to raise an error when these overrides happen, and then the user can go and explicitly modify the upstream **pkgb/1.0** recipe to match the version of PkgA and avoid the override.

In some scenarios, the downstream consumer **pkgd/1.0** might not want to force a dependency on `pkga`. There are several possibilities, for example that PkgA is a conditional requirement that only happens in some operating systems. If `pkgd` defines a normal requirement to `pkga`, then, it will be introducing that edge in the graph, forcing pkga to be used always, in all operating systems. For this purpose the `override` qualifier can be defined in requirement, see *requirements()*.

11.1.4 Versioning and binary compatibility

It is important to note and this point that versioning approaches and strategies should also be consistent with the binary management.

By default, Conan assumes *semver* compatibility, so it will not require to build a new binary for a package when its dependencies change their minor or patch versions. This might not be enough for C or C++ libraries which versioning scheme doesn't strictly follow semver. It is strongly suggested to read more about this in *Defining Package ABI Compatibility*

11.2 Version ranges

Version range expressions are supported, both in `conanfile.txt` and in `conanfile.py` requirements.

The syntax uses brackets. The square brackets are the way to inform Conan that is a version range. Otherwise, versions are plain strings. They can be whatever you want them to be (up to limitations of length and allowed characters).

```
class HelloConan(ConanFile):
    requires = "pkg/[>1.0 <1.8]@user/stable"
```

So when specifying `pkg/[expression]@user/stable`, it means that `expression` will be evaluated as a version range. Otherwise, it will be understood as plain text, so `requires = "pkg/version@user/stable"` always means to use the version `version` literally.

There are some packages that do not follow semver. A popular one would be the OpenSSL package with versions as `1.0.2n`. They cannot be used with version-ranges. To require such packages you always have to use explicit versions (without brackets).

The process to manage plain versions vs version-ranges is also different. The second one requires a "search" in the remote, which is orders of magnitude slower than direct retrieval of the reference (plain versions). Take it into account if you plan to use it for very large projects.

Expressions are those defined and implemented by https://pypi.org/project/node-semver/. Accepted expressions would be:

```
[>1.1 <2.1]                # In such range
[2.8]                      # equivalent to =2.8
[~3.0]                     # compatible, according to semver
[>1.1 || 0.8]              # conditions can be OR'ed
[1.2.7 || >=1.2.9 <2.0.0]  # This range would match the versions 1.2.7, 1.2.9, and 1.4.6,
↪ but not the versions 1.2.8 or 2.0.0.
```

There are two options for the version range:

- `loose=True|False` (default `True`): When using `loose=False` only valid Semantic Versioning strings are accepted.
- `include_prerelease=True|False` (default `False`): If set to `include_prerelease=True`, Conan will include prerelease versions in the search range. Take into account that prerelease versions have lower precedence than the associated normal one (e.g.: `1.0.0 > 1.0.0-beta`).

```
[>1.1 <2.1, include_prerelease=True]           # Would e.g. accept "2.0.0-pre.1" as␣
↪match
[~1.2.3, loose=False]                          # Would only accept correct Semantic␣
↪Versioning strings.
                                               # E.g. version "1.2.3.4" would not be␣
```

(continues on next page)

(continued from previous page)

```
↪accepted.
[~1.2.3, loose=False, include_prerelease=True]  # Both options can be used for the same␣
↪version range.
```

Version range expressions are evaluated at the time of building the dependency graph, from downstream to upstream dependencies. No joint-compatibility of the full graph is computed. Instead, version ranges are evaluated when dependencies are first retrieved.

This means, that if a package A depends on another package B (A->B), and A has a requirement for `C/[>1.2 <1.8]`, this requirement is evaluated first and it can lead to get the version `C/1.7`. If package B has the requirement to `C/[>1.3 <1.6]`, this one will be overwritten by the downstream one, it will output a version incompatibility error. But the "joint" compatibility of the graph will not be obtained. Downstream packages or consumer projects can impose their own requirements to comply with upstream constraints. In this case a override dependency to `C/[>1.3 <1.6]` can be easily defined in the downstream package or project.

The order of search for matching versions is as follows:

- First, the local conan storage is searched for matching versions, unless the **`--update`** flag is provided to **`conan install`**.
- If a matching version is found, it is used in the dependency graph as a solution.
- If no matching version is locally found, it starts to search in the remotes, in order. If some remote is specified with **`-r=remote`**, then only that remote will be used.
- If the **`--update`** parameter is used, then the existing packages in the local conan cache will not be used, and the same search of the previous steps is carried out in the remotes. If new matching versions are found, they will be retrieved, so subsequent calls to **`install`** will find them locally and use them.

Note: Version ranges are not used in generating `package_id` those are always determined by the resolved graph.

11.3 Package Revisions

The goal of the revisions feature is to achieve package immutability, the packages in a server are never overwritten.

Note: Revisions achieve immutability. For achieving reproducible builds and reproducible dependencies, **lockfiles** are used. Lockfiles can capture an exact state of a dependency graph, down to exact versions and revisions, and use it later to force their usage, even if new versions or revisions were uploaded to the servers.

Learn more about *lockfiles here.*

11.3.1 How it works

In the client

- When a **recipe** is exported, Conan calculates a unique ID (revision). For every change, a new recipe revision (RREV) will be calculated. By default it will use the checksum hash of the recipe manifest.

  Nevertheless, the recipe creator can explicitly declare the *revision mode*, it can be either `scm` (uses version control system or raises) or `hash` (use manifest hash).

- When a **package** is created (by running *conan create* or *conan export-pkg*) a new package revision (PREV) will be calculated always using the hash of the package contents. The packages and their revisions (PREVs) belongs to a concrete recipe revision (RREV). The same package ID (for example for Linux/GCC5/Debug), can have multiple revisions (PREVs) that belong to a concrete RREV.

If a client requests a reference like `lib/1.0@conan/stable`, Conan will automatically retrieve the latest revision in case the local cache doesn't contain any revisions already. If a client needs to update an existing revision, they have to ask for updates explicitly with `-u, --update` argument to **`conan install`** command. In the client cache there is **only one revision installed simultaneously**.

The revisions can be pinned when you write a reference (in the recipe requires, a reference in a **`conan install`** command,...) but if you don't specify a revision, the server will retrieve the latest revision.

If you specify a pinned revision in your references, and that revision is not the one present in the Conan cache, and `--update` is not provided, it will fail with an error. This behavior can be change with `core:allow_explicit_revision_update=True [conf]` configuration. It is experimental and can result in later errors (that won't be possible to fix, use it at your own risk), for example as the cache can only host 1 revision, it might happen that multiple pinned references are competing for it, and kicking each others revisions out of the cache while the dependency graph is computed.

You can specify the references in the following formats:

Reference	Meaning
`lib/1.0@conan/stable`	Latest RREV for `lib/1.0@conan/stable`
`lib/1.0@conan/stable#RREV`	Specific RREV for `lib/1.0@conan/stable`
`lib/1.0@conan/stable#RREV:PACKAGE_ID`	A binary package belonging to the specific RREV
`lib/1.0@conan/stable#RREV:PACKAGE_ID#PREV`	A binary package revision PREV belonging to the specific RREV

In the server

By using a new folder layout and protocol it is able to store multiple revisions, both for recipes and binary packages.

11.3.2 How to activate the revisions

You have to explicitly activate the feature by either:

- Adding `revisions_enabled=1` in the `[general]` section of your *conan.conf* file (preferred)
- Setting the `CONAN_REVISIONS_ENABLED=1` environment variable.

Take into account that it changes the default Conan behavior. e.g:

- A client with revisions enabled will only find binary packages that belong to the installed recipe revision. For example, If you create a recipe and run **`conan create . user/channel`** and then you modify the recipe and export it (**`conan export . user/channel`**), the binary package generated in the **`conan create`** command doesn't belong to the new exported recipe. So it won't be located unless the previous recipe is recovered.
- If you generate and upload N binary packages for a recipe with a given revision, then if you modify the recipe, and thus the recipe revision, you need to build and upload N new binaries matching that new recipe revision.

11.3.3 GIT and Line Endings on Windows

Warning: Problem

Git will (by default) checkout files in Windows systems using CRLF line endings, effectively producing different files. As files are different, the Conan revisions will be different from the revisions computed in other platforms such as Linux, resulting in missing the respective binaries in the other revision.

Solution

It is necessary to instruct Git to do the checkout with the same line endings. This can be done several ways, for example, by adding a .gitattributes file:

```
[auto]
  crlf = false
```

11.3.4 Server support

- `conan_server` >= 1.13.
- `Artifactory` >= 6.9.
- `ConanCenter`.

11.4 Lockfiles

Warning: This is an **experimental** feature subject to breaking changes in future releases.

Lockfiles are files that store the information of a dependency graph, including the exact versions, revisions, options, and configuration of that dependency graph. These files allow for later achieving reproducible results, and installing or using the exact same dependencies even when the requirements are not fully reproducible, for example when using version ranges or using package revisions.

11.4.1 Introduction

Warning: This is an **experimental** feature subject to breaking changes in future releases.

Let's introduce lockfiles by example, with 2 packages, package `pkgb` that depends on package `pkga`.

Note: The code used in this section, including a *build.py* script to reproduce it, is in the examples repository: https://github.com/conan-io/examples. You can go step by step reproducing this example while reading the below documentation.

```
$ git clone https://github.com/conan-io/examples.git
$ cd features/lockfiles/intro
# $ python build.py only to run the full example, but better go step by step
```

Locking dependencies

This example uses `full_version_mode`, that is, if a package changes any part of its version, its consumers will need to build a new binary because a new `package_id` will be computed. This example will use version ranges, and it is not necessary to have revisions enabled. It also does not require a server, everything can be reproduced locally.

```
$ conan config set general.default_package_id_mode=full_version_mode
```

Let's start by creating from the recipe and source in the `pkga` folder, a first `pkg/0.1@user/testing` package in our local cache:

```
$ conan create pkga pkga/0.1@user/testing
```

Now we want to start developing and testing the code for `pkgb`, but we want to create a "snapshot" of the dependency graph, to isolate our development from possible changes (note that the recipe in *pkgb/conanfile.py* contains a require like `requires = "pkga/[>0.0]@user/testing"`).

```
$ cd pkgb
$ conan lock create conanfile.py --user=user --channel=testing --lockfile-out=locks/pkgb_
↪deps.lock
```

This will create a *pkgb_deps.lock* file in the *locks* folder. Note that we have passed the user and channel of the future package that we will create as `--user=user --channel=testing`.

Let's have a look at the lockfile:

```
{
    "graph_lock": {
        "nodes": {
            "0": {
                "ref": "pkgb/0.1@user/testing",
                "options": "shared=False",
                "requires": ["1"],
                "path": "..\\conanfile.py",
                "context": "host"
            },
            "1": {
                "ref": "pkga/0.1@user/testing",
                "options": "",
                "package_id": "4024617540c4f240a6a5e8911b0de9ef38a11a72",
                "prev": "0",
                "context": "host"
            }
        },
        "revisions_enabled": false
    },
    "version": "0.4",
    "profile_host": "[settings]\narch=x86_64\narch_build=x86_64\nbuild_type=Release\
↪ncompiler=Visual Studio\ncompiler.runtime=MD\ncompiler.version=15\nos=Windows\nos_
↪build=Windows\n[options]\n[tool_requires]\n[env]\n"
}
```

We can see the pkga/0.1@user/testing dependency in the lockfile, together with its package_id. This dependency is fully locked. The pkgb/0.1@user/testing doesn't have a package_id yet, because so far it is just a local *conanfile.py* as a consumer, not a package. But the user/testing user and channel are already defined.

It is important to note that the *pkgb_deps.lock* lockfile contains the current profile for the current configuration.

At this moment we have captured the dependency graph for pkgb. Now, it would be possible that a new version of pkga is created:

```
$ cd ..
# The recipe generates different package code depending on the version, automatically
$ conan create pkga pkga/0.2@user/testing
```

If now we install and build our code in pkgb we would get:

```
$ mkdir pkgb/build
$ cd pkgb/build
$ conan install ..
> ... pkga/0.2@user/testing from local cache - Cache
# Example for VS, use your compiler here
$ cmake ../src -G "Visual Studio 15 Win64"
$ cmake --build . --config Release
$ ./bin/greet
HelloA 0.2 Release
HelloB Release!
Greetings Release!
```

But as explained above, the purpose of the lockfile is to capture the dependencies and use them later. Let's pass the lockfile as an argument to guarantee the usage of the locked pkga/0.1@user/testing dependency:

```
$ conan install .. --lockfile=../locks/pkgb_deps.lock
> ... pkga/0.1@user/testing from local cache - Cache
$ cmake ../src -G "Visual Studio 15 Win64"
$ cmake --build . --config Release
$ ./bin/greet
HelloA 0.1 Release
HelloB Release!
Greetings Release!
```

That's it. We managed to depend on pkga/0.1@user/testing instead of the pkga/0.2@user/testing although the later satisfies the version range and is available in the cache. Using the same dependency was possible because we used the information stored in the lockfile.

Immutability

A core concept of lockfiles is their immutability and the integrity of its data:

Important: The information stored in a lockfile cannot be changed. Any attempt to modify locked data will result in an error.

For example, if now we try to do a **conan install** that also builds pkga from source:

```
$ conan install .. --lockfile=../locks/pkgb_deps.lock --build=pkga
ERROR: Cannot build 'pkga/0.1@user/testing' because it is already locked in the input␣
↪lockfile
```

It is an error, because the `pkga/0.1@user/testing` dependency was fully locked. When the lockfile was created, the `pkga/0.1@user/testing` was found, including a binary, and that information was stored. Everytime this lockfile is used, it assumes this package and binary exist and it will try to get them, but it will never allow to re-build, because that can violate the integrity of the lockfile. For example, if we were using `package_revision_mode`, a new binary of `pkga` would produce new package-ids of all its consumers, that will not match the package-ids stored in the lockfile.

It is possible though to control what is being locked with the `--build` argument provided to the **`conan lock create`** command.

The same principle applies if we try to create a package for `pkgb` and it tries to alter the user and channel `user/testing` that were provided at the time of the **`conan lock create`** command used above.

```
$ cd ..
$ conan create . user/stable --lockfile=locks/pkgb_deps.locked
ERROR: Attempt to modify locked pkgb/0.1@user/testing to pkgb/0.1@user/stable
```

Again, it is important to keep the integrity. Package recipes can have conditional or parameterized dependencies, based on user and channel for example. If we try to create the `pkgb` package with different user and channel, it could result in a different dependency graph, totally incompatible with the one captured in the lockfile. If `pkgb/0.1@user/testing` was stored in the lockfile, any command using this lockfile must respect and keep it without changes.

Note: A package in a lockfile is fully locked if it contains a `prev` (package revision) field defined. Fully locked packages cannot be built from sources. Partially locked packages do not contain a `prev` defined. They lock the reference and the package-id, and they can be built from sources.

Reproducibility

That doesn't mean that a lockfile cannot evolve at all. Using the **`--lockfile`** argument, we are able to create `pkgb/0.1@user/testing` guaranteeing it is being created depending on `pkga/0.1@user/testing`. Additionally, if we use the **`--lockfile-out`** argument, we can obtain an updated version of the lockfile:

```
$ conan create . user/testing --lockfile=locks/pkgb_deps.lock --lockfile-out=locks/pkgb.
↪lock
```

And if we inspect the new *locks/pkgb.lock* file:

```
{
    ...
    "0": {
        "ref": "pkgb/0.1@user/testing",
        "options": "shared=False",
        "package_id": "2418b211603ca0a3858d9dd1fc1108d54a4cab99",
        "prev": "0",
        "modified": true,
        "requires": ["1"],
        "context": "host"
    }
```

(continues on next page)

(continued from previous page)

```
    ...
}
```

Note that some fields of the lockfile are now completed, as the modified flag, that indicates that `pkgb` was built in the conan create command. That information can be useful in the CI environment to know which packages were built by different jobs. Those modified flags can be reset using the **`conan lock clean-modified`**. Also, it can be appreciated in *locks/pkgb.lock* that now `pkgb/0.1@user/testing` is fully locked, as a package (not a local *conanfile.py*), and contains a `package_id`. So if we try to use this new file for creating the package again, it will error, as a package that is fully locked cannot be rebuilt:

```
$ conan create . user/testing --lockfile=locks/pkgb.lock
ERROR: Attempt to modify locked pkgb/0.1@user/testing to pkgb/0.1@user/testing
```

But we can reproduce the same set of dependencies and the creation of `pkgb`, using the *pkgb_deps.lock* lockfile:

```
$ conan create . user/testing --lockfile=locks/pkgb_deps.lock # OK
```

The *pkgb.lock* can be used later in time to install the `pkgb` application (the `pkgb` *conanfile.py* contains a `deploy()` method for convenience for this example), and get the same package and dependencies:

```
$ cd ..
$ mkdir consume
$ cd consume
$ conan install pkgb/0.1@user/testing --lockfile=../pkgb/locks/pkgb.lock
$ ./bin/greet
HelloA 0.1 Release
HelloB Release!
Greetings Release!
```

As long as we have the *pkgb.lock* lockfile, we will be able to robustly reproduce this install, even if the packages were uploaded to a server, if there are new versions that satisfy the version ranges, etc.

Important: All the examples and documentation of this section is done with version ranges and revisions disabled. Lockfiles also work and can lock both recipe and package revisions, with the same behavior as version-ranges. All is necessary is to enable revisions. The only current limitation is that the local cache cannot store more than one revision at a time, but that is a limitation of the cache and unrelated to lockfiles.

11.4.2 Multiple configurations

> **Warning:** This is an **experimental** feature subject to breaking changes in future releases.

In the previous section we managed just 1 configuration, for the default profile. In many applications, packages need to be built with several different configurations, typically managed by different profile files.

Note: This section continues with the previous example with the *Introduction*. The code used in this section, including a *build.py* script to reproduce it, is in the examples repository: https://github.com/conan-io/examples. You can go step by step reproducing this example while reading the below documentation.

```
$ git clone https://github.com/conan-io/examples.git
$ cd features/lockfiles/intro
# $ python build.py only to run the full example, but better go step by step
```

Lets start in the *features/lockfiles/intro* of the examples repository, remove the previous packages, and create both release and debug `pkga` packages:

```
$ conan remove "pkg*" -f
$ conan create pkga pkga/0.1@user/testing
$ conan create pkga pkga/0.1@user/testing -s build_type=Debug
```

Now, we could (don't do it) create 2 different lockfiles, one for each configuration:

```
# DO NOT type these commands, we'll do it better below
$ cd pkgb
$ conan lock create conanfile.py --user=user --channel=testing --lockfile-out=locks/pkgb_
↪release.lock
$ conan lock create conanfile.py --user=user --channel=testing --lockfile-out=locks/pkgb_
↪debug.lock -s build_type=Debug
```

Important: The dependency graph is different for each different configuration/profile. Not only the package-ids, but also because of conditional requirements, the dependencies can be different. Then, it is necessary to create a lockfile for every different configuration/profile.

But, what if a new `pkga/0.2@user/testing` version was created in the time between both commands? Although this is unlikely to happen in this example, because everything is local. However, it could happen that `pkga` was in a server and the CI uploads a new `pkga/0.2@user/testing` version while we are running the above commands.

Base lockfiles

Conan proposes a "base" lockfile, with the **`--base`** argument, that will capture only the versions and topology of the graph, but not the package-ids:

```
$ cd pkgb
$ conan lock create conanfile.py --user=user --channel=testing --lockfile-out=locks/pkgb_
↪base.lock --base
```

Let's inspect the *locks/pkgb_base.lock* lockfile:

```
{
    "graph_lock": {
        "nodes": {
            "0": {
                "ref": "pkgb/0.1@user/testing",
                "requires": ["1"],
                "path": "..\\conanfile.py",
                "context": "host"
            },
            "1": {
                "ref": "pkga/0.1@user/testing",
```

(continues on next page)

(continued from previous page)

```
                "context": "host"
            }
        },
        "revisions_enabled": false
    },
    "version": "0.4"
}
```

This lockfile is different to the ones in the previous section. It does not store the `profile`, and it does not capture the package-ids or the options of the nodes. It captures the topology of the graph, and the package references and versions.

At this point, the new `pkga/0.2@user/testing` version packages could be created:

```
$ cd ..
# The recipe generates different package code depending on the version, automatically
$ conan create pkga pkga/0.2@user/testing
$ conan create pkga pkga/0.2@user/testing -s build_type=Debug
```

Using the "base" *locks/pkgb_base.lock* lockfile, now we can obtain a new lockfile for both debug and release configurations, and it is guaranteed that both will use the `pkga/0.1@user/testing` dependency, and not the new one:

```
$ cd pkgb
$ conan lock create conanfile.py --user=user --channel=testing --lockfile=locks/pkgb_
↪base.lock --lockfile-out=locks/pkgb_deps_debug.lock -s build_type=Debug
$ conan lock create conanfile.py --user=user --channel=testing --lockfile=locks/pkgb_
↪base.lock --lockfile-out=locks/pkgb_deps_release.lock
```

Now, we will have 2 lockfiles, *locks/pkgb_deps_debug.lock* and *locks/pkgb_deps_release.lock*. Each one will lock different profiles and different package-id of `pkga/0.1@user/testing`.

Note: In Conan 1.X, if you are generating lockfiles with separate build and host profiles, your base lockfiles must also use separate build and host profiles. For example, here we are generating a base lockfile that will be used to generate lockfiles for a Linux and Windows build:

```
# The build and host profiles you choose for the base lockfile should
# include all dependencies needed by all lockfiles you will generate
# from the base lockfile.
$ conan lock create conanfile.py -pr:b release -pr:h debug --lockfile-out=base.lock --
↪base

# Use the base lockfile to generate lockfiles for a Linux and Windows
# build.
$ conan lock create conanfile.py -pr:b linux-rel -pr:h linux-dbg --lockfile=base.lock --
↪lockfile-out=linux.lock
$ conan lock create conanfile.py -pr:b windows-rel -pr:h windows-dbg --lockfile=base.
↪lock --lockfile-out=windows.lock
```

For more information, please see GitHub issue #9446.

Locked configuration

The lockfiles store the effective configuration, settings, options, resulting from the used profiles and command line arguments. That configuration arguments can be passed to the `conan lock create` command, but not when using lockfiles. For example:

```
$ mkdir build && cd build
$ conan install .. --lockfile=../locks/pkgb_deps_debug.lock -s build_type=Debug
ERROR: Cannot use profile, settings, options or env 'host' when using lockfile
```

results in an error, because the *locks/pkgb_deps_debug.lock* already stores the `settings.build_type` and passing it in the command line could only result in inconsistencies and errors.

Important: Lockfiles store the full effective profile configuration. It is not possible to pass configuration, settings, options or profile arguments when using lockfiles (only when creating the lockfiles)

With the two captured lockfiles, now we can locally build and run our `pkgb` application for both configurations, guaranteeing the dependency to `pkga/0.1@user/testing`:

```
$ conan install .. --lockfile=../locks/pkgb_deps_release.lock
$ cmake ../src -G "Visual Studio 15 Win64"
$ cmake --build . --config Release
$ ./bin/greet
HelloA 0.1 Release
HelloB Release!
Greetings Release!
$ conan install .. --lockfile=../locks/pkgb_deps_debug.lock
$ cmake --build . --config Debug
$ ./bin/greet
HelloA 0.1 Debug
HelloB Debug!
Greetings Debug!
```

We can create `pkgb` package again for both configurations:

```
$ cd ..
$ conan create . user/testing --lockfile=locks/pkgb_deps_release.lock --lockfile-
↪out=locks/pkgb_release.lock
$ conan create . user/testing --lockfile=locks/pkgb_deps_debug.lock --lockfile-out=locks/
↪pkgb_debug.lock
```

And we could still use the lockfiles later in time to install the `pkgb` package with the same dependencies and configuration that were used to create that package:

```
$ cd ..
$ mkdir consume
$ cd consume
$ conan install pkgb/0.1@user/testing --lockfile=../pkgb/locks/pkgb_release.lock
$ ./bin/greet
HelloA 0.1 Release
HelloB Release!
Greetings Release!
$ conan install pkgb/0.1@user/testing --lockfile=../pkgb/locks/pkgb_debug.lock
```

(continues on next page)

(continued from previous page)

```
$ ./bin/greet
HelloA 0.1 Debug
HelloB Debug!
Greetings Debug!
```

As you can see, the immutability principle remains. If we try to use *pkgb_release.lock* to create the `pkgb` package again instead of the *pkgb_deps_release.lock* lockfile, it will error, as `pkgb` would be already fully locked in the former.

11.4.3 Evolving lockfiles

> **Warning:** This is an **experimental** feature subject to breaking changes in future releases.

As described before, lockfiles are immutable, they cannot change the information they contain. If some install or create command tries to change some data in a lockfile, it will error. This doesn't mean that operations on lockfiles cannot be done, as it is possible to create a new lockfile from an existing one. We have already done this, obtaining a full lockfile for a specific configuration from an initial "base" lockfile.

There are several scenarios you might want to create a new lockfile from an existing one.

Deriving a partial lockfile

Lets say that we have an application `app/1.0` that depends on `libc/1.0` that depends on `libb/1.0` that finally depends on `liba/1.0`. We could capture a "base" lockfile from it, and then several full lockfiles, one per configuration:

```
$ conan lock create --reference=app/1.0@ --base --lockfile-out=app_base.lock
$ conan lock create --reference=app/1.0@ --lockfile=app_base.lock -s build_type=Release -
↪-lockfile-out=app_release.lock
$ conan lock create --reference=app/1.0@ --lockfile=app_base.lock -s build_type=Debug --
↪lockfile-out=app_debug.lock
```

Now a developer wants to start testing some changes in `libb`, using the same dependencies versions defined in the lockfile. As `libb` is locked, it will not be possible to create a new version `libb/1.1` or build a new binary for it with the existing lockfiles. But we can create a new lockfile for it in different ways. For example, we could derive directly from the *app_release.lock* and *app_debug.lock* lockfiles:

```
$ git clone <libb-repo> && cd libb
$ conan lock create conanfile.py --lockfile=app_release.lock --lockfile-out=libb_deps_
↪release.lock
$ conan lock create conanfile.py --lockfile=app_debug.lock --lockfile-out=libb_deps_
↪debug.lock
```

This will create partial lockfiles, only for `libb` dependencies, i.e. locking `liba/1.0`, that can be used while installing, building and testing `libb`.

But it is also possible to derive a new "base" profile from *app_base.lock* only for libb dependencies, and then compute from it the configuration specific profiles.

These partial lockfiles will be smaller than the original app lockfiles, not containing information at all about `app` and `libc`.

Unlocking packages with –build

It is also possible to derive a partial lockfile for `libb/1.0` without cloning the `libb` repository, directly with:

```
$ conan lock create --reference=libb/1.0 --lockfile=app_release.lock --lockfile-out=libb_
↪release.lock
$ conan lock create --reference=libb/1.0 --lockfile=app_debug.lock --lockfile-out=libb_
↪debug.lock
```

These new lockfiles could be used to install the `libb/1.0` package, without building it, but if we tried to build it from sources, it will fail:

```
$ conan install libb/1.0@ --lockfile=libb_release.lock # Works
$ conan install libb/1.0@ --build=libb --lockfile=libb_release.lock # Fails, libb is␣
↪locked
```

The second scenario fails. This is because when the *app_release.lock* lockfile was captured, it completely locked all the information (including `libb/1.0`'s package revision). If we try to build a new binary, the lock protection will raise. If we want to "unlock" the binary package revision, we need to tell the lockfile when we are capturing such lockfile, that we plan to build it, with the **`--build`** argument:

```
# Note the --build=libb argument
$ conan lock create --reference=libb/1.0 --build=libb --lockfile=app_release.lock --
↪lockfile-out=libb_release.lock
# This will work, building a new binary
$ conan install libb/1.0@ --build=libb --lockfile=libb_release.lock --lockfile-out=libb_
↪release2.lock
```

As usual, if you are building a new binary, it is desired to provide a **`--lockfile-out=libb_release2.lock`** to capture such a new binary package revision in the new lockfile.

Integrating a partial lockfile

This would be the opposite flow. Lets take the previous *libb_deps_release.lock* and *libb_deps_debug.lock* lockfiles and create new `libb/1.1` packages with it, and obtaining new lockfiles:

```
# in the libb source folder
$ conan  create . --lockfile=libb_deps_release.lock --lockfile-out=libb_release.lock
$ conan  create . --lockfile=libb_deps_debug.lock --lockfile-out=libb_debug.lock
```

These lockfiles will be containing locked information to `liba/1.0` and a new `libb/1.1` version. Now we would like to check if `app/1.0` will pick this new version, and in case it is used, we would like to rebuild whatever is necessary (that is part of the next CI section).

Important: It is not possible to pick the old *app_base.lock*, *app_release.lock* or *app_debug.lock* lockfiles and inject the new `libb/1.1` version, as this would be violating the integrity of the lockfile. Nothing guarantees that the downstream packages will effectively use the new version, as it might fall outside the valid range defined in `libc/1.0`, for example. Also, downstream consumers `app/1.0` and `libc/1.0` could result in different package-ids as a result of having a new dependency, and this goes against the immutability of the lockfile data, as the package-ids for them would be already locked.

Let's create new lockfiles that will use the existing `libb_debug.lock` and `libb_release.lock` information if possible:

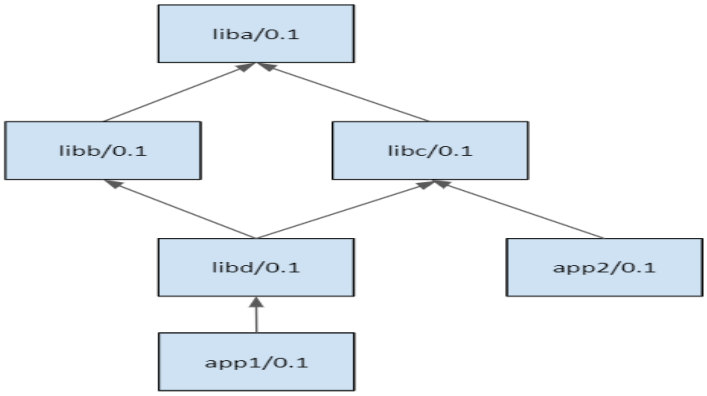
```
$ conan lock create --reference=app/1.0@ --lockfile=libb_release.lock --lockfile-out=app_
↪release.lock
$ conan lock create --reference=app/1.0@ --lockfile=libb_debug.lock --lockfile-out=app_
↪debug.lock
```

This will create new *app_release.lock* and *app_debug.lock* that will have both `libb/1.1` and `liba/1.0` locked. If for some reason, `libc/1.0` had fixed a `requires = "libb/1.0"`, then the resulting lockfile would resolve and lock `libb/1.0` instead. The `build-order` command (see next section) will tell us that there is nothing to build, as it is effectively computing the same lockfile that existed before. It is also possible, and a CI pipeline could do it, to directly check that `libb/1.1` is defined inside the new lockfiles. If it is not there, it means that it didn't integrate, and nothing needs to be done downstream.

11.4.4 Build order in lockfiles

Warning: This is an **experimental** feature subject to breaking changes in future releases.

In this section we are going to use the following packages, defining this dependency graph.



Note: The code used in this section, including a *build.py* script to reproduce it, is in the examples repository: https://github.com/conan-io/examples. You can go step by step reproducing this example while reading the below documentation.

```
$ git clone https://github.com/conan-io/examples.git
$ cd features/lockfiles/build_order
# $ python build.py only to run the full example, but better go step by step
```

The example in this section uses `full_version_mode`, that is, if a package changes any part of its version, its consumers will need to build a new binary because a new `package_id` will be computed. This example will use version ranges, and it is not necessary to have revisions enabled. It also does not require a server, everything can be reproduced locally.

```
$ conan config set general.default_package_id_mode=full_version_mode
```

Let's start by creating the initial dependency graph, without binaries (just the exported recipes), in our local cache:

```
$ conan export liba liba/0.1@user/testing
$ conan export libb libb/0.1@user/testing
$ conan export libc libc/0.1@user/testing
$ conan export libd libd/0.1@user/testing
$ conan export app1 app1/0.1@user/testing
$ conan export app2 app2/0.1@user/testing
```

Now we will create a lockfile that captures the dependency graph for `app1/0.1@user/testing`. In the same way we created lockfiles for a local *conanfile.py* in a user folder, we can also create a lockfile for a recipe in the Conan cache, with the **`--reference`** argument:

```
$ conan lock create --reference=app1/0.1@user/testing --lockfile-out=app1.lock
```

The resulting *app1.lock* lockfile will not be able to completely lock the binaries because such binaries do not exist at all. This can be checked in the *app1.lock* file, the packages do not contain a package revision (`prev`) field at all:

```
{
   ...
   "4": {
   "ref": "liba/0.1@user/testing",
   "options": "",
   "package_id": "5ab84d6acfe1f23c4fae0ab88f26e3a396351ac9",
   "context": "host"
   }
   ...
}
```

We can now compute the "build-order" of the dependency graph. The "build-order" lists in order all the packages that needs to be built from sources. The logic is the following:

- If a package is fully locked (it contains a package revision field `prev` in the lockfile), it will not be built from sources and will **never** appear in the build-order list.
- If a package is not fully locked (it does **not** contain a package revision `prev` in the lockfile), it will appear in the build-order list. This situation happens both when the package binary doesn't exist yet, or when the `--build` argument was used while creating the lockfile.

```
$ conan lock build-order app1.lock --json=build_order.json
```

The resulting *build_order.json* file is a list of lists, structured by levels of possible parallel builds:

```
[
  # First level liba
  [["liba/0.1@user/testing", "5ab8...1ac9", "host", "4"]],
  # Second level libb and libc
  [["libb/0.1@user/testing", "cfd1...ec23", "host", "3"],
   ["libc/0.1@user/testing", "cfd1...ec23", "host", "5"]],
  # Third level libd
  [["libd/0.1@user/testing", "d075...5b9d", "host", "2"]],
  # Fourth level libd
  [["app1/0.1@user/testing", "3bf2...5188", "host", "1"]]
]
```

Every item in the outer list is a "level" in the graph, a set of packages that needs to be built, and are independent of every other package in the level, so they can be built in parallel. Levels in the build order must be respected, as the

second level cannot be built until all the packages in the first level are built and so on. In this example, once the build of `liba/0.1@user/testing` finishes, as it is the only item in the first level, the second level can start, and it can build both `libb/0.1@user/testing` and `libc/0.1@user/testing` in parallel. It is necessary that both of them finish their build to be able to continue to the third level, that contains `libd/0.1@user/testing`, because this package depends on them.

Every item in each level has 4 elements: `[ref, package_id, context, node-id]`. At the moment the only necessary one is the first one. The `ref` value is the one that can be used for example in a **conan install** command like:

```
$ conan install <ref> --build=<ref> --lockfile=mylock.lock
```

The last value, the `node-id` could be used in cases where the `ref` is not enough to address a given package in the graph, for example when the same package can be found in the graph multiple times. In this case, explicitly adding the `--lockfile-node-id` argument can resolve the ambiguity (this is an **experimental feature**, subject to breaking changes):

```
$ conan install <ref> --build=<ref> --lockfile=mylock.lock --lockfile-node-id=<node-id>
```

Defining builds

The definition of what needs to be built comes from the existing binaries plus the **--build** argument in the **conan lock create**.

Let's build all the binaries for the exported packages first:

```
# Build app1 and dependencies
$ conan install app1/0.1@user/testing --build=missing
```

Now that there are binaries for all packages in the cache, let's capture them in a new lockfile and compute the build order:

```
# Create a new lockfile now with all the package binaries
$ conan lock create --reference=app1/0.1@user/testing --lockfile-out=app1.lock
# And check which one needs to be built
$ conan lock build-order app1.lock --json=build_order.json
# The build order is emtpy, nothing to build
[]
```

The result of this build order is empty. As the **conan lock create** found existing binaries, everything is fully locked, nothing needs to be built.

If we specify the **--build** flag, then the behavior is different:

```
$ conan lock create --reference=app1/0.1@user/testing --lockfile-out=app1.lock --build
# the lockfile will not lock the binaries
# And check which one needs to be built
$ conan lock build-order app1.lock --json=build_order.json
[[["liba/0.1@user/testing", "5ab8...1ac9", "host", "4"]], ...
```

This feature is powerful when combined with `package_id_modes`, because it can automatically define the minimum set of packages that needs to be built for any change in the dependency graph.

Let's say that a new version `libb/1.1@user/testing` is created. But if we check the `libd` *conanfile.py* requirement `libb/[>0.0 <1.0]@user/testing`, we can see that this 1.1 version falls outside of the valid version range. Then, it does not affect `libd` or `app1` and nothing needs to be built:

```
$ conan create libb libb/1.1@user/testing
$ conan lock create --reference=app1/0.1@user/testing --lockfile-out=app1.lock
$ conan lock build-order app1.lock --json=build_order.json
[] # Empty, nothing to build, libb/1.1 does not become part of app1
```

If on the contrary, a new `libb/0.2@user/testing` is created, and we capture a new lockfile, it will contain such new version. Other packages, like `liba` and `libc` are not affected by this new version, and will be fully locked in the lockfile, but the dependents of `libb` now won't be locked and it will be necessary to build them:

```
$ conan create libb libb/0.2@user/testing
$ conan lock create --reference=app1/0.1@user/testing --lockfile-out=app1.lock
$ conan lock build-order app1.lock --json=build_order.json
[[['libd/0.1@user/testing', '97e9...b7f4', 'host', '2']],
 [['app1/0.1@user/testing', '2bf1...e405', 'host', '1']]]
```

So in this case the *app1.lock* is doing these things:

- Fully locking the non-affected packages (`liba/0.1`, `libc/0.1`)
- Fully locking the `libb/0.2`, as the binary that was just created is valid for our `app1` (Note that this might not always be true, and `app1` build could require a different `libb/0.2` binary).
- Partial locking (the version and package-id) of the affected packages that need to be built (`libd/0.1` and `app1/0.1`).
- Retrieving via `build-order` the right order in which the affected packages need to be built.

Recall that a package in a lockfile is fully locked if it contains a `prev` (package revision) field defined. Fully locked packages cannot be built from sources. Partially locked packages do not contain a `prev` defined. They lock the reference and the package-id, and they can be built from sources.

If we want to check if the new `libb/0.2` version affects to the `app2` and something needs to be rebuilt, the process is identical:

```
$ conan lock create --reference=app2/0.1@user/testing --lockfile-out=app2.lock
$ conan lock build-order app2.lock --json=build_order2.json
[]
```

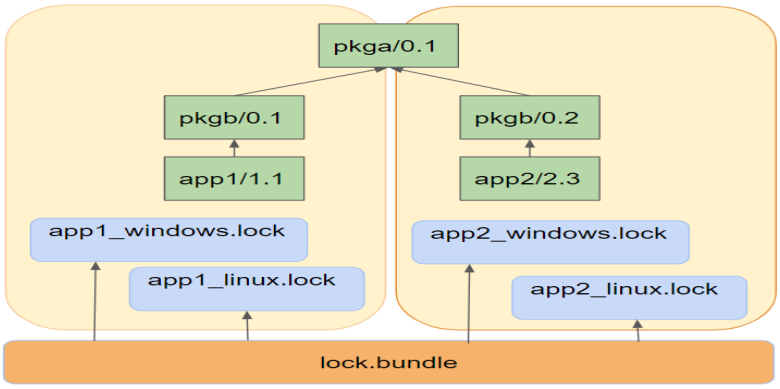
As expected, nothing to build, as `app2` does not depend on `libb` at all.

11.4.5 Lockfile bundles

> **Warning:** This is an **experimental** feature subject to breaking changes in future releases.

Every package build using lockfiles requires a given configuration-specific lockfile, and after the build, that lockfile is updated to include the built package revision. If we have different configurations for different variants as different architectures, compiler versions or Debug/Release, a build will be typically necessary for each one.

In real life, it is also likely that we might want to build together different applications or products, that could be even disconnected, and we want to do it as efficiently and fast (in parallel) as possible. We could have the following situation:



In this diagram we see that we are building and releasing 2 different products in our team: `app1/1.1` and `app2/2.3`. `app1` depends on `pkgb/0.1` (omitting `user/channel` for brevity, but please use it) and `app2` depends on `pkgb/0.2`. In turn, both versions of `pkgb` depend on the same `pkga/0.1` version.

If we are building both products for 2 different configurations each (lets say Windows and Linux), we could capture 4 different lockfiles:

```
$ conan lock create --ref=app1/1.1 --base --lockfile-out=app1_base.lock
$ conan lock create --ref=app2/2.3 --base --lockfile-out=app2_base.lock

$ conan lock create --ref=app1/1.1 -s os=Windows --lockfile=app1_base.lock --lockfile-
↪out=app1_windows.lock
$ conan lock create --ref=app1/1.1 -s os=Linux  --lockfile=app1_base.lock --lockfile-
↪out=app1_linux.lock
$ conan lock create --ref=app2/2.3 -s os=Windows  --lockfile=app2_base.lock --lockfile-
↪out=app2_windows.lock
$ conan lock create --ref=app2/2.3 -s os=Linux  --lockfile=app2_base.lock --lockfile-
↪out=app2_linux.lock
```

If we launched these 4 lockfiles builds in parallel, we can see that `pkga/0.1` will be built 4 times, 2 times in Windows and 2 times in Linux. The extra build in each OS is redundant and can be avoided. But we need a way to orchestrate it, that is what a lockfile bundle is for.

Creating a lockfile bundle

Creating a lockfile bundle can be done with the `conan lock bundle create` command, passing the list of all lockfiles for all configurations and products, and obtaining one single output bundle:

```
$ conan lock bundle create app1_windows.lock app1_linux.lock app2_windows.lock app2_
↪linux.lock --bundle-out=lock.bundle
```

Inspecting the resulting lockfile bundle file, we can see it is a json file with the following structure:

```
"lock_bundle": {
    "app1/1.1@#584778f98ba1d0eb7c80a5ae1fe12fe2": {
        "packages": [{
                "package_id": "3bcd6800847f779e0883ee91b411aad9ddd8e83c" ,
                "lockfiles": {
                    "app1_windows.lock": [
                        "1"
```

(continues on next page)

(continued from previous page)

```
                    ]
                },
                "prev": null,
                "modified": null
            }, {
                "package_id": "60fbb0a22359b4888f7ecad69bcdfcd6e70e2784",
                "lockfiles": {
                    "app1_linux.lock": [
                        "1"
                    ]
                },
                "prev": null,
                "modified": null
            }
        ],
        "requires": [
            "pkgb/0.1@#cd8f22d6f264f65398d8c534046e8e20"
        ]
    }
}
```

The bundle groups items per "recipe reference", included the recipe revision, like `app1/1.1@#584778f98ba1d0eb7c80a5ae1fe12fe2`. For each one, it will list all different binaries, identified by their `package_id` that are involved in the different lockfiles, listing all lockfiles for each `package_id`. In this case, as `app1` only belongs to app1 lockfiles, only one lockfile `app1_windows.lock`, `app1_linux.lock` is in each `package_id`. Also, the package revision `prev` is listed, in this case being `null`, because there is no locked binary in the lockfiles, but is going to be built.

Note: The relative path between the bundle file and the lockfile files need to be maintained. In the example `app1_linux.lock` means that the lockfile is located in the same folder as the bundle file itself. If moving the bundle to a different machine, the lockfiles should be moved too, maintaining the same relative layout.

The interesting part is in the `pkga/0.1` information in the bundle:

```
"pkga/0.1@#f096d7d54098b7ad7012f9435d9c33f3": {
    "packages": [{
            "package_id": "3475bd55b91ae904ac96fde0f106a136ab951a5e",
            "lockfiles": {
                "app1_windows.lock": [
                    "3"
                ],
                "app2_windows.lock": [
                    "3"
                ]
            },
            "prev": null,
            "modified": null
        }
    ]
}
```

Now we can see that for one `package_id` there are actually 2 different lockfiles that require it. Both `app1` and `app2`

depend in this case on `pkga/0.1`. This is the information that can be used to avoid duplicated builds.

Using a lockfile bundle to build

The lockfile bundles also can compute a "build order" over the bundle, that will give an ordered list of lists of the package references that need to be built. In our case we could do:

```
$ conan lock bundle build-order lock.bundle --json=build_order.json
[
    ["pkga/0.1@#f096d7d54098b7ad7012f9435d9c33f3"],
    ["pkgb/0.1@#cd8f22d6f264f65398d8c534046e8e20", "pkgb/0.2@
↪#cd8f22d6f264f65398d8c534046e8e20"],
    ["app1/0.1@#584778f98ba1d0eb7c80a5ae1fe12fe2", "app2/0.1@
↪#3850895c1eac8223c43c71d525348019"]
]
```

The result is a list of lists. Every inner list is a "level", it is formed by mutually independent references that can be built in parallel, because they don't depend on each other. But every level will have dependencies to the previous levels, so it is necessary to build those levels in order.

The build order list can be iterated, building the packages in order. The necessary information is in the bundle file itself, so we can read it and use it, something like:

```
# Get the build order
build_order = json.loads(open("build_order.json").read())

# Read the bundle
bundle = json.loads(open("lock.bundle").read())
bundle = bundle["lock_bundle"]
for level in build_order:  # iterate the build_order
    for ref in level:  # All refs in this level could be built in parallel
        # Now get the package_ids and lockfile information
        package_ids = bundle[ref]["package_id"]
        for pkg_id, info in package_ids.items():
            lockfiles = info["lockfiles"]
            lockfile = next(iter(sorted(lockfiles)))  # Get the first one, all should be␣
↪valid to build same packag_id

            os.system("conan install {ref} --build={ref} --lockfile={lockfile} "
                      "--lockfile-out={lockfile}".format(ref=ref, lockfile=lockfile))
            os.system("conan lock bundle update lock.bundle")
```

This works under the hypothesis that the same binary, identified by the same `package_id` will be obtained irrespective of which lockfile or final product is used to build it. If this doesn't hold true, then the `package_id` policies should be revised until this condition is met.

Important: Recall that this is an orchestration mechanism, that can be used to distribute the actual `conan install` tasks to different agents, based on the lockfile itself, we might need some logic to send that build to one or another build server. If we didn't want to orchestrate and everything can be built in this machine a `conan install app1/1.1@ --lockfile={lockfile} --build=missing` would build all the necessary dependencies in the graph, in the current agent.

Note that the builds themselves are using regular lockfiles. The bundle does not contain the necessary information to reproduce the dependency graph that is needed to create packages.

The command `conan lock bundle update lock.bundle` manages to update all the connected lockfiles after a reference has been built. When the build is fired, it is done using 1 of the lockfiles, for a given configuration. That lockfile will get the updated package revision and status. The `conan lock bundle update` does this process in 2 steps:

- Scan all connected lockfiles for every `ref` recipe reference and `package_id`, and collect those that have been modified.
- Propagate the modified information to all the other connected lockfiles.

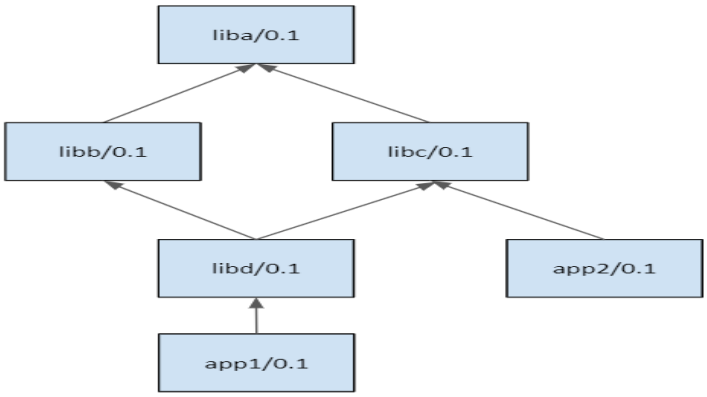
After `conan lock bundle update`, all packages sharing the same reference and `package_id` should have the same status (marked "modified" and same package revision). The "modified" state for the lockfile bundles can be cleaned using the command `conan lock bundle clean-modified` that will clean that flag from both the *.bundle* file and the individual *.lock* files.

11.4.6 Lockfiles in Continuous Integration

Warning: This is an **experimental** feature subject to breaking changes in future releases.

This section provides an example of application of the **lockfiles** in a Continuous Integration case. It doesn't aim to present a complete solution or the only possible one, depending on the project, the team, the requirements, the constraints, etc., other approaches might be recommended.

In this section we are going to use the same packages than in the previous one, defining this dependency graph.



The example scenario is a developer doing some changes in `libb`, that include bumping the version to `libb/0.2`. We will structure the CI in two parts:

- Building `libb/0.2@user/testing` to check that it is working fine.
- Building the downstream applications `app1/0.1@user/testing` and `app2/0.2@user/testing` to check if they build correctly, or if they are broken by those changes.

Note: The code used in this section, including a *build.py* script to reproduce it, is in the examples repository: https://github.com/conan-io/examples. You can go step by step reproducing this example while reading the below documentation.

```
$ git clone https://github.com/conan-io/examples.git
$ cd features/lockfiles/ci
# $ python build.py only to run the full example, but better go step by step
```

The example in this section uses `full_version_mode`, that is, if a package changes any part of its version, its consumers will need to build a new binary because a new `package_id` will be computed.

```
$ conan config set general.default_package_id_mode=full_version_mode
```

This sets the *default* package ID mode. Be aware, however, that if any of your packages provide their own *package_id()* implementation, for example explicitly setting a different mode for a dependency, *full_version_mode* might not be used for that package.

This example will use version ranges, and it is not necessary to have revisions enabled. It also does not require a server, everything can be reproduced locally, although the usage of different repositories will be introduced.

Repositories

When a developer does some changes, the CI wants to build those changes, create packages, and check if everything is ok. But while checking it, it is better to not pollute the main Conan remote repository with temporary packages until we are fully sure that it is not breaking anything. So we could use 2 repositories:

- `conan-develop`: this would be the team/project reference repository. Developers and CI will use this by default to retrieve Conan packages with precompiled binaries. Similarly to a git "develop" branch, it could be assumed that the packages in this repository work correctly, have been tested before being put there. It could also be expected that the repository contains pre-compiled binaries, so building from sources shouldn't be necessary.
- `conan-build`: a repository mainly for CI purposes. When CI is creating packages in a pipeline, it can put those packages in this repository, so they can still be used in the CI pipelines, be fetched by some build agents to build other packages. These temporary packages will not disrupt the operations and usage of `conan-develop` repository used by other CI jobs and developers.

Let's create the first version of the packages, for both Debug and Release configurations:

```
$ conan create liba liba/0.1@user/testing -s build_type=Release
$ conan create libb libb/0.1@user/testing -s build_type=Release
$ conan create libc libc/0.1@user/testing -s build_type=Release
$ conan create libd libd/0.1@user/testing -s build_type=Release
$ conan create app1 app1/0.1@user/testing -s build_type=Release
$ conan create app2 app2/0.1@user/testing -s build_type=Release
$ conan create liba liba/0.1@user/testing -s build_type=Debug
...
```

Now let's say that one developer does some change to `libb`:

```
$ vim libb/conanfile.py
# do some changes and save
```

These changes are local in this example, in reality they will be typically in the form of a Pull Request, wanting to merge those changes in the main "develop" branch.

Package pipeline

The first thing the CI will do is to build `libb/0.2@user/testing` package, containing the developer changes, for different configurations. As we want to make sure that all different configurations are built with the same versions of the dependencies, the first thing is to capture a "base" lockfile of the dependencies of `libb`:

```
$ cd libb
$ conan lock create conanfile.py --name=libb --version=0.2 --user=user --channel=testing
  --lockfile-out=../locks/libb_deps_base.lock --base
```

This will capture the *libb_deps_base.lock* file with the versions of `libb` dependencies, in this case `liba/0.1@user/testing`. Now that we have this file, new versions of `liba` could be created, but they will not be used:

```
$ cd ..
$ conan create liba liba/0.2@user/testing
```

We want to test the changes for several different configurations, so the first step would be to derive a new lockfile for each configuration/profile from the *libb_deps_base.lock*:

```
$ cd libb

# Derive one lockfile per profile/configuration
$ conan lock create conanfile.py --name=libb --version=0.2
  --user=user --channel=testing --lockfile=../locks/libb_deps_base.lock
  --lockfile-out=../locks/libb_deps_debug.lock -s build_type=Debug
$ conan lock create conanfile.py --name=libb --version=0.2
  --user=user --channel=testing --lockfile=../locks/libb_deps_base.lock
  --lockfile-out=../locks/libb_deps_release.lock

# Create the package binaries, one with each lockfile
$ conan create . libb/0.2@user/testing --lockfile=../locks/libb_deps_release.lock
$ conan create . libb/0.2@user/testing --lockfile=../locks/libb_deps_debug.lock
```

Note: It is important to note that it is not necessary to build all configurations in this build agent. One of the advantages of using lockfiles is that the build can be delegated to other agents, as long as they get the right commit of `libb` repo and the lockfile, they can build the desired package with the right dependencies.

Once everything is building ok, and `libb/0.2@user/testing` package is created correctly for all profiles, we want to check if this new version can be integrated safely in its consumers. When using revisions (not this example), it is important to capture the recipe revision, and lock it too. We can capture the recipe revision doing an export, creating a new *libb_base.lock* lockfile:

```
$ conan export . libb/0.2@user/testing --lockfile=../locks/libb_deps_base.lock
  --lockfile-out=../locks/libb_base.lock
```

Products pipeline

There is an important question to be addressed: **when a package changes, what other packages consuming it should be rebuilt to account for this change?**. The problem might be harder than it seems at first sight, or from the observation of the graph above. It shows that `libd/0.1` has a dependency to `libb/0.1`, does it mean that a new `libb/0.2` should produce a re-build of `libd/0.1` to link with the new version? Not always, if `libd` had a pinned dependency and not a version range, it will never resolve to the new version, and then it doesn't and it cannot be rebuilt unless some developer makes some changes to `libd` and bumps the requirement.

In this example, `libd` contains a version range, and if we evaluate it, we will see that the new `libb/0.2` version lies within the range, and then yes, it needs a new binary to be built, otherwise our repository of packages will have missing binaries.

One important problem is the combinatoric explosion that happens downstream. Projects evolve and packages will eventually have many versions and even many revisions. In our example, we could have in our repository many `libd/0.0.1`, `libd/0.0.2`, ..., `libd/0.0.34` versions, all of them with a requirement to `libb`. Each one could be in turn consumed by multiple `app1` versions.

We could think to consider as consumer only the latest version of `libd`. But it is also totally possible that some developer has already uploaded a `libd/2.0` version, with a breaking new API, aimed for the next major version of `app1`.

So the only alternative to be both efficient and have a robust Continuous Integration of changes in our core "products" is to explictly define those "products". In our case we will define that our products are `app1/0.1@user/testing` and `app2/0.1@user/testing`. This product definition could change as we keep doing releases of our products to our customers.

The first step in the products pipeline would be to capture the lockfiles for the different configurations we want to build for our products. As explained above, we can first capture a "base" lockfile of `app1/0.1@user/testing`, using the previous *libb_base.lock*, to make sure that we are using the locked versions for both `libb/0.2@user/testing` and `liba/0.1@user/testing`, as this was the snapshot of existing versions when the CI pipeline started, even if later a `liba/0.2@user/testing` was created.

```
$ conan lock create --reference=app1/0.1@user/testing --lockfile=locks/libb_base.lock
  --lockfile-out=locks/app1_base.lock --base
```

The *app1_base.lock* lockfile will capture and lock `libd/0.1@user/testing` and `libc/0.1@user/testing`. Now, even if those packages also got new versions, they will not be used, even if they fit in the version range. The *app1_base.lock* lockfile can be in turn used to capture complete lockfiles, one per profile/configuration:

```
$ conan lock create --reference=app1/0.1@user/testing --lockfile=locks/app1_base.lock
  --lockfile-out=locks/app1_release.lock
$ conan lock create --reference=app1/0.1@user/testing --lockfile=locks/app1_base.lock
  --lockfile-out=locks/app1_debug.lock -s build_type=Debug
```

The build-order can now be computed, also for each configuration:

```
$ conan lock build-order locks/app1_release.lock --json=bo_release.json
[[['libd/0.1@user/testing', 'b03c813b34cfab7a095fd903f7e8df2114e2b858', 'host', '4']],
 [['app1/0.1@user/testing', '15d2c695ed8d421c0d8932501fc654c8083e6582', 'host', '3']]]

$ conan lock build-order locks/app1_debug.lock --json=bo_debug.json
[[['libd/0.1@user/testing', '67a26cfbef78ad4905bec085664768c209d14fda', 'host', '4']],
 [['app1/0.1@user/testing', '680239a70c97f93d4d3dba4dec1b148d45ed087a', 'host', '3']]]
```

The build order tells that we need to build `libd/0.1@user/testing` and `app1/0.1@user/testing` in that order, for both Release and Debug configurations (again this can also be delegated to other build agents)

That build can be done with command:

```
$ conan install libd/0.1@user/testing --build=libd/0.1@user/testing --lockfile=locks/
↪app1_release.lock
  --lockfile-out=locks/app1_release_updated.lock
```

Note that we are creating a new temporary *app1_release_updated.lock* lockfile, that will contain and lock the binary produced by the build of `libd`. If this was implemented in CI, the *app1_release.lock* would be sent to the build agent, and it would return a modified *app1_release_updated.lock*. The way to integrate this information into the existing lockfile, necessary to keep building other downstream packages is:

```
$ conan lock update locks/app1_release.lock locks/app1_release_updated.lock
```

Now that *locks/app1_release.lock* is updated we could launch in exactly the same way the build of `app1`:

```
$ conan install app1/0.1@user/testing --build=app1/0.1@user/testing --lockfile=locks/
↪app1_release.lock
  --lockfile-out=locks/app1_release_updated.lock
```

The process will be repeated (or it could also run in parallel) for the Debug configuration.

After the `app1/0.1@user/testing` product pipeline finishes, then the `app2/0.2@user/testing` one will be started. With this setup and example, it is very important that the products pipelines are ran sequentially, otherwise it is possible that the same binaries are unnecesarily built more than once.

When the products pipeline finishes it means that the changes proposed by the developer in their Pull Request that would result in a new `libb/0.2@user/testing` package are safe to be merged and will be integrated in our product packages without problems. When the Pull Request is merged there might be two alternatives:

- The merge is a merge commit, with a different revision and possible different source as the result of a real merge, than the source used in this CI job. Then it is necessary to fire again a new job that will build these packages.
- If the merge is a clean fast-forward, then the packages that were built in this job would be valid, and could be copied from the repository `conan-build` to the `conan-develop`.

After the `app1` lockfile is created it could be possible to install all the binaries referenced in that lockfile using the **`conan lock install`**:

```
$ conan lock install app1_release_updated.lock -g deploy
```

It is also possible to use this command for just installing the recipes but not the binaries adding the `--recipes` argument:

```
$ conan lock install app1_release_updated.lock --recipes
```

CHAPTER

TWELVE

MASTERING CONAN

This section provides an introduction to important productivity and useful features of Conan:

12.1 Use conanfile.py for consumers

You can use a `conanfile.py` for installing/consuming packages, even if you are not creating a package with it. You can also use the existing `conanfile.py` in a given package while developing it to install dependencies. There's no need to have a separate `conanfile.txt`.

Let's take a look at the complete `conanfile.txt` from the previous *timer* example with POCO library, in which we have added a couple of extra generators

```
[requires]
poco/1.9.4

[generators]
gcc
cmake
txt

[options]
poco:shared=True
openssl:shared=True

[imports]
bin, *.dll -> ./bin # Copies all dll files from the package "bin" folder to my project
↪"bin" folder
lib, *.dylib* -> ./bin # Copies all dylib files from the package "lib" folder to my␣
↪project "bin" folder
```

The equivalent `conanfile.py` file is:

```
from conans import ConanFile, CMake

class PocoTimerConan(ConanFile):
    settings = "os", "compiler", "build_type", "arch"
    requires = "poco/1.9.4" # comma-separated list of requirements
    generators = "cmake", "gcc", "txt"
    default_options = {"poco:shared": True, "openssl:shared": True}
```

(continues on next page)

(continued from previous page)

```
    def imports(self):
        self.copy("*.dll", dst="bin", src="bin") # From bin to bin
        self.copy("*.dylib*", dst="bin", src="lib") # From lib to bin
```

Note that this `conanfile.py` doesn't have a name, version, or `build()` or `package()` method, as it is not creating a package. They are not required.

With this `conanfile.py` you can just work as usual. Nothing changes from the user's perspective. You can install the requirements with (from mytimer/build folder):

```
$ conan install ..
```

12.1.1 conan build

One advantage of using `conanfile.py` is that the project build can be further simplified, using the conanfile.py `build()` method.

If you are building your project with CMake, edit your `conanfile.py` and add the following `build()` method:

```
from conans import ConanFile, CMake

class PocoTimerConan(ConanFile):
    settings = "os", "compiler", "build_type", "arch"
    requires = "poco/1.9.4"
    generators = "cmake", "gcc", "txt"
    default_options = {"poco:shared": True, "openssl:shared": True}

    def imports(self):
        self.copy("*.dll", dst="bin", src="bin") # From bin to bin
        self.copy("*.dylib*", dst="bin", src="lib") # From lib to bin

    def build(self):
        cmake = CMake(self)
        cmake.configure()
        cmake.build()
```

Then execute, from your project root:

```
$ conan install . --install-folder build
$ conan build . --build-folder build
```

The **`conan install`** command downloads and prepares the requirements of your project (for the specified settings) and the **`conan build`** command uses all that information to invoke your `build()` method to build your project, which in turn calls `cmake`.

This **`conan build`** will use the settings used in the **`conan install`** which have been cached in the local *conaninfo.txt* and file in your build folder. This simplifies the process and reduces the errors of mismatches between the installed packages and the current project configuration. Also, the *conanbuildinfo.txt* file contains all the needed information obtained from the requirements: `deps_cpp_info`, `deps_env_info`, `deps_user_info` objects.

If you want to build your project for **x86** or another setting just change the parameters passed to **`conan install`**:

```
$ conan install . --install-folder build_x86 -s arch=x86
$ conan build . --build-folder build_x86
```

Implementing and using the conanfile.py `build()` method ensures that we always use the same settings both in the installation of requirements and the build of the project, and simplifies calling the build system.

12.1.2 Other local commands

Conan implements other commands that can be executed locally over a consumer `conanfile.py` which is in user space, not in the local cache:

- **`conan source <path>`**: Execute locally the *conanfile.py* `source()` method.
- **`conan package <path>`**: Execute locally the *conanfile.py* `package()` method.

These commands are mostly used for testing and debugging while developing a new package, before **exporting** such package recipe into the local cache.

See also:

Check the section *Reference/Commands* to find out more.

12.2 Conditional settings, options and requirements

Remember, in your `conanfile.py` you can use the value of your options to:

- Add requirements dynamically
- Change values of other options
- Assign values to options of your requirements

The `configure()` method might be used to hardcoded values for options of the requirements. It is strongly discouraged to use it to change the settings values. Please remember that `settings` are a configuration *input*, so it doesn't make sense to modify it in the recipes.

Also, for options, a more flexible solution is to define dependencies options values in the `default_options`, not in the `configure()` method. Setting the values in `configure()` won't allow to override them and it will make really hard (even impossible) to resolve some conflicts. Use it only when it is absolutely necessary that the package dependencies use those options.

Here is an example of what we could do in our **configure method**:

```
class Recipe(ConanFile):
  ...
  requires = "poco/1.9.4" # We will add OpenSSL dynamically "openssl/1.0.2t"
  ...

  def configure(self):
      # We can control the options of our dependencies based on current options
      self.options["openssl"].shared = self.options.shared

      # Maybe in windows we know that OpenSSL works better as shared (false)
      if self.settings.os == "Windows":
         self.options["openssl"].shared = True

         # Or adjust any other available option
         self.options["poco"].other_option = "foo"
```

(continues on next page)

(continued from previous page)

```
    # We could check the presence of an option
    if "shared" in self.options:
        pass

def requirements(self):
    # Or add a new requirement!
    if self.options.testing:
       self.requires("OpenSSL/2.1@memsharded/testing")
    else:
       self.requires("openssl/1.0.2u")

def build(self):
    # We can check the final values of options of our requirements
    if self.options['poco'].that_option != "bar":
        raise ConanInvalidConfiguration("Who modified this option?!")
```

12.2.1 Constrain settings and options

Sometimes there are libraries that are not compatible with specific settings like libraries that are not compatible with an architecture, or options that only make sense for an operating system. It can also be useful when there are settings under development.

There are two approaches for this situation:

- **Use `validate()` to raise an error for non-supported configurations**:

  This approach is the first one evaluated when Conan loads the recipe so it is quite handy to perform checks of the input settings. It relies on the set of possible settings inside your *settings.yml* file, so it can be used to constrain any recipe.

  ```
  from conans.errors import ConanInvalidConfiguration
  ...
  def validate(self):
      if self.settings.os == "Windows":
        raise ConanInvalidConfiguration("This library is not compatible with Windows")
  ```

  Tip: Use the *Invalid configuration* exception to make Conan return with a special error code. This will indicate that the configuration used for settings or options is not supported.

  This same method is also valid for `options` and `config_options()` method and it is commonly used to remove options for one setting:

  ```
  def config_options(self):
      if self.settings.os == "Windows":
          del self.options.fPIC
  ```

Note: For managing invalid configurations, please check the new experimental `validate()` method (*validate()*).

- **Constrain settings inside a recipe**:

This approach constrains the settings inside a recipe to a subset of them, and it is normally used in recipes that are never supposed to work out of the restricted settings.

```
from conans import ConanFile

class MyConan(ConanFile):
    name = "myconanlibrary"
    version = "1.0.0"
    settings = {"os": None, "build_type": None, "compiler": None, "arch": ["x86_64
↪"]}
```

The disadvantage of this is that possible settings are hardcoded in the recipe, and in case new values are used in the future, it will require the recipe to be modified explicitly.

Important: Note: the use of the `None` value in the `os`, `compiler` and `build_type` settings described above will allow them to take the values from *settings.yml* file

---

We strongly recommend the use of the first approach whenever it is possible, and use the second one only for those cases where a stronger constrain is needed for a particular recipe.

See also:

Check the reference section *configure(), config_options()* to find out more.

12.3 Build policies

By default, **conan install** command will search for a binary package (corresponding to our settings and defined options) in a remote. If it's not present the install command will fail.

As previously demonstrated, we can use the **--build** option to change the default **conan install** behavior:

- **--build some_package** will build only "some_package".
- **--build missing** will build only the missing requires.
- **--build** will build all requirements from sources.
- **--build outdated** will try to build from code if the binary is not built with the current recipe or when missing binary package.
- **--build cascade** will build from code all the nodes with some dependency being built (for any reason). Can be used together with any other build policy. Useful to make sure that any new change introduced in a dependency is incorporated by building again the package.
- **--build pattern*** will build only the packages with the reference starting with "pattern".
- **--build=* --build=!some_package1 --build=!some_package2** will build all requirements from sources, except for some_package1 and some_package2.

With the `build_policy` attribute in the *conanfile.py* the package creator can change the default Conan's build behavior. The allowed build_policy values are:

- `missing`: If no binary package is found, Conan will build it without the need to invoke Conan install with **--build missing** option.
- `always`: The package will be built always, **retrieving each time the source code** executing the "source" method.

- `never`: (experimental, available from Conan 1.37) Never builds this package from source, this package can only be created with a `conan export-pkg` command.

```
class PocoTimerConan(ConanFile):
    settings = "os", "compiler", "build_type", "arch"
    requires = "poco/1.9.4" # comma-separated list of requirements
    generators = "cmake", "gcc", "txt"
    default_options = {"poco:shared": True, "poco:shared": True}
    build_policy = "always" # "missing"
```

These build policies are especially useful if the package creator doesn't want to provide binary package; for example, with header only libraries.

The `always` policy will retrieve the sources each time the package is installed, so it can be useful for providing a "latest" mechanism or ignoring the uploaded binary packages.

The package pattern can be referred as a case-sensitive fnmatch pattern of the package name or the full package reference. e.g **`--build poco`**, **`--build poc*`**, **`--build zlib/*`**, **`--build *@conan/stable`** or **`--build zlib/1.2.11`**.

12.4 Environment variables

There are several use cases for environment variables:

- Conan global configuration environment variables (e.g. `CONAN_COMPRESSION_LEVEL`). They can be configured in *conan.conf* or as system environment variables, and control Conan behavior.
- Package recipes can access environment variables to determine their behavior. A typical example would be when launching CMake. It will check for CC and CXX environment variables to define the compiler to use. These variables are mostly transparent for Conan, and just used by the package recipes.
- **Environment variables can be set in different ways:**
  - global, at the OS level, with **`export VAR=Value`** or in Windows **`SET VAR=Value`**.
  - In the Conan command line: **`conan install -e VAR=Value`**.
  - In profile files.
  - In package recipes in the `self.env_info` field, so they are activated for dependent recipes.

12.4.1 Defining environment variables

You can use *profiles* to define environment variables that will apply to your recipes. You can also use **`-e`** parameter in **`conan install`**, **`conan info`** and **`conan create`** commands.

```
[env]
CC=/usr/bin/clang
CXX=/usr/bin/clang++
```

If you want to override an environment variable that a package has inherited from its requirements, you can use either **profiles** or **`-e`** to do it:

```
conan install . -e mypkg:PATH=/other/path
```

If you want to define an environment variable, but you want to append the variables declared in your requirements, you can use the `[]` syntax:

```
$ conan install . -e PATH=[/other/path]
```

This way the first entry in the PATH variable will be */other/path*, but the PATH values declared in the requirements of the project will be appended at the end using the system path separator.

12.4.2 Automatic environment variables inheritance

If your dependencies define some env_info variables in the package_info() method, they will be automatically applied before calling the consumer *conanfile.py* methods source(), build(), package() and imports(). You can read more about env_info object *here*.

For example, if you are creating a package for a tool, you can define the variable PATH:

```
class ToolExampleConan(ConanFile):
    name = "my_tool_installer"
    ...

    def package_info(self):
        self.env_info.path.append(os.path.join(self.package_folder, "bin"))
```

If another Conan recipe requires the *my_tool_installer* in the source(), build(), package() and imports(), the bin folder of the my_tool_installer package will be automatically appended to the system PATH. If my_tool_installer packages an executable called my_tool_executable in the *bin* of the package folder, we can directly call the tool because it will be available in the path:

```
class MyLibExample(ConanFile):
    name = "my_lib_example"
    ...

    def build(self):
        self.run(["my_tool_executable", "some_arguments"])
```

You could also set CC, CXX variables if we are packing a compiler to define what compiler to use or any other environment variable. Read more about tool packages *here*.

12.5 Virtual Environments

Conan offers three special Conan generators to create virtual environments:

- virtualenv: Declares the *self.env_info* variables of the requirements.
- virtualbuildenv: Special build environment variables for autotools/visual studio.
- virtualrunenv: Special environment variables to locate executables and shared libraries in the requirements.

These virtual environment generators create two executable script files (.sh or .bat depending on the current operating system), one to activate the virtual environment (set the environment variables) and one to deactivate it.

You can aggregate two or more virtual environments, that means that you can activate a virtualenv and then activate a virtualrunenv so you will have available the environment variables declared in the env_info object of the requirements plus the special environment variables to locate executables and shared libraries.

12.5.1 Virtualenv generator

Conan provides a **virtualenv** generator, able to read from each dependency the *self.env_info* variables declared in the `package_info()` method and generate two scripts "activate" and "deactivate". These scripts set/unset all env variables in the current shell.

Example:

The recipe of `cmake/3.16.3` appends to the PATH variable the package folder/bin.

You can check existing CMake Conan package versions in *conancenter* with:

```
$ conan search cmake* -r=conancenter
```

In the **bin** folder there is a **cmake** executable:

```
def package_info(self):
  self.env_info.path.append(os.path.join(self.package_folder, "bin"))
```

Let's prepare a virtual environment to have cmake available in the path. Open `conanfile.txt` and change (or add) **virtualenv** generator:

```
[requires]
cmake/3.16.3

[generators]
virtualenv
```

Run **`conan install`**:

```
$ conan install .
```

You can also avoid the creation of the *conanfile.txt* completely and directly do:

```
$ conan install cmake/3.16.3 -g=virtualenv
```

Activate the virtual environment, and now you can run `cmake --version` to check that you have the installed CMake in path.

```
$ source activate.sh # Windows: activate.bat without the source
$ cmake --version
```

Two sets of scripts are available on all platforms - `activate.sh`/`deactivate.sh` and `activate.ps1`/`deactivate.ps1` if you are using powershell. In addition Windows has `activate.bat`/`deactivate.bat` Deactivate the virtual environment (or close the console) to restore the environment variables:

```
$ source deactivate.sh # Windows: deactivate.bat or deactivate.ps1 without the source
```

See also:

Read the Howto *Create installer packages* to learn more about the virtual environment feature. Check the section *Reference/virtualenv* to see the generator reference.

12.5.2 Virtualbuildenv environment

Use the generator `virtualbuildenv` to activate an environment that will set the environment variables for Autotools and Visual Studio.

The generator will create `activate_build` and `deactivate_build` files.

See also:

Read More about the building environment variables defined in the sections *Building with autotools* and *Build with Visual Studio*.

Check the section *Reference/virtualbuildenv* to see the generator reference.

12.5.3 Virtualrunenv generator

Use the generator `virtualrunenv` to activate an environment that will:

- Append to `PATH` environment variable every `bin` folder of your requirements.
- Append to `LD_LIBRARY_PATH` and `DYLD_LIBRARY_PATH` environment variables each `lib` folder of your requirements.

The generator will create `activate_run` and `deactivate_run` files. This generator is especially useful:

- If you are requiring packages with shared libraries and you are running some executable that needs those libraries.
- If you have a requirement with some tool (executable) and you need it in the path.

In the previous example of the `cmake` recipe, even if the cmake package doesn't declare the `self.env_info.path` variable, using the virtualrunenv generator, the `bin` folder of the package will be available in the PATH. So after activating the virtual environment we could just run `cmake` in order to execute the package's cmake.

See also:

- *Reference/Tools/environment_append*

12.6 Logging

12.6.1 How to log and debug a conan execution

You can use the *CONAN_TRACE_FILE* environment variable to log and debug several Conan command execution. Set the `CONAN_TRACE_FILE` environment variable pointing to a log file.

Example:

```
export CONAN_TRACE_FILE=/tmp/conan_trace.log # Or SET in windows
conan install zlib/1.2.8@lasote/stable
```

The */tmp/conan_trace.log* file:

```
{"_action": "COMMAND", "name": "install", "parameters": {"all": false, "build": null,
↪"env": null, "file": null, "generator": null, "manifests": null, "manifests_interactive
↪": null, "no_imports": false, "options": null, "package": null, "profile": null,
↪"reference": "zlib/1.2.8@lasote/stable", "remote": null, "scope": null, "settings":␣
↪null, "update": false, "verify": null, "werror": false}, "time": 1485345289.250117}
{"_action": "REST_API_CALL", "duration": 1.8255090713500977, "headers": {"Authorization
```

(continues on next page)

(continued from previous page)


```
↪": "**********", "X-Client-Anonymous-Id": "**********", "X-Client-Id": "lasote2", "X-
↪Conan-Client-Version": "0.19.0-dev"}, "method": "GET", "time": 1485345291.092218, "url
↪": "https://server.conan.io/v1/conans/zlib/1.2.8/lasote/stable/download_urls"}
{"_action": "DOWNLOAD", "duration": 0.4136989116668701, "time": 1485345291.506399, "url
↪": "https://conanio-production.s3.amazonaws.com/storage/zlib/1.2.8/lasote/stable/
↪export/conanmanifest.txt"}
{"_action": "DOWNLOAD", "duration": 0.10367798805236816, "time": 1485345291.610335, "url
↪": "https://conanio-production.s3.amazonaws.com/storage/zlib/1.2.8/lasote/stable/
↪export/conanfile.py"}
{"_action": "DOWNLOAD", "duration": 0.059114933013916016, "time": 1485345291.669744, "url
↪": "https://conanio-production.s3.amazonaws.com/storage/zlib/1.2.8/lasote/stable/
↪export/conan_export.tgz"}
{"_action": "DOWNLOADED_RECIPE", "_id": "zlib/1.2.8@lasote/stable", "duration": 2.
↪40762996673584, "files": {"conan_export.tgz": "/home/laso/.conan/data/zlib/1.2.8/
↪lasote/stable/export/conan_export.tgz", "conanfile.py": "/home/laso/.conan/data/zlib/1.
↪2.8/lasote/stable/export/conanfile.py", "conanmanifest.txt": "/home/laso/.conan/data/
↪zlib/1.2.8/lasote/stable/export/conanmanifest.txt"}, "remote": "conan.io", "time":␣
↪1485345291.670017}
{"_action": "REST_API_CALL", "duration": 0.4844989776611328, "headers": {"Authorization
↪": "**********", "X-Client-Anonymous-Id": "**********", "X-Client-Id": "lasote2", "X-
↪Conan-Client-Version": "0.19.0-dev"}, "method": "GET", "time": 1485345292.160912, "url
↪": "https://server.conan.io/v1/conans/zlib/1.2.8/lasote/stable/packages/
↪c6d75a933080ca17eb7f076813e7fb21aaa740f2/download_urls"}
{"_action": "DOWNLOAD", "duration": 0.06388187408447266, "time": 1485345292.225308, "url
↪": "https://conanio-production.s3.amazonaws.com/storage/zlib/1.2.8/lasote/stable/
↪package/c6d75a933080ca17eb7f076813e7fb21aaa740f2/conaninfo.txt?
↪Signature=c1KAOqvxtCUnnQOeYizZ9bgcwwY%3D&Expires=1485352492&
↪AWSAccessKeyId=AKIAJXMWDMVCDMAZQK5Q"}
{"_action": "REST_API_CALL", "duration": 0.8182470798492432, "headers": {"Authorization
↪": "**********", "X-Client-Anonymous-Id": "**********", "X-Client-Id": "lasote2", "X-
↪Conan-Client-Version": "0.19.0-dev"}, "method": "GET", "time": 1485345293.044904, "url
↪": "https://server.conan.io/v1/conans/zlib/1.2.8/lasote/stable/packages/
↪c6d75a933080ca17eb7f076813e7fb21aaa740f2/download_urls"}
{"_action": "DOWNLOAD", "duration": 0.07849907875061035, "time": 1485345293.123831, "url
↪": "https://conanio-production.s3.amazonaws.com/storage/zlib/1.2.8/lasote/stable/
↪package/c6d75a933080ca17eb7f076813e7fb21aaa740f2/conanmanifest.txt"}
{"_action": "DOWNLOAD", "duration": 0.06638002395629883, "time": 1485345293.190465, "url
↪": "https://conanio-production.s3.amazonaws.com/storage/zlib/1.2.8/lasote/stable/
↪package/c6d75a933080ca17eb7f076813e7fb21aaa740f2/conaninfo.txt"}
{"_action": "DOWNLOAD", "duration": 0.3634459972381592, "time": 1485345293.554206, "url
↪": "https://conanio-production.s3.amazonaws.com/storage/zlib/1.2.8/lasote/stable/
↪package/c6d75a933080ca17eb7f076813e7fb21aaa740f2/conan_package.tgz"}
{"_action": "DOWNLOADED_PACKAGE", "_id": "zlib/1.2.8@lasote/
↪stable:c6d75a933080ca17eb7f076813e7fb21aaa740f2", "duration": 1.3279249668121338,
↪"files": {"conan_package.tgz": "/home/laso/.conan/data/zlib/1.2.8/lasote/stable/
↪package/c6d75a933080ca17eb7f076813e7fb21aaa740f2/conan_package.tgz", "conaninfo.txt":
↪"/home/laso/.conan/data/zlib/1.2.8/lasote/stable/package/
↪c6d75a933080ca17eb7f076813e7fb21aaa740f2/conaninfo.txt", "conanmanifest.txt": "/home/
↪laso/.conan/data/zlib/1.2.8/lasote/stable/package/
↪c6d75a933080ca17eb7f076813e7fb21aaa740f2/conanmanifest.txt"}, "remote": "conan.io",
↪"time": 1485345293.554466}
```


In the traces we can see:

1. A command `install` execution.
2. A REST API call to get some `download_urls`.
3. Three files downloaded (corresponding to the previously retrieved urls).
4. `DOWNLOADED_RECIPE` tells us that the recipe retrieving is finished. We can see that the whole retrieve process took 2.4 seconds.
5. Conan client has computed the SHA for the needed binary package and will now retrieve it. So it will request and download the package `package_id` file to perform some checks like outdated binaries.
6. Another REST API call to get some more `download_urls`, for the package files and download them.
7. Finally we get a `DOWNLOADED_PACKAGE` telling us that the package has been downloaded. The download took 1.3 seconds.

If we execute conan install again:

```
export CONAN_TRACE_FILE=/tmp/conan_trace.log # Or SET in windows
conan install zlib/1.2.8@lasote/stable
```

The */tmp/conan_trace.log* file only three lines will be appended:

```
{"_action": "COMMAND", "name": "install", "parameters": {"all": false, "build": null,
↪"env": null, "file": null, "generator": null, "manifests": null, "manifests_interactive
↪": null, "no_imports": false, "options": null, "package": null, "profile": null,
↪"reference": "zlib/1.2.8@lasote/stable", "remote": null, "scope": null, "settings":␣
↪null, "update": false, "verify": null, "werror": false}, "time": 1485346039.817543}
{"_action": "GOT_RECIPE_FROM_LOCAL_CACHE", "_id": "zlib/1.2.8@lasote/stable", "time":␣
↪1485346039.824949}
{"_action": "GOT_PACKAGE_FROM_LOCAL_CACHE", "_id": "zlib/1.2.8@lasote/
↪stable:c6d75a933080ca17eb7f076813e7fb21aaa740f2", "time": 1485346039.827915}
```

1. A command `install` execution.
2. A `GOT_RECIPE_FROM_LOCAL_CACHE` because it's already stored in local cache.
3. A `GOT_PACKAGE_FROM_LOCAL_CACHE` because the package is cached too.

12.6.2 How to log the build process

You can log your command executions `self.run` in a file named *conan_run.log* using the environment variable *CONAN_LOG_RUN_TO_FILE*.

You can also use the variable *CONAN_PRINT_RUN_COMMANDS* to log extra information about the commands being executed.

Package the log files

The *conan_run.log* file will be created in your *build* folder so you can package it the same way you package a library file:

```
def package(self):
    self.copy(pattern="conan_run.log", dst="", keep_path=False)
```

12.7 Sharing the settings and other configuration

If you are using Conan in a company or in an organization, sometimes you need to share the *settings.yml* file, the *profiles*, or even the *remotes* or any other Conan local configuration with the team.

You can use the **conan config install**.

If you want to try this feature without affecting your current configuration, you can declare the CONAN_USER_HOME environment variable and point to a different directory.

Read more in the section *conan config install*.

12.8 Conan local cache: concurrency, Continuous Integration, isolation

Conan needs access to some per user configuration files, such as the **conan.conf** file that defines the basic client app configuration. By convention, this file will be located in the user home folder **~/.conan/**. This folder will also typically store the package cache in **~/.conan/data**. Even though the latter is configurable in *conan.conf*, Conan needs some place to look for this initial configuration file.

There are some scenarios in which you might want to use different initial locations for the Conan client application:

- Continuous Integration (CI) environments, in which multiple jobs can also work concurrently. Moreover, these environments would typically want to run with different user credentials, different remote configurations, etc. Note that using Continuous Integration with the same user, with isolated machine instances (virtual machines), or with sequential jobs is perfectly possible. For example, we use a lot CI cloud services of travis-ci and appveyor.
- Independent per project management and storage. If as a single developer you want to manage different projects with different user credentials and/or different remotes, you might find that having multiple independent caches makes it easier.

Using different caches is very simple. You can just define the environment variable **CONAN_USER_HOME**. By setting this variable to different paths, you have multiple conan caches, something like python "virtualenvs". Just changing the value of **CONAN_USER_HOME**, you can switch among isolated Conan instances that will have independent package storage caches, and also different user credentials, different user default settings, and different remotes configuration.

Note: Use an absolute path or a path starting with ~/ (relative to user home). In Windows do not use quotes.

Windows users:

```
$ SET CONAN_USER_HOME=c:\data
$ conan install . # call conan normally, config & data will be in c:\data\.conan
```

Linux/macOS users:

```
$ export CONAN_USER_HOME=/tmp/conan
$ conan install . # call conan normally, config & data will be in /tmp/conan/.conan
```

You can now:

- Build concurrent jobs, parallel builds in Continuous Integration or locally, by just setting the variable before launching Conan commands.

- You can test locally different user credentials, default configurations, or different remotes, just by switching from one cache to another.

```
$ export CONAN_USER_HOME=/tmp/conan
$ conan search  # using that /tmp/conan cache
$ conan user  # using that /tmp/conan cache

$ export CONAN_USER_HOME=/tmp/conan2
$ conan search  # different packages
$ conan user  # can be different users

$ export CONAN_USER_HOME=/tmp/conan  # just go back to use the other cache
```

12.8.1 Concurrency

Conan local cache support some degree of concurrency, allowing simultaneous creation or installation of different packages, or building different binaries for the same package. However, concurrent operations like removal of packages while creating them will fail. If you need different environments that operate totally independently, you probably want to use different Conan caches for that.

The concurrency is implemented with a Readers-Writers lock mechanism, which in turn uses `fasteners` library file locks to achieve multi-platform portability. As this "mutex" resource is by definition not enough to implement a Readers-Writers solution, some active-wait with time sleeps in a loop is necessary. However, this time sleeps will be rare, only sleeping when there is actually a collision and waiting on a lock.

The lock files will be stored inside each `Pkg/version/user/channel` folder in the local cache, in a `rw` file for locking the entire package, or in a set of locks (one per each different binary package, under a subfolder called `locks`, with each lock named with the binary ID of the package).

It is possible to disable the locking mechanism in `conan.conf`:

```
[general]
cache_no_locks = True
```

12.8.2 System Requirements

When `system_requirements()` runs, Conan creates the `system_reqs` folder. This folder could be created individually by package id or globally when `global_system_requirements` is **True**.

However, sometimes you want to run `system_requirements()` again for a specific package, so you could either remove the `system_reqs.txt` file for the specific package id, or you could *remove system_reqs globally for the package name referred*.

CHAPTER

THIRTEEN

SYSTEMS AND CROSS BUILDING

This section explains how to approach a cross building scenario with Conan and how to use the Windows subsystems (Cygwin, MSYS2).

Todo: Maybe we should divide this section, create one for the general cross building problem and a different one to talk about Windows subsystems.

13.1 Cross building

Cross building (or cross compilation) is the process of generating binaries for a platform that is not the one where the compiling process is running.

Cross compilation is mostly used to build software for an alien device, such as an embedded device where you don't have an operating system nor a compiler available. It's also used to build software for slower devices, like an Android machine or a Raspberry Pi where running the native compilation will take too much time.

In order to cross build a codebase the right toolchain is needed, with a proper compiler (cross compiler), a linker and the set of libraries matching the `host` platform.

13.1.1 GNU triplet convention

According to the GNU convention, there are three platforms involved in the software building:

- **Build platform:** The platform on which the compilation tools are being executed.
- **Host platform:** The platform on which the generated binaries will run.
- **Target platform:** Only when building a cross compiler, it is the platform it will generate binaries for.

Depending on the values of these platforms, there are different scenarios:

- **Native building**: when the `build` and the `host` platforms are the same, it means that the platform where the compiler is running is the same one where the generated binaries will run. This is the most common scenario.
- **Cross building**: when the `build` and the `host` platform are different, it requires a cross compiler running in the build platform that generates binaries for the host platform.

The `target` platform plays an important role when compiling a cross compiler, in that scenario the `target` is the platform the compiler will generate binaries for: in order to be a cross compiler the `host` platform (where the cross compiler will run) has to be different from the `target` platform. If the `build` platform is also different, it is called **Canadian Cross**.

Let's illustrate these scenarios with some examples:

- The Android NDK is a cross compiler to Android: it can be executed in Linux (the `build` platform) to generate binaries for Android (the `host` platform).
- The Android NDK was once compiled, during that compilation a different compiler was used running in a `build` platform (maybe Windows) to generate the actual Android NDK that will run in the `host` platform Linux, and as we saw before, that Android NDK cross compiler will generate binaries for a `target` platform which is Android.

The values of the `build` , `host` and `target` platforms are not absolute, and they depend on the process we are talking about: the `host` when compiling a cross compiler turns into the `build` when using that same cross compiler, or the `target` of the cross compiler is the `host` platform when we are using it to build binaries.

See also:

One way to avoid this complexity is to run the compilation in the host platform, so both `build` and `host` will take the same value and it will be a *native compilation*.

13.1.2 Cross building with Conan

If you want to cross build a Conan package (for example using your Linux machine) to build the `zlib` Conan package for Windows, you need to tell Conan where to find your toolchain/cross compiler.

There are two approaches:

- Using a profile: install the toolchain in your computer and use a `profile` to declare the settings and point to the needed tools/libraries in the toolchain using the `[env]` section to declare, at least, the `CC` and `CXX` environment variables.
- Using tool requires: package the toolchain as a Conan package and include it as a `tool_requires`.

Using a profile

Using a Conan profile we can declare not only the `settings` that will identify our binary (`host` settings), but also all the environment variables needed to use a toolchain or cross compiler. The profile needs the following sections:

- A **[settings]** section containing the regular settings: `os`, `arch`, `compiler` and `build_type` depending on your library. These settings will identify your binary.
- An **[env]** section with a PATH variable pointing to your installed toolchain. Also any other variable that the toolchain expects (read the docs of your compiler). Some build systems need a variable `SYSROOT` to locate where the host system libraries and tools are.

For example, in the following profile we declare the `host` platform to be Windows x86_64 with the compiler, version and other settings we are using. And we add the **[env]** section with all the variables needed to use an installed toolchain:

```
toolchain=/usr/x86_64-w64-mingw32 # Adjust this path
target_host=x86_64-w64-mingw32
cc_compiler=gcc
cxx_compiler=g++

[env]
CONAN_CMAKE_FIND_ROOT_PATH=$toolchain  # Optional, for CMake to find things in that␣
↪folder
CONAN_CMAKE_SYSROOT=$toolchain  # Optional, if we want to define sysroot
CHOST=$target_host
AR=$target_host-ar
AS=$target_host-as
RANLIB=$target_host-ranlib
```

(continues on next page)

(continued from previous page)

```
CC=$target_host-$cc_compiler
CXX=$target_host-$cxx_compiler
STRIP=$target_host-strip
RC=$target_host-windres

[settings]
# We are cross-building to Windows
os=Windows
arch=x86_64
compiler=gcc

# Adjust to the gcc version of your MinGW package
compiler.version=7.3
compiler.libcxx=libstdc++11
build_type=Release
```

You can find working examples at the *bottom of this section*.

Using tool requires

Important: The tool requirement was formerly named "build requirement" and has been renamed to highlight that the usage of this kind of requirement must be for "tools" exclusively, not being valid for libraries to express a "private" require or other meanings.

Warning: This section refers to the **experimental feature** that is activated when using `--profile:build` and `--profile:host` in the command-line. It is currently under development, features can be added or removed in the following versions.

Instead of manually downloading the toolchain and creating a profile, you can create a Conan package with it. Starting with Conan v1.24 and the command line arguments `--profile:host` and `--profile:build` this should be a regular recipe, for older versions some more work is needed.

Conan v1.24 and newer

A recipe with a toolchain is like any other recipe with a binary executable:

```
import os
from conans import ConanFile

class MyToolchainXXXConan(ConanFile):
    name = "my_toolchain"
    version = "0.1"
    settings = "os", "arch", "compiler", "build_type"

    # Implement source() and build() as usual

    def package(self):
```

(continues on next page)

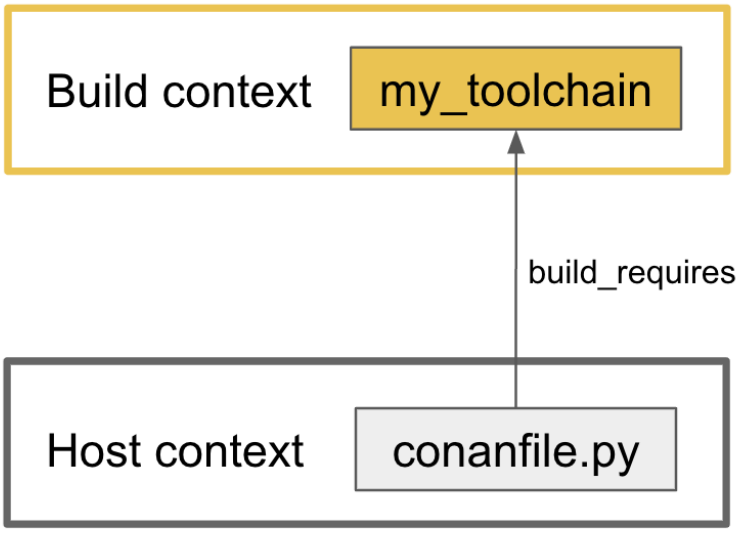
(continued from previous page)

```
        # Copy all the required files for your toolchain
        self.copy("*", dst="", src="toolchain")

    def package_info(self):
        bin_folder = os.path.join(self.package_folder, "bin")
        self.env_info.CC = os.path.join(bin_folder, "mycompiler-cc")
        self.env_info.CXX = os.path.join(bin_folder, "mycompiler-cxx")
        self.env_info.SYSROOT = self.package_folder
```

The Conan package with the toolchain needs to fill the `env_info` object in the *package_info()* method with the same variables we've specified in the examples above in the `[env]` section of profiles.

Then you will need to consume this recipe as any regular *tool requires* that belongs to the `build` context: you need to use the `--profile:build` argument in the command line while creating your library:

```
conan create path/to/conanfile.py --profile:build=profile_build --profile:host=profile_
↪host
```



The profile `profile_build` will contain just the settings related to your `build` platform, where you are running the command, and the `profile_host` will list the settings for the `host` platform (and eventually the `my_toolchain/0.1` as `tool_requires` if it is not listed in the recipe itself).

Conan will apply the appropiate profile to each recipe, and will inject the environment of all the tool requirements that belong to the `build` context before running the `build()` method of the libraries being compiled. That way, the environment variables `CC`, `CXX` and `SYSROOT` from `my_toolchain/0.1` will be available and also the path to the `bindirs` directory from that package.

The above means that **Conan is able to compile the full graph in a single execution**, it will compile the tool requires using the `profile_build` and then it will compile the libraries using the `host_profile` settings applying the environment of the former ones.

Starting with Conan v1.25 (if the user provides the `--profile:build`) it is possible to get the relative context where a recipe is running during a Conan invocation. The object instantiated from the recipe contains the following attributes:

- `self.settings` will always contain the settings corresponding to the binary to build/retrieve. It will contain the settings from the profile `profile_host` when this recipe appears in the `host` context and the settings from

the profile `profile:build` if this object belongs to the `build` context.

- `self.settings_build` will always contain the settings provided in the profile `profile_build`, even if the recipe appears in the `build` context, the tool requirements of the tool requirements are expected to run in the `build` machine too.
- `self.settings_target`: for recipes in the `host` context this attribute will be equal to `None`, for those in the `build` context, if will depend on the level of validation:
  - for recipes that are tool requirements of packages in the `host` context, this attribute will contain the settings from the profile `profile_host`, while
  - for recipes that are tool requirements of other tool requirements the `self.settings_target` will contain the values of the `profile_build`.

With previous attributes, a draft for a recipe that packages a cross compiler could follow this pattern:

```
class CrossCompiler(ConanFile):
    name = "my_compiler"

    settings = "os", "arch", "compiler", "build_type"
    options = {"target": "ANY"}
    default_options = {"shared": False, "target": None}

    def validate(self):
        settings_target = getattr(self, 'settings_target', None)
        if settings_target is None:
            # It is running in 'host', so Conan is compiling this package
            if not self.options.target:
                raise ConanInvalidConfiguration("A value for option 'target' has to be␣
↪provided")
        else:
            # It is running in 'build' and it is being used as a BR, 'target' can be␣
↪inferred from settings
            if self.options.target:
                raise ConanInvalidConfiguration("Value for the option 'target' will be␣
↪computed from settings_target")
            self.options.target = "<target-value>"  # Use 'self.settings_target' to get␣
↪this value
```

Conan older than v1.24

Warning: We ask you to use the previous approach for Conan 1.24 and newer, and avoid any specific modification of your recipes to make them work as tool requirements in a cross building scenario.

With this approach, only one profile is provided in the command line (the `--profile:host` or just `--profile`) and it has to define the `os_build` and `arch_build` settings too. The recipe of this tool requires has to be modified to take into account these settings and the `compiler` and `build_type` settings have to be removed because their values for the `build` platform are not defined in the profile:

```
from conans import ConanFile
import os
```

(continues on next page)

(continued from previous page)

```
class MyToolchainXXXConan(ConanFile):
    name = "my_toolchain"
    version = "0.1"
    settings = "os_build", "arch_build"

    # As typically, this recipe doesn't declare 'compiler' and 'build_type',
    #   the source() and build() methods need a custom implementation
    def build(self):
        # Typically download the toolchain for the 'build' platform
        url = "http://fake_url.com/installers/%s/%s/toolchain.tgz" % (os_build, os_arch)
        tools.download(url, "toolchain.tgz")
        tools.unzip("toolchain.tgz")

    def package(self):
        # Copy all the required files for your toolchain
        self.copy("*", dst="", src="toolchain")

    def package_info(self):
        bin_folder = os.path.join(self.package_folder, "bin")
        self.env_info.PATH.append(bin_folder)
        self.env_info.CC = os.path.join(bin_folder, "mycompiler-cc")
        self.env_info.CXX = os.path.join(bin_folder, "mycompiler-cxx")
        self.env_info.SYSROOT = self.package_folder
```

With this approach we also need to add the path to the binaries to the `PATH` environment variable. The one and only profile has to include a `[tool_requires]` section with the reference to our new packaged toolchain and it will also contain a `[settings]` section with the regular settings plus the `os_build` and `arch_build` ones.

This approach requires a special profile, and it needs a modified recipe without the `compiler` and `build_type` settings, Conan can still compile it from sources but it won't be able to identify the binary properly and it can be really to tackle if the tool requirements has other Conan dependencies.

Host settings `os_build`, `arch_build`, `os_target` and `arch_target`

Warning: These settings are being reviewed and might be deprecated in the future, we encourage you to try not to use them. If you need help with your use case, please open an issue in the Conan repository and we will help you.

Before Conan v1.24 the recommended way to deal with cross building was to use some extra settings like `os_build`, `arch_build` and `os_target` and `arch_target`. These settings have a special meaning for some Conan tools and build helpers, but they also need to be listed in the recipes themselves creating a dedicated set of recipes for *installers* and *tools* in general. This approach should be superseeded with the introduction in Conan 1.24 of the command line arguments `--profile:host` and `--profile:build` that allow to declare two different profiles with all the information needed for the corresponding platforms (see section above this one).

The meaning of those settings is the following:

- The settings `os_build` and `arch_build` identify the `build` platform according to the GNU convention triplet. These settings are detected the first time you run Conan with the same values than the `host` settings, so by default, we are doing **native building**. You will probably never need to change the value of this setting because they describe where are you running Conan.

- The settings `os_target` and `arch_target` identify the `target` platform. If you are building a cross compiler, these settings specify where the compiled code will run.

The rest of settings, as we already know, identify the `host` platform.

13.1.3 ARM architecture reference

Remember that the Conan settings are intended to unify the different names for operating systems, compilers, architectures etc.

Conan has different architecture settings for ARM: `armv6`, `armv7`, `armv7hf`, `armv8`. The "problem" with ARM architecture is that it's frequently named in different ways, so maybe you are wondering what setting do you need to specify in your case.

Here is a table with some typical ARM platforms:

Platform	Conan setting
Raspberry PI 1	`armv6`
Raspberry PI 2	`armv7` or `armv7hf` if we want to use the float point hard support
Raspberry PI 3	`armv8` also known as armv64-v8a
Visual Studio	`armv7` currently Visual Studio builds `armv7` binaries when you select ARM.
Android armbeabi-v7a	`armv7`
Android armv64-v8a	`armv8`
Android armeabi	`armv6` (as a minimal compatible, will be compatible with v7 too)

13.1.4 Examples

Examples using profiles

Linux to Windows

- Install the needed toolchain, in Ubuntu:

```
sudo apt-get install g++-mingw-w64 gcc-mingw-w64
```

- Create a file named **linux_to_win64** with the contents:

```
toolchain=/usr/x86_64-w64-mingw32 # Adjust this path
target_host=x86_64-w64-mingw32
cc_compiler=gcc
cxx_compiler=g++

[env]
CONAN_CMAKE_FIND_ROOT_PATH=$toolchain  # Optional, for CMake to find things in that␣
↪folder
CONAN_CMAKE_SYSROOT=$toolchain  # Optional, if we want to define sysroot
CHOST=$target_host
AR=$target_host-ar
AS=$target_host-as
RANLIB=$target_host-ranlib
CC=$target_host-$cc_compiler
```

(continues on next page)

(continued from previous page)

```
CXX=$target_host-$cxx_compiler
STRIP=$target_host-strip
RC=$target_host-windres

[settings]
# We are cross-building to Windows
os=Windows
arch=x86_64
compiler=gcc

# Adjust to the gcc version of your MinGW package
compiler.version=7.3
compiler.libcxx=libstdc++11
build_type=Release
```

- Clone an example recipe or use your own recipe:

```
git clone https://github.com/memsharded/conan-hello.git
```

- Call **conan create** using the created **linux_to_win64**

```
$ cd conan-hello && conan create . conan/testing --profile ../linux_to_win64
...
[ 50%] Building CXX object CMakeFiles/example.dir/example.cpp.obj
[100%] Linking CXX executable bin/example.exe
[100%] Built target example
```

A *bin/example.exe* for Win64 platform has been built.

Windows to Raspberry Pi (Linux/ARM)

- Install the toolchain: https://gnutoolchains.com/raspberry/ You can choose different versions of the GCC cross compiler. Choose one and adjust the following settings in the profile accordingly.
- Create a file named **win_to_rpi** with the contents:

```
target_host=arm-linux-gnueabihf
standalone_toolchain=C:/sysgcc/raspberry
cc_compiler=gcc
cxx_compiler=g++

[settings]
os=Linux
arch=armv7 # Change to armv6 if you are using Raspberry 1
compiler=gcc
compiler.version=6
compiler.libcxx=libstdc++11
build_type=Release

[env]
CONAN_CMAKE_FIND_ROOT_PATH=$standalone_toolchain/$target_host
CONAN_CMAKE_SYSROOT=$standalone_toolchain/$target_host/sysroot
```

(continues on next page)

(continued from previous page)

```
PATH=[$standalone_toolchain/bin]
CHOST=$target_host
AR=$target_host-ar
AS=$target_host-as
RANLIB=$target_host-ranlib
LD=$target_host-ld
STRIP=$target_host-strip
CC=$target_host-$cc_compiler
CXX=$target_host-$cxx_compiler
CXXFLAGS=-I"$standalone_toolchain/$target_host/lib/include"
```

The profiles to target Linux are all very similar. You probably just need to adjust the variables declared at the top of the profile:

- **target_host**: All the executables in the toolchain starts with this prefix.
- **standalone_toolchain**: Path to the toolchain installation.
- **cc_compiler/cxx_compiler**: In this case `gcc`/`g++`, but could be `clang`/`clang++`.
- Clone an example recipe or use your own recipe:

```
git clone https://github.com/memsharded/conan-hello.git
```

- Call **`conan create`** using the created profile.

```
$ cd conan-hello && conan create . conan/testing --profile=../win_to_rpi
...
[ 50%] Building CXX object CMakeFiles/example.dir/example.cpp.obj
[100%] Linking CXX executable bin/example
[100%] Built target example
```

A *bin/example* for Raspberry PI (Linux/armv7hf) platform has been built.

Windows to Windows CE

The Windows CE (WinCE) operating system is supported for CMake and MSBuild. Since WinCE depends on the MSVC compiler, Visual Studio and the according Windows CE platform SDK for the WinCE device have to be installed on the build host.

The `os.platform` defines the WinCE Platform SDK and is equal to the `Platform` in Visual Studio.

Some examples for Windows CE platforms:

- `SDK_AM335X_SK_WEC2013_V310`
- `STANDARDSDK_500 (ARMV4I)`
- `Windows Mobile 5.0 Pocket PC SDK (ARMV4I)`
- `Toradex_CE800 (ARMV7)`

The `os.version` defines the WinCE version and must be "`5.0`", "`6.0`" or "`7.0`".

CMake supports Visual Studio 2008 (`compiler.version=9`) and Visual Studio 2012 (`compiler.version=11`).

Example of an Windows CE conan profile:

```
[settings]
os=WindowsCE
os.version=8.0
os.platform=Toradex_CE800 (ARMV7)
arch=armv7
compiler=Visual Studio
compiler.version=11

# Release configuration
build_type=Release
compiler.runtime=MD
```

Note: Further information about CMake and WinCE can be found in the CMake documentation:

CMake - Cross Compiling for Windows CE

Linux/Windows/macOS to Android

Cross-building a library for Android is very similar to the previous examples, except the complexity of managing different architectures (armeabi, armeabi-v7a, x86, arm64-v8a) and the Android API levels.

Download the Android NDK here and unzip it.

Note: If you are in Windows the process will be almost the same, but unzip the file in the root folder of your hard disk (`C:\`) to avoid issues with path lengths.

Note: If you are using Android Studio, you may use already available Android NDK

To use the `clang` compiler, create a profile `android_21_arm_clang`. Once again, the profile is very similar to the RPI one:

```
include(default)
target_host=aarch64-linux-android
android_ndk=/Users/sse4/Library/Android/sdk/ndk-bundle  # Adjust this path
api_level=21
[settings]
arch=armv8
build_type=Release
compiler=clang
compiler.libcxx=libc++
compiler.version=9
os=Android
os.api_level=$api_level
[tool_requires]
[options]
[env]
PATH=[$android_ndk/toolchains/llvm/prebuilt/darwin-x86_64/bin]  # Adjust this path
CHOST=$target_host
```

(continues on next page)

(continued from previous page)

```
AR=$target_host-ar
AS=$target_host-as
RANLIB=$target_host-ranlib
CC=$target_host$api_level-clang
CXX=$target_host$api_level-clang++
LD=$target_host-ld
STRIP=$target_host-strip
CONAN_CMAKE_TOOLCHAIN_FILE=$android_ndk/build/cmake/android.toolchain.cmake
```

- Clone, for example, the zlib library to try to build it to Android

```
git clone https://github.com/conan-io/conan-center-index.git
```

- Call **conan create** using the created profile.

```
$ cd conan-center-index/recipes/zlib/1.2.11 && conan create . 1.2.11@ -pr:h ../android_
↪21_arm_clang -pr:b default

...
-- Build files have been written to: /tmp/conan-zlib/test_package/build/
↪ba0b9dbae0576b9a23ce7005180b00e4fdef1198
Scanning dependencies of target enough
[ 50%] Building C object CMakeFiles/enough.dir/enough.c.o
[100%] Linking C executable bin/enough
[100%] Built target enough
zlib/1.2.11 (test package): Running test()
```

A **bin/enough** for Android ARM platform has been built.

Examples using tool requires

You can find one example on how to use tool requires for cross-compiling to iOS in the *iOS integration section* in the documentation.

See also:

- Check the *Creating conan packages to install dev tools* to learn more about how to create Conan packages for tools.
- Check the msys2 tool require recipe as an example of packaging a compiler.

—

See also:

Reference links

ARM

- https://developer.arm.com/documentation/dui0773/j/compiling-c-and-c—code/specifying-a-target-architecture–processor–and-instruction-set
- https://developer.arm.com/documentation/dui0472/latest/compiler-command-line-options

ANDROID

- https://developer.android.com/ndk/guides/standalone_toolchain

VISUAL STUDIO

- https://docs.microsoft.com/en-us/visualstudio/msbuild/msbuild-command-line-reference?view=vs-2017

See also:

- See *conan.conf file* and *Environment variables* sections to know more.
- See *AutoToolsBuildEnvironment build helper* reference.
- See *CMake build helper* reference.
- See CMake cross-building wiki to know more about cross-building with CMake.

13.2 Windows Subsystems

On Windows, you can run different *subsystems* that enhance the operating system with UNIX capabilities.

Conan supports `MSYS2`, `CYGWIN`, `WSL` and in general any subsystem that is able to run a `bash` shell.

Many libraries use these subsystems in order to use the Unix tools like the `Autoconf` suite that generates `Makefiles`.

The difference between MSYS2 and CYGWIN is that MSYS2 is oriented to the development of native Windows packages, while CYGWIN tries to provide a complete POSIX-like system to run any Unix application on it.

For that reason, we recommend the use of MSYS2 as a subsystem to be used with Conan.

13.2.1 Operation Modes

The `MSYS2` and `CYGWIN` can be used with different operation modes:

- You can use them together with MinGW to build Windows-native software.
- You can use them together with any other compiler to build Windows-native software, even with Visual Studio.
- You can use them with MinGW to build specific software for the subsystem, with a dependency to a runtime DLL (`msys-2.0.dll` and `cygwin1.dll`)

If you are building specific software for the subsystem, you have to specify a value for the setting `os.subsystem`, if you are only using the subsystem for taking benefit of the UNIX tools but generating native Windows software, you shouldn't specify it.

13.2.2 Running commands inside the subsystem

self.win_bash

This is an experimental feature introduced in Conan 1.39. It supersedes the `run(..., win_bash=True)` argument but if the `run(..., win_bash=True)` is used, it will have priority so the compatibility with the previous behavior is guaranteed.

The `self.win_bash` is an attribute of the conanfile, when set to `True` and only when running in Windows (you don't need to check if you are in Windows), it will run the `self.run()` commands inside a bash shell.

Note: The `bash.exe` that will run **is not auto-detected** or read from the `CONAN_BASH_PATH` anymore, neither the subsystem to be used. These are the config variables used:

- `tools.microsoft.bash:subsystem`: Values can be `msys2`, `cygwin`, `msys` and `wsl`.

- `tools.microsoft.bash:path`: Path to the `bash.exe`

The new *Autotools, AutotoolsToolchain, AutotoolsDeps and PkgConfigDeps* will work automatically when `self.win_bash` is set.

self.run()

In a Conan recipe, you can use the `self.run` method specifying the parameter `win_bash=True` that will call automatically to the tool *tools.run_in_windows_bash*.

It will use the **bash** in the path or the **bash** specified for the environment variable *CONAN_BASH_PATH* to run the specified command.

Conan will automatically escape the command to match the detected subsystem. If you also specify the `msys_mingw` parameter to False, and the subsystem is `MSYS2` it will run in Windows-native mode, the compiler won't link against the `msys-2.0.dll`.

AutoToolsBuildEnvironment

Note: From Conan 1.39 the new *Autotools* build helper will use the `self.win_bash` conanfile attribute (see above) to adjust automatically all the paths to the subsystem.

In the constructor of the build helper, you have the `win_bash` parameter. Set it to `True` to run the `configure` and `make` commands inside a bash.

13.2.3 Controlling the build environment

Building software in a Windows subsystem for a different compiler than MinGW can sometimes be painful. The reason is how the subsystem finds your compiler/tools in your system.

For example, the icu library requires Visual Studio to be built in Windows, but also a subsystem able to build the Makefile. A very common problem and example of the pain is the `link.exe` program. In the Visual Studio suite, `link.exe` is the linker, but in the `MSYS2` environment the `link.exe` is a tool to manage symbolic links.

Conan is able to prioritize the tools when you use `tool_requires`, and put the tools in the PATH in the right order.

There are some packages you can use as `tool_requires`:

- From ConanCenter:
  - **mingw-w64/8.1**: MinGW compiler installer as a Conan package.
  - **msys2/20190524@**: MSYS2 subsystem as a Conan package (Conan Center Index).
  - **cygwin_installer/2.9.0@bincrafters/stable**: Cygwin subsystem as a Conan package.

For example, create a profile and name it *msys2_mingw* with the following contents:

```
[tool_requires]
mingw_installer/1.0@conan/stable
msys2/20190524

[settings]
os_build=Windows
```

(continues on next page)

(continued from previous page)

```
os=Windows
arch=x86_64
arch_build=x86_64
compiler=gcc
compiler.version=4.9
compiler.exception=seh
compiler.libcxx=libstdc++11
compiler.threads=posix
build_type=Release
```

Then you can have a *conanfile.py* that can use `self.run()` with `win_bash=True` to run any command in a bash terminal or use the `AutoToolsBuildEnvironment` to invoke `configure/make` in the `subsystem`:

```
from conans import ConanFile
import os


class MyToolchainXXXConan(ConanFile):
    name = "mylib"
    version = "0.1"
    ...

    def build(self):
        self.run("some_command", win_bash=True)

        env_build = AutoToolsBuildEnvironment(self, win_bash=True)
        env_build.configure()
        env_build.make()

     ...
```

Apply the profile in your recipe to create a package using the MSYS2 and MINGW:

```
$ conan create . user/testing --profile msys2_mingw
```

As we included in the profile the `MinGW` and then the `MSYS2` build_require, when we run a command, the PATH will contain first the MinGW tools and finally the MSYS2.

What could we do with the Visual Studio issue with `link.exe`? You can pass an additional parameter to `run_in_windows_bash` with a dictionary of environment variables to have more priority than the others:

```
def build(self):
    # ...
    vs_path = tools.vcvars_dict(self)["PATH"] # Extract the path from the vcvars_dict␣
↪tool
    tools.run_in_windows_bash(self, command, env={"PATH": vs_path})
```

So you will get first the `link.exe` from the Visual Studio.

Also, Conan has a tool `tools.remove_from_path` that you can use in a recipe to temporarily remove a tool from the path if you know that it can interfere with your build script:

```
class MyToolchainXXXConan(ConanFile):
    name = "mylib"
```

(continues on next page)

(continued from previous page)

```
    version = "0.1"
    ...

    def build(self):
        with tools.remove_from_path("link"):
            # Call something
            self.run("some_command", win_bash=True)

     ...
```

CHAPTER

FOURTEEN

EXTENDING CONAN

This section provides an introduction to extension capabilities of Conan:

14.1 Customizing settings

There is a file in *<userhome>/.conan/settings.yml* that contains a default definition of the allowed `settings` values for Conan package recipes. It looks like:

```
os:
    Windows:
        subsystem: [None, cygwin, msys, msys2, wsl]
    Linux:
    Macos:
        version: [None, "10.6", "10.7", "10.8", "10.9", "10.10", "10.11", "10.12", "10.13
↪", "10.14"]
    Android:
        api_level: ANY
    iOS:
        version: ["7.0", "7.1", "8.0", "8.1", "8.2", "8.3", "9.0", "9.1", "9.2", "9.3",
↪"10.0", "10.1", "10.2", "10.3", "11.0", "11.1", "11.2", "11.3", "11.4", "12.0", "12.1"]
    watchOS:
        version: ["4.0", "4.1", "4.2", "4.3", "5.0", "5.1"]
    FreeBSD:
    SunOS:
    Emscripten:
arch: [x86, x86_64, ppc32, ppc64le, ppc64, armv4, armv4i, armv5el, armv5hf, armv6, armv7,
↪ armv7hf, armv7s, armv7k, armv8, armv8_32, armv8.3, sparc, sparcv9, mips, mips64, avr,␣
↪s390, s390x, asm.js, wasm]
compiler:
    gcc:
        version: ["4.1", "4.4", "4.5", "4.6", "4.7", "4.8", "4.9",
                "5", "5.1", "5.2", "5.3", "5.4", "5.5",
                "6", "6.1", "6.2", "6.3", "6.4",
                "7", "7.1", "7.2", "7.3",
                "8", "8.1", "8.2",
                "9"]
        libcxx: [libstdc++, libstdc++11]
        threads: [None, posix, win32] #  Windows MinGW
        exception: [None, dwarf2, sjlj, seh] # Windows MinGW
        cppstd: [None, 98, gnu98, 11, gnu11, 14, gnu14, 17, gnu17, 20, gnu20]
```

(continues on next page)

(continued from previous page)

```
    Visual Studio:
        runtime: [MD, MT, MTd, MDd]
        version: ["8", "9", "10", "11", "12", "14", "15", "16"]
        toolset: [None, v90, v100, v110, v110_xp, v120, v120_xp,
                v140, v140_xp, v140_clang_c2, LLVM-vs2012, LLVM-vs2012_xp,
                LLVM-vs2013, LLVM-vs2013_xp, LLVM-vs2014, LLVM-vs2014_xp,
                LLVM-vs2017, LLVM-vs2017_xp, v141, v141_xp, v141_clang_c2, v142,
                llvm, ClangCL]
        cppstd: [None, 14, 17, 20]
```

This are the **default** settings and values. They are a common syntax and notation for having package binary IDs that are common to all developers. They are also used for validation, for example if you write in a profile `[settings]` something like `os=Windos` (note the typo), then it will raise an error, telling you it is not a recognized `os` and offering a list of available `os`. Also, note how the sub-settings are different for different platforms, for example the standard C++ library (`compiler.libcxx`) exists for the `gcc` compiler, but not for `Visual Studio` compiler. And in the same way, `Visual Studio` has a `runtime` sub-setting that is missing in `gcc`. Trying to incorrectly use or define these sub-settings in the wrong compiler will also raise an error.

These settings are good for defining a base for Open Source packages, and for a large number of mainstream configurations. But it is likely that you might need finer detail of definition of the binaries that are being created.

For example, it is possible that you are managing binaries for older Linux distros, like RHEL 6, or old Centos, besides other modern distributions. The problem is that the binaries compiled for modern distributions will not work (will not be binary compatible, or ABI incompatible) in those older distributions, mainly because of different versions of glibc. We would need a way to model the differences of the binaries for those platforms. Check out the section *Deployment challenges* which explains mentioned situation in detail.

14.1.1 Adding new settings

It is possible to add new settings at the root of the *settings.yml* file, something like:

```
os:
    Windows:
        subsystem: [None, cygwin, msys, msys2, wsl]
distro: [None, RHEL6, CentOS, Debian]
```

If we want to create different binaries from our recipes defining this new setting, we would need to add to our recipes that:

```
class Pkg(ConanFile):
    settings = "os", "compiler", "build_type", "arch", "distro"
```

The value `None` allows for not defining it (which would be a default value, valid for all other distros). It is possible to define values for it in the profiles:

```
[settings]
os = "Linux"
distro = "CentOS"
compiler = "gcc"
```

And use their values to affect our build if desired:

```
class Pkg(ConanFile):
    settings = "os", "compiler", "build_type", "arch", "distro"

    def build(self):
        cmake = CMake(self)
        if self.settings.distro == "CentOS":
            cmake.definitions["SOME_CENTOS_FLAG"] = "Some CentOS Value"
            ...
```

14.1.2 Adding new sub-settings

The above approach requires modification to all recipes to take it into account. It is also possible to define kind of incompatible settings, like `os=Windows` and `distro=CentOS`. While adding new settings is totally possible, it might make more sense for other cases, but for this example it is more adequate to add it as above subsetting of the `Linux` OS:

```
os:
    Windows:
        subsystem: [None, cygwin, msys, msys2, wsl]
    Linux:
        distro: [None, RHEL6, CentOS, Debian]
```

With this definition we could define our profiles as:

```
[settings]
os = "Linux"
os.distro = "CentOS"
compiler = "gcc"
```

And any attempt to define `os.distro` for another `os` value rather than `Linux` will raise an error.

As this is a subsetting, it will be automatically taken into account in all recipes that declare an `os` setting. Note that having a value of `distro=None` possible is important if you want to keep previously created binaries, otherwise you would be forcing to always define a specific distro value, and binaries created without this subsetting, won't be usable anymore.

The sub-setting can also be accessed from recipes:

```
class Pkg(ConanFile):
    settings = "os", "compiler", "build_type", "arch"  # Note, no "distro" defined here

    def build(self):
        cmake = CMake(self)
        if self.settings.os == "Linux" and self.settings.os.distro == "CentOS":
            cmake.definitions["SOME_CENTOS_FLAG"] = "Some CentOS Value"
```

14.1.3 Add new values

In the same way we have added a new `distro` subsetting, it is possible to add new values to existing settings and subsettings. For example, if some compiler version is not present in the range of accepted values, you can add those new values.

You can also add a completely new compiler:

```
os:
    Windows:
        subsystem: [None, cygwin, msys, msys2, wsl]
   ...
compiler:
    gcc:
        ...
    mycompiler:
        version: [1.1, 1.2]
    Visual Studio:
```

This works as the above regarding profiles, and the way they can be accessed from recipes. The main issue with custom compilers is that the builtin build helpers, like `CMake`, `MSBuild`, etc, internally contains code that will check for those values. For example, the `MSBuild` build helper will only know how to manage the `Visual Studio` setting and sub-settings, but not the new compiler. For those cases, custom logic can be implemented in the recipes:

```
class Pkg(ConanFile):
    settings = "os", "compiler", "build_type", "arch"

    def build(self):
        if self.settings.compiler == "mycompiler":
            my_custom_compile = ["some", "--flags", "for", "--my=compiler"]
            self.run(["mycompiler", "."] + my_custom_compile)
```

Note: You can also remove items from *settings.yml* file. You can remove compilers, OS, architectures, etc. Do that only in the case you really want to protect against creation of binaries for other platforms other than your main supported ones. In the general case, you can leave them, the binary configurations are managed in **profiles**, and you want to define your supported configurations in profiles, not by restricting the *settings.yml*

Note: If you customize your *settings.yml*, you can share, distribute and sync this configuration with your team and CI machines with the *conan config install* command.

14.2 Python requires

Warning: This is an **experimental** feature subject to breaking changes in future releases.

Note: This syntax supersedes the *legacy python_requires()* syntax. The most important changes are:

- These new python_requires affect the consumers `package_id`. So different binaries can be managed, and CI systems can re-build affected packages according to package ID modes and versioning policies.

- The syntax defines a *class attribute* instead of a module function call, so recipes are cleaner and more aligned with other types of requirements.
- The new python_requires will play better with lockfiles and deterministic dependency graphs.
- They are able to extend base classes more naturally without conflicts of ConanFile classes.

14.2.1 Introduction

The `python_requires` feature is a very convenient way to share files and code between different recipes. A python requires is similar to any other recipe, it is the way it is required from the consumer what makes the difference.

A very simple recipe that we want to reuse could be:

```
from conans import ConanFile

myvar = 123

def myfunct():
    return 234

class Pkg(ConanFile):
    pass
```

And then we will make it available to other packages with `conan export`. Note that we are not calling `conan create`, because this recipe doesn't have binaries. It is just the python code that we want to reuse.

```
$ conan export . pyreq/0.1@user/channel
```

We can reuse the above recipe functionality declaring the dependency in the `python_requires` attribute and we can access its members using `self.python_requires["<name>"].module`:

```
from conans import ConanFile

class Pkg(ConanFile):
    python_requires = "pyreq/0.1@user/channel"

    def build(self):
        v = self.python_requires["pyreq"].module.myvar  # v will be 123
        f = self.python_requires["pyreq"].module.myfunct()  # f will be 234
        self.output.info("%s, %s" % (v, f))
```

```
$ conan create . pkg/0.1@user/channel
...
pkg/0.1@user/channel: 123, 234
```

It is also possible to require more than one python-require, and use the package name to address the functionality:

```
from conans import ConanFile

class Pkg(ConanFile):
    python_requires = "pyreq/0.1@user/channel", "other/1.2@user/channel"
```

(continues on next page)

(continued from previous page)

```
    def build(self):
        v = self.python_requires["pyreq"].module.myvar  # v will be 123
        f = self.python_requires["other"].module.otherfunc("some-args")
```

14.2.2 Extending base classes

A common use case would be to declare a base class with methods we want to reuse in several recipes via inheritance. We'd write this base class in a python-requires package:

```
from conans import ConanFile

class MyBase(object):
    def source(self):
        self.output.info("My cool source!")
    def build(self):
        self.output.info("My cool build!")
    def package(self):
        self.output.info("My cool package!")
    def package_info(self):
        self.output.info("My cool package_info!")

class PyReq(ConanFile):
    name = "pyreq"
    version = "0.1"
```

And make it available for reuse with:

```
$ conan export . pyreq/0.1@user/channel
```

Note that there are two classes in the recipe file:

- `MyBase` is the one intended for inheritance and doesn't extend `ConanFile`.
- `PyReq` is the one that defines the current package being exported, it is the recipe for the reference `pyreq/0.1@user/channel`.

Once the package with the base class we want to reuse is available we can use it in other recipes to inherit the functionality from that base class. We'd need to declare the `python_requires` as we did before and we'd need to tell Conan the base classes to use in the attribute `python_requires_extend`. Here our recipe will inherit from the class `MyBase`:

```
from conans import ConanFile

class Pkg(ConanFile):
    python_requires = "pyreq/0.1@user/channel"
    python_requires_extend = "pyreq.MyBase"
```

The resulting inheritance is equivalent to declare our `Pkg` class as `class Pkg(pyreq.MyBase, ConanFile)`. So creating the package we can see how the methods from the base class are reused:

```
$ conan create . pkg/0.1@user/channel
...
pkg/0.1@user/channel: My cool source!
pkg/0.1@user/channel: My cool build!
```

(continues on next page)

(continued from previous page)

```
pkg/0.1@user/channel: My cool package!
pkg/0.1@user/channel: My cool package_info!
...
```

If there is extra logic needed to extend from a base class, like composing the base class settings with the current recipe, the `init()` method can be used for it:

```
class PkgTest(ConanFile):
    license = "MIT"
    settings = "arch", # tuple!
    python_requires = "base/1.1@user/testing"
    python_requires_extend = "base.MyConanfileBase"

    def init(self):
        base = self.python_requires["base"].module.MyConanfileBase
        self.settings = base.settings + self.settings  # Note, adding 2 tuples = tuple
        self.license = base.license  # License is overwritten
```

For more information about the `init()` method visit *init()*

Limitations

There are a few limitations that should be taken into account:

- `name` and `version` fields shouldn't be inherited. `set_name()` and `set_version()` might be used.
- `short_paths` cannot be inherited from a `python_requires`. Make sure to specify it directly in the recipes that need the paths shortened in Windows.
- `exports`, `exports_sources` shouldn't be inherited from a base class, but explictly defined directly in the recipes. A reusable alternative might be using the `SCM` component.
- `build_policy` shouldn't be inherited from a base class, but explictly defined directly in the recipes.
- Mixing Python inheritance with `python_requires_extend` should be avoided, because the inheritance order can be different than the expected one. Multiple level `python_requires_extend` might be possible, but don't mix both approaches (also in general try to avoid multiple inheritance and multiple level hierarchies, try to keep it simple).

14.2.3 Reusing files

It is possible to access the files exported by a recipe that is used with `python_requires`. We could have this recipe, together with a *myfile.txt* file containing the "Hello" text.

```
from conans import ConanFile

class PyReq(ConanFile):
    exports = "*"
```

```
$ echo "Hello" > myfile.txt
$ conan export . pyreq/0.1@user/channel
```

Now the recipe has been exported, we can access its path (the place where *myfile.txt* is) with the `path` attribute:

```
import os
from conans import ConanFile, load

class Pkg(ConanFile):
    python_requires = "pyreq/0.1@user/channel"

    def build(self):
        pyreq_path = self.python_requires["pyreq"].path
        myfile_path = os.path.join(pyreq_path, "myfile.txt")
        content = load(myfile_path)  # content = "Hello"
        self.output.info(content)
        # we could also copy the file, instead of reading it
```

Note that only `exports` work for this case, but not `exports_sources`.

14.2.4 PackageID

The `python-requires` will affect the `package_id` of the packages using those dependencies. By default, the policy is `minor_mode`, which means:

- Changes to the **patch** version of a python-require will not affect the package ID. So depending on `"pyreq/1.2.3"` or `"pyreq/1.2.4"` will result in identical package ID (both will be mapped to `"pyreq/1.2.Z"` in the hash computation). Bump the patch version if you want to change your common code, but you don't want the consumers to be affected or to fire a re-build of the dependants.
- Changes to the **minor** or **major** version will produce a different package ID. So if you depend on `"pyreq/1.2.3"`, and you bump the version to `"pyreq/1.3.0"`, then, you will need to build new binaries that are using that new python-require. Bump the minor or major version if you want to make sure that packages requiring this python-require will be built using these changes in the code.
- Both changing the **minor** and **major** requires a new package ID, and then a build from source. You could use changes in the **minor** to indicate that it should be source compatible, and consumers wouldn't need to do changes, and changes in the **major** for source incompatible changes.

As with the regular `requires`, this default can be customized. First you can customize it at attribute global level, modifying the *conan.conf* `[general]` variable `default_python_requires_id_mode`, which can take the values `unrelated_mode`, `semver_mode`, `patch_mode`, `minor_mode`, `major_mode`, `full_version_mode`, `full_recipe_mode` and `recipe_revision_mode`.

For example, if you want to make the package IDs never be affected by any change in the versions of python-requires, you could do:

Listing 1: *conan.conf* configuration file

```
[general]
default_python_requires_id_mode=unrelated_mode
```

Read more about these modes in *Using package_id() for Package Dependencies*.

It is also possible to customize the effect of `python_requires` per package, using the `package_id()` method:

```
from conans import ConanFile

class Pkg(ConanFile):
    python_requires ="pyreq/[>=1.0]"
```

(continues on next page)

(continued from previous page)

```
    def package_id(self):
        self.info.python_requires.patch_mode()
```

14.2.5 Resolution of python-requires

There are few things that should be taken into account when using `python-requires`:

- Python requires recipes are loaded by the interpreter just once, and they are common to all consumers. Do not use any global state in the `python-requires` recipes.
- Python requires are private to the consumers. They are not transitive. Different consumers can require different versions of the same python-require.
- `python-requires` can use version ranges expressions.
- `python-requires` can `python-require` other recipes too, but this should probably be limited to very few cases, we recommend to use the simplest possible structure.
- `python-requires` can conflict if they require other recipes and create conflicts in different versions.
- `python-requires` cannot use regular `requires` or `tool_requires`.
- It is possible to use `python-requires` without user and channel.
- `python-requires` can use native python `import` to other python files, as long as these are exported together with the recipe.
- `python-requires` should not create packages, but use `export` only.
- `python-requires` can be used as editable packages too.
- `python-requires` are locked in lockfiles.

14.3 Python requires (legacy)

Warning: This feature has been superseded by the new *Python requires*. Even if this is an **experimental** feature subject to breaking changes in future releases, this legacy `python_requires` syntax has not been removed yet, but it will be removed in Conan 2.0.

The `python_requires()` feature is a very convenient way to share files and code between different recipes. A *Python Requires* is just like any other recipe, it is the way it is required from the consumer what makes the difference.

The *Python Requires* recipe file, besides exporting its own required sources, can export files to be used by the consumer recipes and also python code in the recipe file itself.

Let's have a look at an example showing all its capabilities (you can find all the sources in Conan examples repository):

- Python requires recipe:

```
import os
import shutil
from conans import ConanFile, CMake, tools
from scm_utils import get_version
```

(continues on next page)

(continued from previous page)

```
class PythonRequires(ConanFile):
    name = "pyreq"
    version = "version"

    exports = "scm_utils.py"
    exports_sources = "CMakeLists.txt"


def get_conanfile():

    class BaseConanFile(ConanFile):

        settings = "os", "compiler", "build_type", "arch"
        options = {"shared": [True, False]}
        default_options = {"shared": False}
        generators = "cmake"
        exports_sources = "src/*"

        def source(self):
            # Copy the CMakeLists.txt file exported with the python requires
            pyreq = self.python_requires["pyreq"]
            shutil.copy(src=os.path.join(pyreq.exports_sources_folder,
↪"CMakeLists.txt"),
                        dst=self.source_folder)

            # Rename the project to match the consumer name
            tools.replace_in_file(os.path.join(self.source_folder,
↪"CMakeLists.txt"),
                                  "add_library(mylibrary ${sources})",
                                  "add_library({} ${{sources}})".
↪format(self.name))


        def build(self):
            cmake = CMake(self)
            cmake.configure()
            cmake.build()

        def package(self):
            self.copy("*.h", dst="include", src="src")
            self.copy("*.lib", dst="lib", keep_path=False)
            self.copy("*.dll", dst="bin", keep_path=False)
            self.copy("*.dylib*", dst="lib", keep_path=False)
            self.copy("*.so", dst="lib", keep_path=False)
            self.copy("*.a", dst="lib", keep_path=False)

        def package_info(self):
            self.cpp_info.libs = [self.name]

    return BaseConanFile
```

- Consumer recipe

```
from conans import ConanFile, python_requires


base = python_requires("pyreq/version@user/channel")

class ConsumerConan(base.get_conanfile()):
    name = "consumer"
    version = base.get_version()

    # Everything else is inherited
```

We must make available for other to use the recipe with the *Python Requires*, this recipe won't have any associated binaries, only the sources will be needed, so we only need to execute the export and upload commands:

```
$ conan export . pyreq/version@user/channel
$ conan upload pyreq/version@user/channel -r=myremote
```

Now any consumer will be able to reuse the business logic and files available in the recipe, let's have a look at the most common use cases.

14.3.1 Import a python requires

To import a recipe as a *Python requires* it is needed to call the `python_requires()` function with the reference as the only parameter:

```
base = python_requires("pyreq/version@user/channel")
```

All the code available in the *conanfile.py* file of the imported recipe will be available in the consumer through the `base` variable.

Important: There are **several important considerations** regarding `python_requires()`:

- They are required at every step of the conan commands. If you are creating a package that `python_requires("MyBase/...")`, the `MyBase` package should be already available in the local cache or to be downloaded from the remotes. Otherwise, conan will raise a "missing package" error.
- They do not affect the package binary ID (hash). Depending on different version, or different channel of such `python_requires()` do not change the package IDs as the normal dependencies do.
- They are imported only once. The python code that is reused is imported only once, the first time it is required. Subsequent requirements of that conan recipe will reuse the previously imported module. Global initialization at parsing time and global state are discouraged.
- They are transitive. One recipe using `python_requires()` can be also consumed with a `python_requires()` from another package recipe.
- They are not automatically updated with the `--update` argument from remotes.
- Different packages can require different versions in their `python_requires()`. They are private to each recipe, so they do not conflict with each other, but it is the responsibility of the user to keep consistency.
- They are not overridden from downstream consumers. Again, as they are private, they are not affected by other packages, even consumers

---

14.3.2 Reuse python sources

In the example proposed we are using two functions through the `base` variable: `base.get_conanfile()` and `base.get_version()`. The first one is defined directly in the *conanfile.py* file, but the second one is in a different source file that was exported together with the `pyreq/version@user/channel` recipe using the `exports` attribute.

This works without any Conan magic, it is just plain Python and you can even return a class from a function and inherit from it. That's just what we are proposing in this example: all the business logic in contained in the *Python Requires* so every recipe will reuse it automatically. The consumer only needs to define the `name` and `version`:

```
from conans import ConanFile, python_requires


base = python_requires("pyreq/version@user/channel")

class ConsumerConan(base.get_conanfile()):
    name = "consumer"
    version = "version"

    # Everything else is inherited
```

while all the functional code is defined in the *python requires* recipe file:

```
from conans import ConanFile, python_requires

[...]

def get_conanfile():

    class BaseConanFile(ConanFile):
        def source(self):
            [...]

        def build(self):
            [...]
```

14.3.3 Reuse source files

Up to now, we have been reusing python code, but we can also package files within the *python requires* recipe and consume them afterward, that's what we are doing with a *CMakeList.txt* file, it will allow us to share the CMake code and ensure that all the libraries using the same *python requires* will have the same build script. These are the relevant code snippets from the example files:

- The *python requires* exports the needed sources (the file exists next to this *conanfile.py*):

```
class PythonRequires(ConanFile):
    name = "pyreq"
    version = "version"

    exports_sources = "CMakeLists.txt"

    [...]
```

The file will be exported together with the recipe `pyreq/version@user/channel` during the call to `conan export . pyreq/version@user/channel` as it is expected for any Conan package.

- The consumer recipe will copy the file from the *python requires* folder, we need to make this copy ourselves, there is nothing run automatically during the `python_requires()` call:

```
class BaseConanFile(ConanFile):
    [...]

    def source(self):
        # Copy the CMakeLists.txt file exported with the python requires
        pyreq = self.python_requires["pyreq"]
        shutil.copy(src=os.path.join(pyreq.exports_sources_folder,
↪"CMakeLists.txt"),
                    dst=self.source_folder)

        # Rename the project to match the consumer name
        tools.replace_in_file(os.path.join(self.source_folder, "CMakeLists.
↪txt"),
                              "add_library(mylibrary ${sources})",
                              "add_library({} ${{sources}})".format(self.
↪name))
```

As you can see, in the inherited `source()` method, we are copying the *CMakeLists.txt* file from the *exports_sources* folder of the python requires (take a look at the *python_requires attribute*), and modifying a line to name the library with the current recipe name.

In the example, our `ConsumerConan` class will also inherit the `build()`, `package()` and `package_info()` method, turning the actual *conanfile.py* of the library into a mere declaration of the name and version.

You can find the full example in the Conan examples repository.

14.4 Creating a custom build helper for Conan

If Conan doesn't have a build helper for the build tool you are using, you can create a custom build helper with the *Python requires*. You can create a package defining the build helper for that build tool and reuse it later in the consumers importing the build helper as a *Python requires*.

As you probably know, build helpers are wrappers of the build tool that help with the conversion of the Conan settings to the build tool's ones. They assist users with the compilation of libraries and applications in the *build()* method of a recipe.

As an example, we are going to create a minimal implementation of a build helper for the Waf build system . First, we need to create a recipe for the `python_requires` that will export *waf_environment.py*, where all the implementation of the build helper is.

```
from conans import ConanFile
from waf_environment import WafBuildEnvironment


class PythonRequires(ConanFile):
    name = "waf-build-helper"
    version = "0.1"
    exports = "waf_environment.py"
```

As we said, the build helper is responsible for translating Conan settings to something that the build tool understands. That can be passing arguments through the command line when invoking the tool or creating files that will take as an input. In this case, the build helper for *Waf* will create one file named *waf_toolchain.py* that will contain linker and compiler flags based on the Conan settings.

To pass that information to *Waf* in the file, you have to modify its configuration environment through the `conf.env` variable setting all the relevant flags. We will also define a `configure` and a `build` method. Let's see how the most important parts of *waf_environment.py* file that defines the build helper could look. In this case, for simplification, the build helper will only add flags depending on the conan setting value for the `build_type`.

```
class WafBuildEnvironment(object):
    def __init__(self, conanfile):
        self._conanfile = conanfile
        self._settings = self._conanfile.settings

    def build_type_flags(self, settings):
        if "Visual Studio" in self._compiler:
            if self._build_type == "Debug":
                return ['/Zi', '/FS']
            elif self._build_type == "Release":
                return ['/O2']
        else:
            if self._build_type == "Debug":
                return ['-g']
            elif self._build_type == "Release":
                return ['-O3']

    def _toolchain_content(self):
        sections = []
        sections.append("def configure(conf):")
        sections.append("    conf.env.CXXFLAGS = conf.env.CXXFLAGS or []")
        _build_type_flags = build_type_flags(self._settings)
        sections.append("    conf.env.CXXFLAGS.extend({})".format(_build_type_flags))
        return "\n".join(sections)

    def _save_toolchain_file(self):
        filename = "waf_conan_toolchain.py"
        content = self._toolchain_content()
        output_path = self._conanfile.build_folder
        save(os.path.join(output_path, filename), content)

    def configure(self, args=None):
        self._save_toolchain_file()
        args = args or []
        command = "waf configure " + " ".join(arg for arg in args)
        self._conanfile.run(command)

    def build(self, args=None):
        args = args or []
        command = "waf build " + " ".join(arg for arg in args)
        self._conanfile.run(command)
```

Now you can export your custom build helper to the local cache, or upload to a remote:

```
$ conan export .
```

After exporting this package to the local cache you can use this custom build helper to compile our packages using the *Waf* build system. Just add the necessary configuration files for *Waf* and import the `python_requires`. The *conanfile.py* of that package could look similar to this:

```
from conans import ConanFile


class TestWafConan(ConanFile):
    python_requires = "waf-build-helper/0.1"
    settings = "os", "compiler", "build_type", "arch"
    name = "waf-consumer"
    generators = "Waf"
    requires = "mylib-waf/1.0"
    tool_requires = "WafGen/0.1", "waf/2.0.19"
    exports_sources = "wscript", "main.cpp"

    def build(self):
        waf = self.python_requires["waf-build-helper"].module.WafBuildEnvironment(self)
        waf.configure()
        waf.build()
```

As you can see in the *conanfile.py* we also are requiring the build tool and a generator for that build tool. If you want more detailed information on how to integrate your own build system in Conan, please check this blog-post about that topic.

14.5 Hooks

Warning: This is an **experimental** feature subject to breaking changes in future releases.

The Conan hooks is a feature intended to extend the Conan functionalities and let users customize the client behavior at determined points.

14.5.1 Hook structure

A hook is a Python function that will be executed at certain points of Conan workflow to customize the client behavior without modifying the client sources or the recipe ones. In the *hooks reference* you can find the full list of hook functions and exhaustive documentation about their arguments.

Hooks can implement any functionality: it could be Conan commands, recipe interactions such as exporting or packaging, or interactions with the remotes.

Here is an example of a simple hook:

Listing 2: *example_hook.py*

```
from conans import tools
```

(continues on next page)

(continued from previous page)

```
def pre_export(output, conanfile, conanfile_path, reference, **kwargs):
    test = "%s/%s" % (reference.name, reference.version)
    for field in ["url", "license", "description"]:
        field_value = getattr(conanfile, field, None)
        if not field_value:
            output.error("%s Conanfile doesn't have '%s'. It is recommended to add it␣
↪as attribute: %s"
                         % (test, field, conanfile_path))

def pre_source(output, conanfile, conanfile_path, **kwargs):
    conanfile_content = tools.load(conanfile_path)
    if "def source(self):" in conanfile_content:
        test = "[IMMUTABLE SOURCES]"
        valid_content = [".zip", ".tar", ".tgz", ".tbz2", ".txz"]
        invalid_content = ["git checkout master", "git checkout devel", "git checkout␣
↪develop"]
        if "git clone" in conanfile_content and "git checkout" in conanfile_content:
            fixed_sources = True
            for invalid in invalid_content:
                if invalid in conanfile_content:
                    fixed_sources = False
        else:
            fixed_sources = False
            for valid in valid_content:
                if valid in conanfile_content:
                    fixed_sources = True

        if not fixed_sources:
            output.error("%s Source files does not come from and immutable place.␣
↪Checkout to a "
                         "commit/tag or download a compressed source file for %s" %␣
↪(test, str(reference)))
```

This hook checks the recipe content prior to it being exported and prior to downloading the sources. Basically the `pre_export()` function checks the attributes of the `conanfile` object to see if there is an URL, a license and a description and if missing, warns the user with a message through the `output`. This is done **before** the recipe is exported to the local cache.

The `pre_source()` function checks if the recipe contains a `source()` method (this time it is using the *conanfile.py* content instead of the `conanfile` object) and in that case it checks if the download of the sources are likely coming from immutable places (a compressed file or a determined **`git checkout`**). This is done **before** the **source()** method of the recipe is called.

Any kind of Python script can be executed. You can create global functions and call them from different hook functions, import from a relative module and warn, error or even raise to abort the Conan client execution.

Other useful task where a hook may come handy are the upload and download actions. There are **pre** and **post** functions for every download/upload as a whole and for fine download tasks such as recipe and package downloads/uploads.

For example they can be used to sign the packages (including a file with the signature) when the package is created and check that signature every time they are downloaded.

Listing 3: *signing_hook.py*

```
import os
from conans import tools

SIGNATURE = "this is my signature"

def post_package(output, conanfile, conanfile_path, **kwargs):
    sign_path = os.path.join(conanfile.package_folder, ".sign")
    tools.save(sign_path, SIGNATURE)
    output.success("Package signed successfully")

def post_download_package(output, conanfile_path, reference, package_id, remote_name,␣
↪**kwargs):
    package_path = os.path.abspath(os.path.join(os.path.dirname(conanfile_path), "..",
↪"package", package_id))
    sign_path = os.path.join(package_path, ".sign")
    content = tools.load(sign_path)
    if content != SIGNATURE:
        raise Exception("Wrong signature")
```

14.5.2 Importing from a module

The hook interface should always be placed inside a Python file with the name of the hook and stored in the *~/.conan/hooks* folder. However, you can use functionalities from imported modules if you have them installed in your system or if they are installed with Conan:

Listing 4: example_hook.py

```
import requests
from conans import tools

def post_export(output, conanfile, conanfile_path, reference, **kwargs):
    cmakelists_path = os.path.join(os.path.dirname(conanfile_path), "CMakeLists.txt")
    tools.replace_in_file(cmakelists_path, "PROJECT(MyProject)", "PROJECT(MyProject CPP)
↪")
    r = requests.get('https://api.github.com/events')
```

You can also import functionalities from a relative module:

```
hooks
├── custom_module
│   ├── custom.py
│   └── __init__.py
└── my_hook.py
```

Inside the *custom.py* from my *custom_module* there is:

```
def my_printer(output):
    output.info("my_printer(): CUSTOM MODULE")
```

And it can be used in the hook importing the module, just like regular Python:

```
from custom_module.custom import my_printer

def pre_export(output, conanfile, conanfile_path, reference, **kwargs):
    my_printer(output)
```

14.5.3 Storage, activation and sharing

Hooks are Python files stored under *~/.conan/hooks* folder and **their file name should be the same used for activation** (the *.py* extension could be indicated or not).

The activation of the hooks is done in the *conan.conf* section named `[hooks]`. The hook names or paths listed under this section will be considered activated.

Listing 5: *conan.conf*

```
...
[hooks]
attribute_checker.py
conan-center.py
my_custom_hook/hook.py
```

They can be easily activated and deactivated from the command line using the **`conan config set`** command:

```
$ conan config set hooks.my_custom_hook/hook  # Activates 'my_custom_hook'

$ conan config rm hooks.my_custom_hook/hook  # Deactivates 'my_custom_hook'
```

There is also an environment variable *CONAN_HOOKS* that you can use to declare which hooks should be activated.

Hooks are considered part of the Conan client configuration and can be shared as usual with the *conan config install* command. However, they can also be managed in isolated Git repositories cloned into the *~/.conan/hooks* folder:

```
$ cd ~/.conan/hooks
$ git clone https://github.com/conan-io/hooks.git conan_hooks
$ conan config set hooks.conan_hooks/hooks/conan-center.py
```

This way you can easily change from one version to another.

14.5.4 Official Hooks

There are some officially maintained hooks in its own repository in GitHub, including the `attribute_checker` that has been packaged with Conan sources for several versions (although it is distributed with Conan still, it is no longer maintained and we may remove it in the future, so we encourage you to install the one in the hooks repository and activate it).

Using the hooks in the official repository is as easy as installing them and activating the ones of interest:

```
conan config install https://github.com/conan-io/hooks.git -sf hooks -tf hooks
conan config set hooks.attribute_checker
```

14.6 Template system

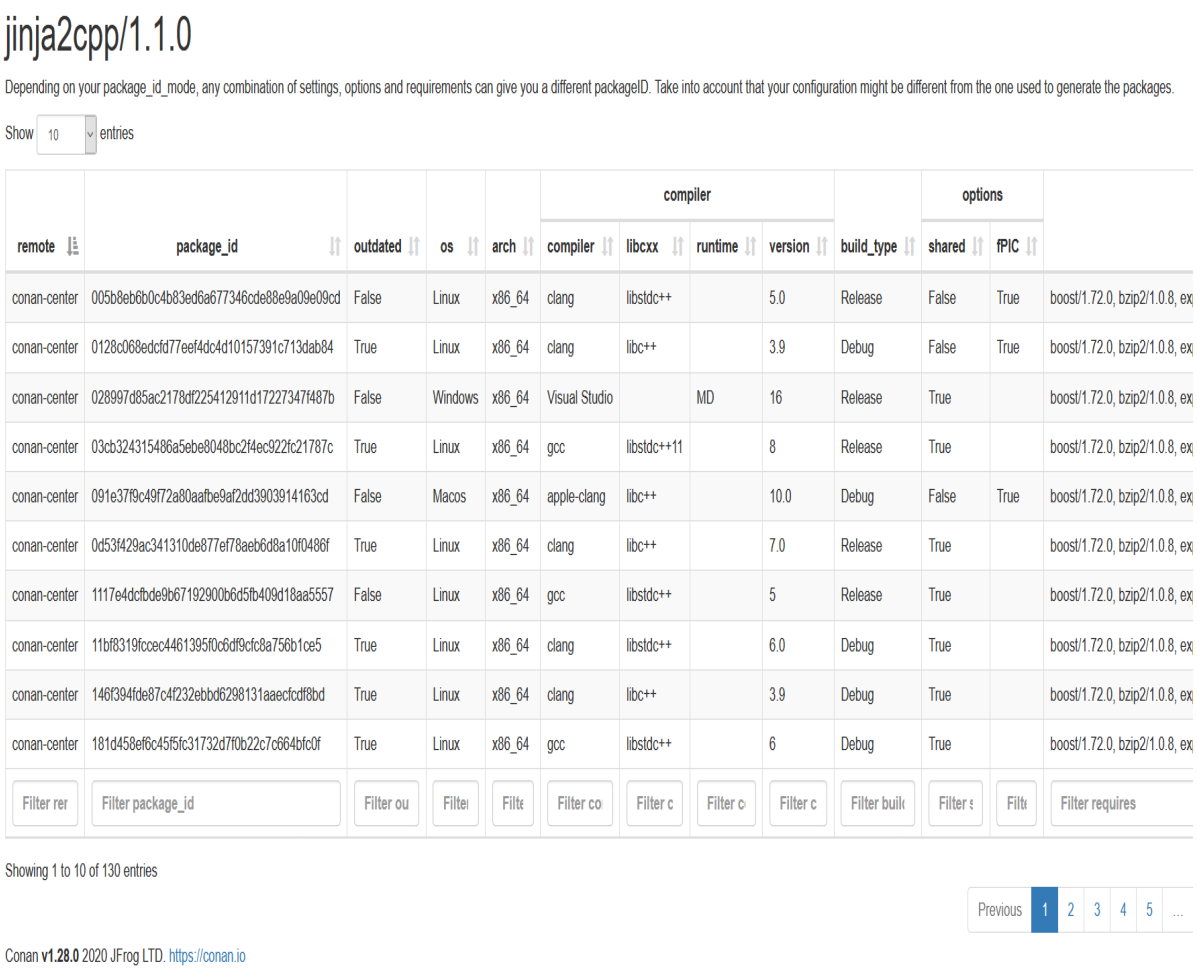
The user can provide their own templates to override some of the files that Conan generates in runtime. This can help to provide custom visualization for some outputs that satisfies specific use-cases or more detailed inputs for companies that want some standarization when creating new recipes for packages.

User provided templates to override Conan default ones, must be stored in the Conan cache under a *templates* directory (*<conan_cache>/templates*). Use *conan config* command to distribute them among your developer team.

14.6.1 HTML output for `conan search` table

> **Warning:** This has to be an considered as an **experimental** feature, we might change the context provided to this templates once we have more examples from the community.

The *conan search* command can generate an HTML table with the results of the query when looking for binaries



jinja2cpp/1.1.0

Depending on your package_id_mode, any combination of settings, options and requirements can give you a different packageID. Take into account that your configuration might be different from the one used to generate the packages.

Show 10 entries

					compiler					options		
remote	package_id	outdated	os	arch	compiler	libcxx	runtime	version	build_type	shared	fPIC	
conan-center	005b8eb6b0c4b83ed6a677346cde88e9a09e09cd	False	Linux	x86_64	clang	libstdc++		5.0	Release	False	True	boost/1.72.0, bzip2/1.0.8, ex
conan-center	0128c068edcfd77eef4dc4d10157391c713dab84	True	Linux	x86_64	clang	libc++		3.9	Debug	False	True	boost/1.72.0, bzip2/1.0.8, ex
conan-center	028997d85ac2178df225412911d17227347f487b	False	Windows	x86_64	Visual Studio		MD	16	Release	True		boost/1.72.0, bzip2/1.0.8, ex
conan-center	03cb324315486a5ebe8048bc2f4ec922fc21787c	True	Linux	x86_64	gcc	libstdc++11		8	Release	True		boost/1.72.0, bzip2/1.0.8, ex
conan-center	091e37f9c49f72a80aafbe9af2dd3903914163cd	False	Macos	x86_64	apple-clang	libc++		10.0	Debug	False	True	boost/1.72.0, bzip2/1.0.8, ex
conan-center	0d53f429ac341310de877ef78aeb6d8a10f0486f	True	Linux	x86_64	clang	libc++		7.0	Release	True		boost/1.72.0, bzip2/1.0.8, ex
conan-center	1117e4dcfbde9b67192900b6d5fb409d18aa5557	False	Linux	x86_64	gcc	libstdc++		5	Release	True		boost/1.72.0, bzip2/1.0.8, ex
conan-center	11bf8319fccec4461395f0c6df9cfc8a756b1ce5	True	Linux	x86_64	clang	libstdc++		6.0	Debug	True		boost/1.72.0, bzip2/1.0.8, ex
conan-center	146f394fde87c4f232ebbd6298131aaecfcdf8bd	True	Linux	x86_64	clang	libc++		3.9	Debug	True		boost/1.72.0, bzip2/1.0.8, ex
conan-center	181d458ef6c45f5fc31732d7f0b22c7c664bfc0f	True	Linux	x86_64	gcc	libstdc++		6	Debug	True		boost/1.72.0, bzip2/1.0.8, ex
Filter rer	Filter package_id	Filter ou	Filte	Filte	Filter co	Filter c	Filter c	Filter c	Filter build	Filter s	Filte	Filter requires

Showing 1 to 10 of 130 entries

Previous 1 2 3 4 5 ...

Conan **v1.28.0** 2020 JFrog LTD. https://conan.io

This is the default Conan provides, but you can use your own Jinja2 documentation template to customize this output to your needs:

- *<cache>/templates/output/search_table.html*.

Context

Conan feeds this template with the information about the packages found, this information is called context and it contains these objects:

- `base_template_path`: absolute path to the directory where the chosen template file is located. It is needed if your output file needs to link assets distributed together with the template file.
- `search`: it contains the pattern used in the command line to search packages.
- `results`: this object contains all the information retrieved from the remotes, it is used to get the headers and the rows.

When the output is a table, the first thing needed are the headers, these can be a single row or two rows like the image above. In order to get the headers you should use `results.get_headers(keys)` with a list of extra `keys` you want to include (see example below). Conan will always return a header for all the different settings and options values, with this `keys` list variable you can retrieve other information that might be useful in your table like `remote`, `reference`, `outdated` or `package_id`.

Then you can use the returned object to get the actual headers:

- single row headers: it just returns a list with all the headers, it is straightforward to use:

```
<thead>
    <tr>
        {%- set headers = results.get_headers(keys=['remote', 'package_id',
↪'outdated']) %}
        {%- for header in headers.row(n_rows=1) %}
        <th>{{ header }}</th>
        {%- endfor %}
    </tr>
</thead>
```

- two-rows headers: it returns a list of tuples like the following one:

```
[('os', ['']), ('arch', ['']), ('compiler', ['', 'version', 'libcxx']),]
```

The first element for this tuple is intended for the top row, while the second element lists all the sub-settings in the top header category. An empty string means there is no category, like `compiler=Visual Studio`.

Composing the table headers in HTML requires some more code in the template:

```
<thead>
    {%- set headers = results.get_headers(keys=['remote', 'package_id', 'outdated
↪']) %}
    {%- set headers2rows = headers.row(n_rows=2) %}
    <tr>
        {%- for category, subheaders in headers2rows %}
            <th rowspan="{% if subheaders|length == 1 and not subheaders[0] %}2{%␣
↪else %}1{% endif %}" colspan="{{ subheaders|length }}">
                {{ category }}
            </th>
        {%- endfor %}
    </tr>
    <tr>
        {%- for category, subheaders in headers2rows %}
            {%- if subheaders|length != 1 or subheaders[0] != '' %}
```

(continues on next page)

(continued from previous page)

```
                {%- for subheader in subheaders %}
                    <th>{{ subheader|default(category, true) }}</th>
                {%- endfor %}
            {%- endif %}
        {%- endfor %}
    </tr>
</thead>
```

Once the headers are done, iterating the rows is easy. You should use `results.packages()` to get an iterable with the list of results and then, for each of the rows, the fields. You need to provide the `headers` to retrieve the fields you need in the proper order according to the table headers:

```
<tbody>
    {%- for package in results.packages() %}
        <tr>
            {%- for item in package.row(headers) %}
                <td>{{ item if item != None else ''}}</td>
            {%- endfor %}
        </tr>
    {%- endfor %}
</tbody>
```

Additionally, the `package` object in the snippet above that represents one of the query results contain some fields that can be useful to compose the text for an `alt` field in the HTML:

- `remote`
- `reference` or `recipe`
- `package_id`
- `outdated`

14.6.2 Graph output for `conan info` command

Warning: This has to be an considered as an **experimental** feature, we might change the context provided to this templates once we have more exmpales from the community.

The *conan info* command can generate a visualization of the dependency graph, it comes in two flavors: *html* and *dot* (GraphViz), but both take the same template parameters. Conan will use the following input files, if found, inside the Conan cache folder:

- HTML output: *<cache>/templates/output/info_graph.html*.
- DOT output: *<cache>/templates/output/info_graph.dot*.

Context

These files should be valid Jinja2 documentation templates and they will be feed with the following context:

- `base_template_path`: absolute path to the directory where the choosen template file is located. It is needed if your output file needs to link assets distributed together with the template file (see HTML example linking CSS and JS files).
- `graph`: this object contains all the information from the graph of dependencies. It offers the following API:
  - `graph.nodes`: list of `Node` objects with the information for each Conan package included in the graph (see below API for this `Node` object).
  - `graph.edges`: list of tuples with all the connections in the dependency graph. First item in the tuple is the consumer `Node` and second item the required `Node`.
  - `graph.binary_color(node)`: function that retrieves the Conan default color based on the `node.binary` value.

The `Node` objects in the context provide all the required information about each package:

- `node.label`: display name for the conanfile.
- `node.short_label`: `name/version` parts of the Conan reference.
- `node.package_id`: the package identifier.
- `node.is_build_requires`:
- **`node.binary`: it identifies where the binary comes from (cache, download, build,** missing, update).
- `node.data()`: returns a dictionary that contains data from the recipe, members are `url`, `homepage`, `license`, `author` and `topics`.

Examples

This is are two examples of templates Conan is currently using for the basic functionality, you can refer to the Jinja2 documentation for more information about the logic and filters your can use in these templates.

Let's us know if you have a cool template you want to share with the Conan community.

Dot files:

Default template for the DOT output contains just the node names and the edges:

```
digraph {
    {%- for src, dst in graph.edges %}
        "{{ src.label }}" -> "{{ dst.label }}"
    {%- endfor %}
}
```

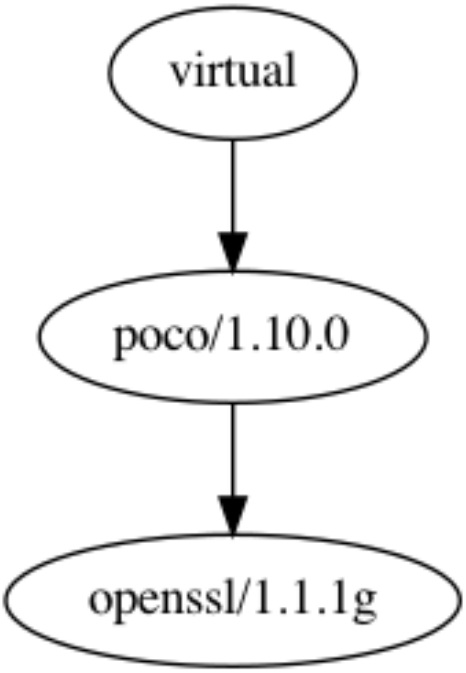
The output will compose a valid `dot` file:

```
conan info poco/1.10.0@ --graph=poco.dot
```

```
digraph {
        "poco/1.10.0" -> "openssl/1.1.1g"
        "virtual" -> "poco/1.10.0"
}
```

Use `dot` to render the default view of the generated graph:

```
dot -Tpng poco.dot > poco.png
```



HTML files:

HTML templates are more complicated than `dot` ones, but the HTML can provide a nicer view of the graph and easily include JavaScript to create an interactive view of the graph.

In this example we assume you have distributed the following files to your cache folder:

```
<cache>/templates/output/css/vis.min.css
<cache>/templates/output/js/vis.min.js
<cache>/templates/output/info_graph.html
```

Our template will we the *info_graph.html* file, and it will use the assets from the local files provided in the cache (most use cases will use files from the internet using the full URL).

These are some snippets from the *info_graph.html* template, it uses the vis.js library:

```
<html lang="en">
    <head>
        {# ... #}
        <script type="text/javascript" src="{{ base_template_path }}/js/vis.min.js"></
↪script>
        <link href="{{ base_template_path }}/css/vis.min.css" rel="stylesheet" type=
```

(continues on next page)

(continued from previous page)

```
↪"text/css"/>
    </head>

    <body>
        {# ... #}

        <div style="width: 100%;">
            <div id="mynetwork"></div>
        </div>

        {# ... #}

        <script type="text/javascript">
            var nodes = new vis.DataSet([
                {%- for node in graph.nodes %}
                    {
                        id: {{ node.id }},
                        label: '{{ node.short_label }}',
                        shape: '{% if node.is_build_requires %}ellipse{% else %}box{%␣
↪endif %}',
                        color: { background: '{{ graph.binary_color(node) }}'},
                        fulllabel: '<h3>{{ node.label }}</h3>' +
                                '<ul>' +
                                '    <li><b>id</b>: {{ node.package_id }}</li>' +
                                {%- for key, value in node.data().items() %}
                                {%- if value %}
                                '    <li><b>{{ key }}</b>: {{ value }}</li>' +
                                {%- endif %}
                                {%- endfor %}
                                '</ul>'
                    }{%- if not loop.last %},{% endif %}
                {%- endfor %}
            ]);
            var edges = new vis.DataSet([
                {%- for src, dst in graph.edges %}
                    { from: {{ src.id }}, to: {{ dst.id }} }{%- if not loop.last %},{%␣
↪endif %}
                {%- endfor %}
            ]);

            var container = document.getElementById('mynetwork');
            var data = {
                nodes: nodes,
                edges: edges
            };
            var network = new vis.Network(container, data, options);
        </script>
    </body>
</html>
```

14.6.3 Package scaffolding for `conan new` command

Warning: This functionality has to be considered as an **experimental** feature. We might change the context provided for these templates once we have more examples from the community.

Using the Conan command **`conan new`** is a very convenient way to start a new project with an example *conanfile.py*. This command has a `--template` argument that you can use to provide a path to a template file for the `conanfile.py` itself or even a path to a folder containing files for a C++ project using Conan recipes.

The argument `--template` can take an absolute path or a relative path. If relative, Conan will look for the files starting in the Conan cache folder `templates/command/new/`. This is very useful in combination with *conan config install* because you can easily share these templates with all your team.

Note: For backwards compatibility reasons, if the `--template` argument takes the path to a single file Conan will look for it in the cache at the path `templates/<filename>` first. This will likely be removed in Conan v2.0.

This mechanism lets you have the Conan cache templates containing not only a *conanfile.py*, but the full C++ project scaffolding. Thus just a single command can get you started:

```
$ conan new mypackage/version --template=header_only
$ conan new mypackage/version --template=conan-center
```

Conan will process all the files found in that folder using Jinja2 engine and also the paths to those files. Thus the following template directory (that does match the conventions for conan-center-index recipes):

```
conan-center/{{name}}/config.yml
            /{{name}}/all/conanfile.py
            /{{name}}/all/conandata.yml
            /{{name}}/all/test_package/conanfile.py
            /{{name}}/all/test_package/CMakeLists.txt
            /{{name}}/all/test_package/main.cpp
```

will be translated to:

```
conan-center/mypackage/config.yml
            /mypackage/all/conanfile.py
            /mypackage/all/conandata.yml
            /mypackage/all/test_package/conanfile.py
            /mypackage/all/test_package/CMakeLists.txt
            /mypackage/all/test_package/main.cpp
```

Then the contents of all the files will be rendered using Jinja2 syntax as well, thus substituting content values with context values - as we will see in the next section.

Context

All the files should be valid Jinja2 templates. They will be feed with the following context:

- `name` and `version`: defined from the command line.
- `package_name`: a *CamelCase* variant of the name. Any valid Conan package name like `package_name`, `package+name`, `package.name` or `package-name` will be converted into a suitable name for a Python class, `PackageName`.
- `conan_version`: an object that renders as the current Conan version, e.g. `1.24.0`.

Example

This is a very simple example for a header only library:

```
# Recipe autogenerated with Conan {{ conan_version }} using `conan new --
↪template` command

from conans import ConanFile


class {{package_name}}Conan(ConanFile):
    name = "{{ name }}"
    version = "{{ version }}"
    settings = "os", "arch", "compiler", "build_type"
    exports_sources = "include/*"

    def package(self):
        self.copy("*.hpp", dst="include")
        self.copy("LICENSE.txt", dst="licenses")

    def package_id(self):
        self.info.header_only()
```

Custom definitions

Sometimes it's needed to provide additional variables for the custom templates. For instance, if it's desired to have `description` and `homepage` to be templated as well:

```
# Recipe autogenerated with Conan {{ conan_version }} using `conan new --
↪template` command

from conans import ConanFile


class {{package_name}}Conan(ConanFile):
    name = "{{ name }}"
    version = "{{ version }}"
    description = "{{ description }}"
    homepage = "{{ homepage }}"
    settings = "os", "arch", "compiler", "build_type"
    exports_sources = "include/*"
```

(continues on next page)

(continued from previous page)

```
    def package(self):
        self.copy("*.hpp", dst="include")
        self.copy("LICENSE.txt", dst="licenses")

    def package_id(self):
        self.info.header_only()
```

With the above template it's now easy to overwrite such extra keywords with values from the command line:

```
$ conan new mypackage/version --template=header_only -d homepage=https://www.
↪example.com -d description="the best package"
```

Predefined templates

Available since: 1.40.0

The Conan client has some predefined templates that can be used with the command `new`. These two templates are related to *Layouts* and offer a simple Hello World example:

- *cmake_lib*: Generates a hello world c++ library based on modern Conan recipe (layout + generate) using CMake as the build system.
- *cmake_exe*: Generates a hello world executable based on modern Conan recipe (layout + generate) using CMake as the build system.
- *msbuild_lib*: Generates a hello world c++ library based on modern Conan recipe (layout + generate) using MSBuild as the build system.
- *msbuild_exe*: Generates a hello world executable based on modern Conan recipe (layout + generate) using MSBuild as the build system.
- *meson_lib*: Generates a hello world c++ library based on modern Conan recipe (layout + generate) using Meson as the build system (since Conan 1.45).
- *meson_exe*: Generates a hello world executable based on modern Conan recipe (layout + generate) using Meson as the build system (since Conan 1.45).
- *bazel_lib*: Generates a hello world c++ library based on modern Conan recipe (layout + generate) using Bazel as the build system (since Conan 1.47).
- *bazel_exe*: Generates a hello world executable based on modern Conan recipe (layout + generate) using Bazel as the build system (since Conan 1.47).
- *autotools_lib*: Generates a hello world c++ library based on modern Conan recipe (layout + generate) using Autotools as the build system (since Conan 1.48).
- *autotools_exe*: Generates a hello world executable based on modern Conan recipe (layout + generate) using Autotools as the build system (since Conan 1.48).

A full example can be found in *Creating Packages* section.

CHAPTER

FIFTEEN

INTEGRATIONS

This topical list of build systems, IDEs, and CI platforms provides information on how conan packages can be consumed, created, and continuously deployed/tested with each, as applicable.

15.1 Compilers

Conan can work with any compiler, the most common ones are already declared in the default *settings.yml*:

- *sun-cc*
- *gcc*
- *Visual Studio*
- *clang*
- *apple-clang*
- *qcc*
- *intel*
- *intel-cc*

Note: Remember that you can *customize Conan* to extend the supported compilers, build systems, etc.

Important: If you work with a compiler like `intel` that uses `Visual Studio` in Windows environments and `gcc` in Linux environments and you are wondering how to manage the compatibility between the packages generated with `intel` and the generated with the pure base compiler (`gcc` or `Visual Studio`) check the *Compatible Packages* and *Compatible Compilers* sections.

Important: If you are working with the new Intel oneAPI compilers, then you should use `intel-cc` one and have a look at *Working with Intel compilers* section.

15.2 Build systems

Conan can be integrated with any build system. This can be done with:

- *Generators*: Conan can write file/s in different formats gathering all the information from the dependency tree, like include directories, library names, library dirs...
- *Build Helpers*: Conan provides some classes to help calling your build system, translating the *settings* and *options* to the arguments, flags or environment variables that your build system expect.

15.2.1  CMake

Note: The new, experimental integration with CMake can be found in *conan.tools.cmake*. This is the integration that will become the standard one in Conan 2.0, and the below generators and integrations will be deprecated and removed.

Conan can be integrated with CMake using different generators, build helpers and custom *findXXX.cmake* files:

`cmake` generator

If you are using CMake to build your project, you can use the `cmake` generator to define all your requirements in CMake syntax. It creates a file named `conanbuildinfo.cmake` that can be imported from your `CMakeLists.txt`.

Listing 1: *conanfile.txt*

```
...
[generators]
cmake
```

When **`conan install`** is executed, a file named *conanbuildinfo.cmake* is created.

You can include *conanbuildinfo.cmake* in your project's *CMakeLists.txt* to manage your requirements. The inclusion of *conanbuildinfo.cmake* doesn't alter the CMake environment at all. It simply provides `CONAN_` variables and some useful macros.

Global variables approach

The simplest way to consume it would be to invoke the `conan_basic_setup()` macro, which will basically set global include directories, libraries directories, definitions, etc. so typically it is enough to call:

```
include(${CMAKE_BINARY_DIR}/conanbuildinfo.cmake)
conan_basic_setup()

add_executable(timer timer.cpp)
target_link_libraries(timer ${CONAN_LIBS})
```

The conan_basic_setup() is divided into smaller macros that should be self explanatory. If you need to do something different, you can just call them individually.

Note: This approach makes all dependencies visible to all CMake targets and may also increase the build times due to unneeded include and library path components. This is particularly relevant if you have multiple targets with different dependencies. In that case, you should consider using the *Targets approach*.

Targets approach

For **modern cmake (>=3.1.2)**, you can use the following approach:

```
include(${CMAKE_BINARY_DIR}/conanbuildinfo.cmake)
conan_basic_setup(TARGETS)

add_executable(timer timer.cpp)
target_link_libraries(timer CONAN_PKG::poco)
```

Using TARGETS as argument, conan_basic_setup() will internally call the macro conan_define_targets() which defines cmake INTERFACE IMPORTED targets, one per package. These targets, named CONAN_PKG::PackageName can be used to link against, instead of using global cmake setup.

See also:

Check the *CMake generator* section to read more.

Note: The CMAKE_MODULE_PATH and CMAKE_PREFIX_PATH contain the paths to the self.info.builddirs of every required package. By default, the root package folder is the only one declared in builddirs. Check *cpp_info* for more information.

---

cmake_multi generator

cmake_multi generator is intended for CMake multi-configuration environments, like Visual Studio and Xcode IDEs that do not configure for a specific build_type, like Debug or Release, but rather can be used for both and switch among Debug and Release configurations with a combo box or similar control. The project configuration for cmake is different, in multi-configuration environments, the flow would be:

```
$ cmake .. -G "Visual Studio 14 Win64"
# Now open the IDE (.sln file) or
$ cmake --build . --config Release
```

While in single-configuration environments (Unix Makefiles, etc):

```
$ cmake .. -G "Unix Makefiles" -DCMAKE_BUILD_TYPE=Release
# Build from your IDE, launching make, or
$ cmake --build .
```

The CMAKE_BUILD_TYPE default, if not specified is Debug.

With the regular conan cmake generator, only 1 configuration at a time can be managed. Then, it is a universal, homogeneous solution for all environments. This is the recommended way, using the regular cmake generator, and just go to the command line and switch among configurations:

```
$ conan install . -s build_type=Release ...
# Work in release, then, to switch to Debug dependencies
$ conan install . -s build_type=Debug ...
```

However, end consumers with heavy usage of the IDE, might want a multi-configuration build. The `cmake_multi` **experimental** generator is able to do that. First, both Debug and Release dependencies have to be installed:

```
$ conan install . -g cmake_multi -s build_type=Release ...
$ conan install . -g cmake_multi -s build_type=Debug  ...
```

These commands will generate 3 files: `conanbuildinfo_release.cmake`, `conanbuildinfo_debug.cmake`, and `conanbuildinfo_multi.cmake`, which includes the other two, and enables its use.

> **Warning:** The `cmake_multi` generator is designed as a helper for consumers, but not for creating packages. If you also want to create a package, see *Creating packages* section.

Global variables approach

The consumer project might write a `CMakeLists.txt` like:

```
project(MyHello)
cmake_minimum_required(VERSION 2.8.12)

include(${CMAKE_BINARY_DIR}/conanbuildinfo_multi.cmake)
conan_basic_setup()

add_executable(say_hello main.cpp)
foreach(_LIB ${CONAN_LIBS_RELEASE})
    target_link_libraries(say_hello optimized ${_LIB})
endforeach()
foreach(_LIB ${CONAN_LIBS_DEBUG})
    target_link_libraries(say_hello debug ${_LIB})
endforeach()
```

Targets approach

Or, if using the modern cmake syntax with targets (where `Hello1` is an example package name that the executable `say_hello` depends on):

```
project(MyHello)
cmake_minimum_required(VERSION 2.8.12)

include(${CMAKE_BINARY_DIR}/conanbuildinfo_multi.cmake)
conan_basic_setup(TARGETS)

add_executable(say_hello main.cpp)
target_link_libraries(say_hello CONAN_PKG::Hello1)
```

There's also a convenient macro for linking to all libraries:

```
project(MyHello)
cmake_minimum_required(VERSION 2.8.12)

include(${CMAKE_BINARY_DIR}/conanbuildinfo_multi.cmake)
conan_basic_setup()

add_executable(say_hello main.cpp)
conan_target_link_libraries(say_hello)
```

With this approach, the end user can open the generated IDE project and switch among both configurations, building the project, or from the command line:

```
$ cmake --build . --config Release
# And without having to conan install again, or do anything else
$ cmake --build . --config Debug
```

Creating packages

The `cmake_multi` generator is just for consumption. It cannot be used to create packages. If you want to be able to both use the `cmake_multi` generator to install dependencies and build your project but also to create packages from that code, you need to specify the regular `cmake` generator for package creation, and prepare the *CMakeLists.txt* accordingly, something like:

```
project(MyHello)
cmake_minimum_required(VERSION 2.8.12)

if(EXISTS ${CMAKE_BINARY_DIR}/conanbuildinfo_multi.cmake)
    include(${CMAKE_BINARY_DIR}/conanbuildinfo_multi.cmake)
else()
    include(${CMAKE_BINARY_DIR}/conanbuildinfo.cmake)
endif()

conan_basic_setup()

add_executable(say_hello main.cpp)
conan_target_link_libraries(say_hello)
```

Then, make sure that the generator `cmake_multi` is **not** specified in the conanfiles, but the users specify it in the command line while installing dependencies:

```
$ conan install . -g cmake_multi
```

See also:

Check the section *Reference/Generators/cmake* to read more about this generator.

cmake_paths generator

This generator is especially useful if you are using CMake based only on the find_package feature to locate the dependencies.

The cmake_paths generator creates a file named conan_paths.cmake declaring:

- CMAKE_MODULE_PATH with the folders of the required packages, to allow CMake to locate the included cmake scripts and FindXXX.cmake files. The folder containing the *conan_paths.cmake* (*self.install_folder* when used in a recipe) is also included, so any custom file will be located too. Check *cmake_find_package generator* generator.
- CMAKE_PREFIX_PATH used by find_library() to locate library files (*.a*, *.lib*, *.so*, *.dll*) in your packages and find_dependency() to locate the transitive dependencies.

Listing 2: conanfile.txt

```
[requires]
zlib/1.2.11
...

[generators]
cmake_paths
```

Listing 3: CMakeList.txt

```
cmake_minimum_required(VERSION 3.0)
project(helloworld)
add_executable(helloworld hello.c)
find_package(Zlib)
if(ZLIB_FOUND)
   include_directories(${ZLIB_INCLUDE_DIRS})
   target_link_libraries (helloworld ${ZLIB_LIBRARIES})
endif()
```

In the example above, the zlib/1.2.11 package is not packaging a custom FindZLIB.cmake file, but the FindZLIB.cmake included in the CMake installation directory (*/Modules*) will locate the zlib library from the Conan package because of the CMAKE_PREFIX_PATH used by the find_library().

If the zlib/1.2.11 would have included a custom FindZLIB.cmake in the package root folder or any declared *self.cpp_info.builddirs*, it would have been located because of the CMAKE_MODULE_PATH variable.

Included as a toolchain

You can use the *conan_paths.cmake* as a toolchain without modifying your *CMakeLists.txt* file:

```
$ mkdir build && cd build
$ conan install ..
$ cmake .. -DCMAKE_TOOLCHAIN_FILE=conan_paths.cmake -G "Unix Makefiles" -DCMAKE_BUILD_
↪TYPE=Release
$ cmake --build .
```

Included using the CMAKE_PROJECT_<PROJECT-NAME>_INCLUDE

With CMAKE_PROJECT_<PROJECT-NAME>_INCLUDE you can specify a file to be included by the project() command. If you already have a toolchain file you can use this variable to include the conan_paths.cmake and insert your toolchain with the CMAKE_TOOLCHAIN_FILE.

```
$ mkdir build && cd build
$ conan install ..
$ cmake .. -G "Unix Makefiles" -DCMAKE_BUILD_TYPE=Release -DCMAKE_PROJECT_helloworld_
↪INCLUDE=build/conan_paths.cmake
$ cmake --build .
```

Included in your *CMakeLists.txt*

Listing 4: CMakeList.txt

```
cmake_minimum_required(VERSION 3.0)
project(helloworld)

include(${CMAKE_BINARY_DIR}/conan_paths.cmake)

add_executable(helloworld hello.c)

find_package(zlib)

if(ZLIB_FOUND)
   include_directories(${ZLIB_INCLUDE_DIRS})
   target_link_libraries (helloworld ${ZLIB_LIBRARIES})
endif()
```

```
$ mkdir build && cd build
$ conan install ..
$ cmake .. -G "Unix Makefiles" -DCMAKE_BUILD_TYPE=Release
$ cmake --build .
```

See also:

Check the section *cmake_paths* to read more about this generator.

Note: The CMAKE_MODULE_PATH and CMAKE_PREFIX_PATH contain the paths to the builddirs of every required package. By default the root package folder is the only declared builddirs directory. Check *cpp_info*.

---

cmake_find_package generator

This generator is especially useful if you are using CMake using the find_package feature to locate the dependencies.

The cmake_find_package generator creates a file for each requirement specified in a conanfile.

The name of the files follows the pattern Find<package_name>.cmake. So for the zlib/1.2.11 package, a FindZLIB.cmake file will be generated.

In a conanfile.py

Listing 5: conanfile.py

```
from conans import ConanFile, CMake, tools


class LibConan(ConanFile):
    ...
    requires = "zlib/1.2.11"
    generators = "cmake_find_package"

    def build(self):
        cmake = CMake(self) # it will find the packages by using our auto-generated␣
↪FindXXX.cmake files
        cmake.configure()
        cmake.build()
```

In the previous example, the CMake build helper will automatically adjust the CMAKE_MODULE_PATH to the conanfile.install_folder, where the generated Find<package_name>.cmake is.

In the CMakeList.txt you do not need to specify or include anything related with Conan at all; just rely on the find_package feature:

Listing 6: CMakeList.txt

```
cmake_minimum_required(VERSION 3.0)
project(helloworld)
add_executable(helloworld hello.c)
find_package(ZLIB)

# Global approach
if(ZLIB_FOUND)
   include_directories(${ZLIB_INCLUDE_DIRS})
   target_link_libraries (helloworld ${ZLIB_LIBRARIES})
endif()

# Modern CMake targets approach
if(TARGET ZLIB::ZLIB)
   target_link_libraries(helloworld ZLIB::ZLIB)
endif()
```

```
$ conan create . user/channel

lib/1.0@user/channel: Calling build()
```

(continues on next page)

(continued from previous page)

```
-- The C compiler identification is AppleClang 9.1.0.9020039
...
-- Conan: Using autogenerated FindZLIB.cmake
-- Found: /Users/user/.conan/data/zlib/1.2.11/_/_/package/
↪0eaf3bfbc94fb6d2c8f230d052d75c6c1a57a4ce/lib/libz.a
lib/1.0@user/channel: Package '72bce3af445a371b892525bc8701d96c568ead8b' created
```

In a *conanfile.txt*

If you are using a `conanfile.txt` file in your project, instead of a `conanfile.py`, this generator can be used together with the *cmake_paths* generator to adjust the `CMAKE_MODULE_PATH` and `CMAKE_PREFIX_PATH` variables automatically and let CMake locate the generated `Find<package_name>.cmake` files.

With `cmake_paths`:

Listing 7: conanfile.txt

```
[requires]
zlib/1.2.11
...

[generators]
cmake_find_package
cmake_paths
```

Listing 8: CMakeList.txt

```
cmake_minimum_required(VERSION 3.0)
project(helloworld)
include(${CMAKE_BINARY_DIR}/conan_paths.cmake)
add_executable(helloworld hello.c)
find_package(ZLIB)

# Global approach
if(ZLIB_FOUND)
   include_directories(${ZLIB_INCLUDE_DIRS})
   target_link_libraries (helloworld ${ZLIB_LIBRARIES})
endif()

# Modern CMake targets approach
if(TARGET ZLIB::ZLIB)
   target_link_libraries(helloworld ZLIB::ZLIB)
endif()
```

```
$ mkdir build && cd build
$ conan install ..
$ cmake .. -G "Unix Makefiles" -DCMAKE_BUILD_TYPE=Release
  -- Conan: Using autogenerated FindZLIB.cmake
  -- Found: /Users/user/.conan/data/zlib/1.2.11/_/_/package/
↪0eaf3bfbc94fb6d2c8f230d052d75c6c1a57a4ce/lib/libz.a
  ...
```

(continues on next page)

(continued from previous page)

```
$ cmake --build .
```

Or you can also adjust `CMAKE_MODULE_PATH` and `CMAKE_PREFIX_PATH` manually.

Without **cmake_paths**, adjusting the variables manually:

Listing 9: conanfile.txt

```
[requires]
zlib/1.2.11
...

[generators]
cmake_find_package
```

Listing 10: CMakeList.txt

```
cmake_minimum_required(VERSION 3.0)
project(helloworld)
list(APPEND CMAKE_MODULE_PATH ${CMAKE_BINARY_DIR})
list(APPEND CMAKE_PREFIX_PATH ${CMAKE_BINARY_DIR})

add_executable(helloworld hello.c)
find_package(ZLIB)

# Global approach
if(ZLIB_FOUND)
   include_directories(${ZLIB_INCLUDE_DIRS})
   target_link_libraries (helloworld ${ZLIB_LIBRARIES})
endif()

# Modern CMake targets approach
if(TARGET ZLIB::ZLIB)
   target_link_libraries(helloworld ZLIB::ZLIB)
endif()
```

See also:

Check the section *cmake_find_package* to read more about this generator and the adjusted CMake variables/targets.

cmake_find_package_multi

Warning: This is an **experimental** feature subject to breaking changes in future releases.

This generator is similar to the *cmake_find_package* generator but it allows working with multi-configuration projects like `Visual Studio` with both `Debug` and `Release`. But there are some differences:

- Only works with CMake > 3.0
- It doesn't generate `Find<package_name>.cmake` modules but `<package_name>Config.cmake`/`<package_name>-config.cmake` files.

- The "global" approach is not supported, only "modern" CMake by using targets.

Usage

```
$ conan install . -g cmake_find_package_multi -s build_type=Debug
$ conan install . -g cmake_find_package_multi -s build_type=Release
```

These commands will generate several files for each dependency in your graph, including a <package_name>Config.cmake or <package_name>-config.cmake that can be located by the CMake find_package(<package_name> CONFIG) command.

Important: Add the CONFIG option to find_package so that *module mode* is explicitly skipped by CMake. This helps to solve issues when there is for example a FindXXXX.cmake file in CMake's default modules directory that could be loaded instead of the <package_name>Config.cmake/<package_name>-config.cmake generated by Conan.

The name of the files follows the pattern <package_name>Config.cmake, and <package_name>-config.cmake for lower case names. So for the zlib/1.2.11 package, a zlib-config.cmake file will be generated.

See also:

Check the section *cmake_find_package_multi* to read more about this generator and the adjusted CMake variables/targets.

Build automation

You can invoke CMake from your conanfile.py file and automate the build of your library/project. Conan provides a CMake() helper. This helper is useful in calling the cmake command both for creating Conan packages or automating your project build with the **conan build .** command. The CMake() helper will take into account your settings in order to automatically set definitions and a generator according to your compiler, build_type, etc.

See also:

Check the section *Building with CMake*.

Find Packages

If a FindXXX.cmake file for the library you are packaging is already available, it should work automatically.

Variables **CMAKE_INCLUDE_PATH** and **CMAKE_LIBRARY_PATH** are set with the requirements paths. The CMake **find_library** function will be able to locate the libraries in the package's folders.

So, you can use **find_package** normally:

```
project(MyHello)
cmake_minimum_required(VERSION 2.8.12)

include(conanbuildinfo.cmake)
conan_basic_setup()

find_package("ZLIB")

if(ZLIB_FOUND)
    add_executable(enough enough.c)
```

(continues on next page)

(continued from previous page)

```
    include_directories(${ZLIB_INCLUDE_DIRS})
    target_link_libraries(enough ${ZLIB_LIBRARIES})
else()
    message(FATAL_ERROR "Zlib not found")
endif()
```

In addition to automatic **find_package** support, **CMAKE_MODULE_PATH** variable is set with the requirements root package paths. You can override the default behavior of any find_package() by creating a `findXXX.cmake` file in your package.

Creating a custom FindXXX.cmake file

Sometimes the "official" CMake FindXXX.cmake scripts are not ready to find our libraries (unsupported library names for specific settings, fixed installation directories like `C:\OpenSSL`, etc.) Or maybe there is no "official" CMake script for our library.

In these cases we can provide a custom **FindXXX.cmake** file in our Conan packages.

1. Create a file named FindXXX.cmake and save it in your Conan package root folder, where XXX is the name of the library that we will use in the **find_package** CMake function. For example, we create a `FindZLIB.cmake` and use `find_package(ZLIB)`. We recommend copying the original FindXXX.cmake file from Kitware (folder Modules/FindXXX.cmake), if available, and modifying it to help find our library files, but it depends a lot; maybe you are interested in creating a new one.

If it's not provided, you can create a basic one. Take a look at this example with the ZLIB library:

FindZLIB.cmake

```
find_path(ZLIB_INCLUDE_DIR NAMES zlib.h PATHS ${CONAN_INCLUDE_DIRS_ZLIB})
find_library(ZLIB_LIBRARY NAMES ${CONAN_LIBS_ZLIB} PATHS ${CONAN_LIB_DIRS_ZLIB})

set(ZLIB_FOUND TRUE)
set(ZLIB_INCLUDE_DIRS ${ZLIB_INCLUDE_DIR})
set(ZLIB_LIBRARIES ${ZLIB_LIBRARY})
mark_as_advanced(ZLIB_LIBRARY ZLIB_INCLUDE_DIR)
```

In the first line we find the path where the headers should be found. We suggest the CONAN_INCLUDE_DIRS_XXX. Then repeat for the library names with CONAN_LIBS_XXX and the paths where the libs are CONAN_LIB_DIRS_XXX.

2. In your conanfile.py file add the `FindXXX.cmake` to the `exports_sources` field:

```
class HelloConan(ConanFile):
    name = "hello"
    version = "0.1"
    ...
    exports_sources = ["FindXXX.cmake"]
```

3. In the package method, copy the `FindXXX.cmake` file to the root:

```
class HelloConan(ConanFile):
    name = "hello"
    version = "0.1"
    ...
```

(continues on next page)

(continued from previous page)

```
    exports_sources = ["FindXXX.cmake"]


    def package(self):
        ...
        self.copy("FindXXX.cmake", ".", ".")
```

Other resources:

- If you want to use the Visual Studio 2017 + CMake integration, *check this how-to*

15.2.2 MSBuild (Visual Studio)

If you are using CMake to generate your Visual Studio projects, this is not the right section, go to *CMake* instead. This section is about native integration with Microsoft MSBuild, using properties files.

Conan can be integrated with **MSBuild** natively, the build system of Visual Studio in different ways:

- Using the `conan.tools.microsoft` tools: `MSBuildDeps`, `MSBuildToolchain` and `MSBuild` helpers to generate properties files for your project, containing information about the project dependencies and toolchain. This is the new integration that is experimental but will become the standard one in Conan 2.0. Go to *conan.tools.microsoft* for more information.
- Using the `visual_studio` or `visual_studio_multi` generators to create a MSBuild properties *conanbuildinfo.props* file. This is the older integration, it is more stable now, but it wil be deprecated and removed in Conan 2.0. Keep reading this page for more information.

With *visual_studio* generator

Use the **visual_studio** generator, or **visual_studio_multi**, if you are maintaining your Visual Studio projects, and want to use Conan to to tell Visual Studio how to find your third-party dependencies.

You can use the **visual_studio** generator to manage your requirements via your *Visual Studio* project.

This generator creates a Visual Studio project properties file, with all the *include paths*, *lib paths*, *libs*, *flags* etc., that can be imported in your project.

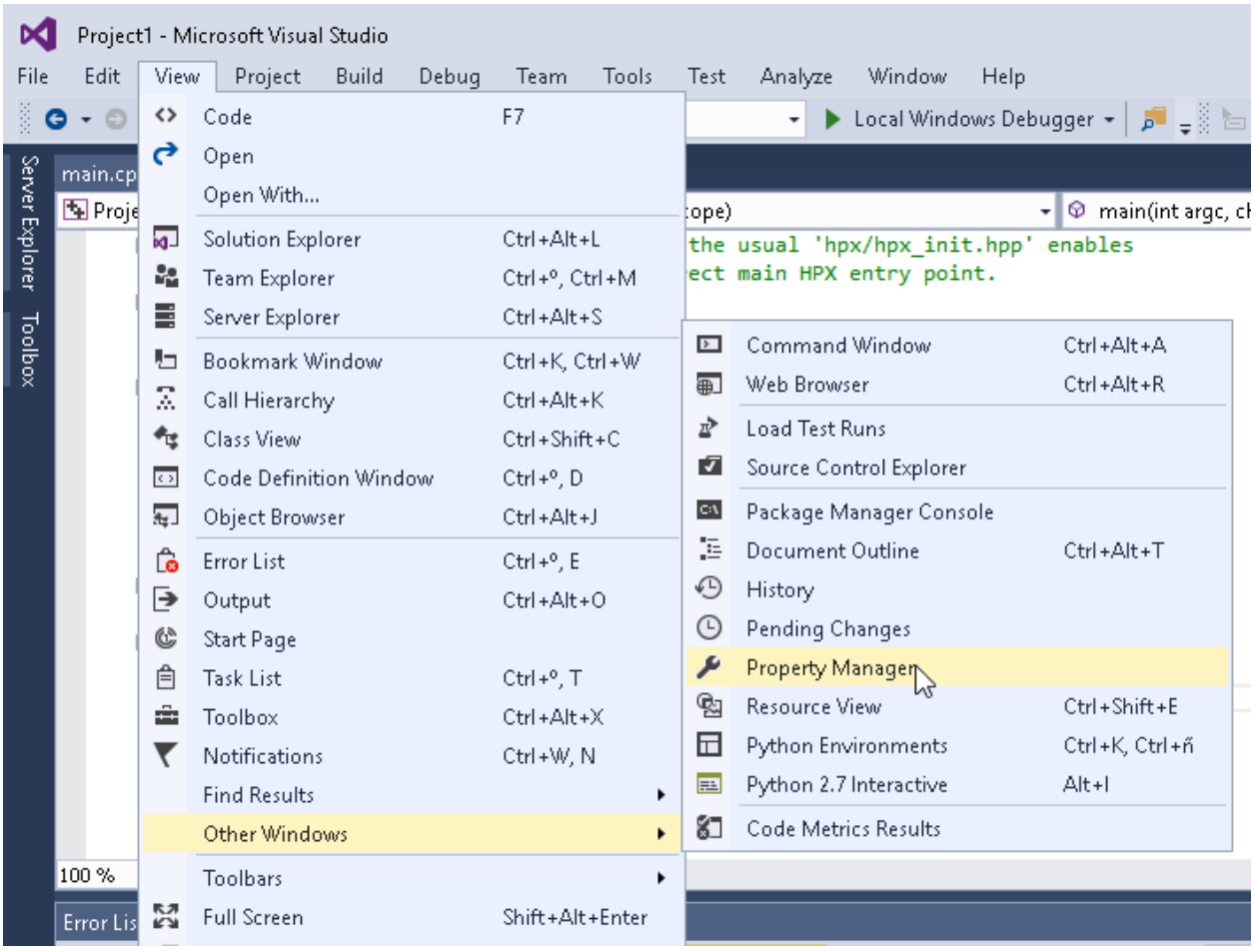
Open `conanfile.txt` and change (or add) the `visual_studio` generator:

```
[requires]
poco/1.9.4

[generators]
visual_studio
```

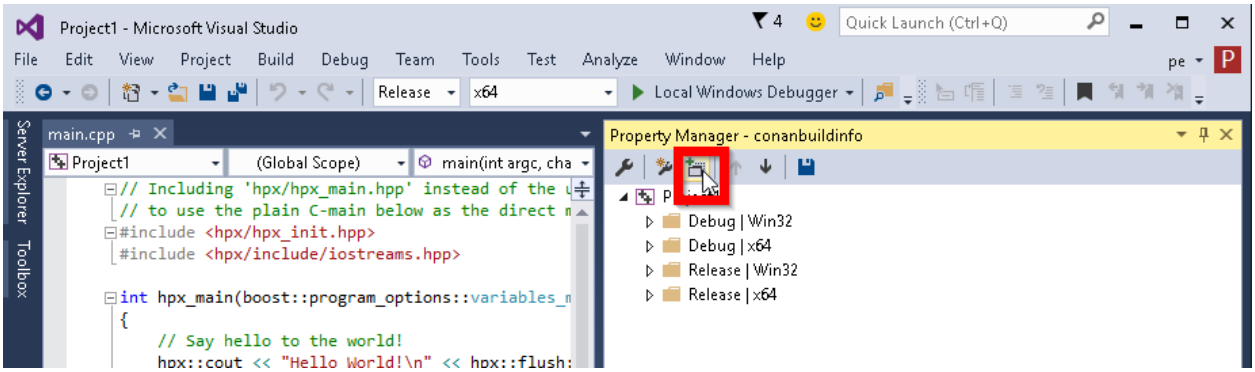
Install the requirements:

```
$ conan install .
```

Go to your Visual Studio project, and open the **Property Manager** (usually in **View -> Other Windows -> Property Manager**).



Click the **+** icon and select the generated `conanbuildinfo.props` file:



Build your project as usual.

Note: Remember to set your project's architecture and build type accordingly, explicitly or implicitly, when issuing the **`conan install`** command. If these values don't match, your build will probably fail.

e.g. **Release/x64**

See also:

Check *visual_studio* for the complete reference.

Build an existing Visual Studio project

You can build an existing Visual Studio from your `build()` method using the *MSBuild()* build helper.

```
from conans import ConanFile, MSBuild

class ExampleConan(ConanFile):
    ...

    def build(self):
        msbuild = MSBuild(self)
        msbuild.build("MyProject.sln")
```

Toolsets

You can use the sub-setting `toolset` of the Visual Studio compiler to specify a custom toolset. It will be automatically applied when using the `CMake()` and `MSBuild()` build helpers. The toolset can also be specified manually in these build helpers with the `toolset` parameter.

By default, Conan will not generate a new binary package if the specified `compiler.toolset` matches an already generated package for the corresponding `compiler.version`. Check the *package_id()* reference to learn more.

See also:

Check the *CMake()* reference section for more info.

15.2.3 Autotools: configure/make

If you are using **configure/make** you can use the **AutoToolsBuildEnvironment** helper. This helper sets `LIBS`, `LDFLAGS`, `CFLAGS`, `CXXFLAGS` and `CPPFLAGS` environment variables based on your requirements.

Check *Building with Autotools* for more info.

15.2.4 Ninja, NMake, Borland

These build systems still don't have a Conan generator for using them natively. However, if you are using CMake, you can instruct Conan to use them instead of the default generator (typically `Unix Makefiles`).

Set it globally in your *conan.conf* file:

```
$ conan config set general.cmake_generator=Ninja
```

or use the environment variable *CONAN_CMAKE_GENERATOR*.

15.2.5 pkg-config and *.pc* files

If you are creating a Conan package for a library (A) and the build system uses *.pc* files to locate its dependencies (B and C) that are Conan packages too, you can follow different approaches.

The main issue to address is the absolute paths. When a user installs a package in the local cache, the directory will probably be different from the directory where the package was created. This could be because of the different computer, the change in Conan home directory or even a different user or channel:

For example, in the machine where the packages were created:

```
/home/user/lasote/.data/storage/zlib/1.2.11/conan/stable
```

In the machine where the library is being reused:

```
/custom/dir/.data/storage/zlib/1.2.11/conan/testing
```

You can see that *.pc* files containing absolute paths won't work with locating the dependencies.

Example of a *.pc* file with an absolute path:

```
prefix=/Users/lasote/.conan/data/zlib/1.2.11/lasote/stable/package/
↪b5d68b3533204ad67e01fa587ad28fb8ce010527
exec_prefix=${prefix}
libdir=${exec_prefix}/lib
sharedlibdir=${libdir}
includedir=${prefix}/include

Name: zlib
Description: zlib compression library
Version: 1.2.11

Requires:
Libs: -L${libdir} -L${sharedlibdir} -lz
Cflags: -I${includedir}
```

To solve this problem there are different approaches that can be followed.

Approach 1: Import and patch the prefix in the *.pc* files

In this approach your **library A** will import to a local directory the *.pc* files from **B** and **C**, then, as they will contain absolute paths, the recipe for **A** will patch the paths to match the current installation directory.

You will need to package the *.pc* files from your dependencies. You can adjust the `PKG_CONFIG_PATH` to let **pkg-config** tool locate them.

```
import os
from conans import ConanFile, tools

class LibAConan(ConanFile):
    name = "libA"
    version = "1.0"
    settings = "os", "compiler", "build_type", "arch"
    exports_sources = "*.cpp"
    requires = "libB/1.0@conan/stable"
```

(continues on next page)

(continued from previous page)

```
    def build(self):
        lib_b_path = self.deps_cpp_info["libB"].rootpath
        copyfile(os.path.join(lib_b_path, "libB.pc"), "libB.pc")
        # Patch copied file with the libB path
        tools.replace_prefix_in_pc_file("libB.pc", lib_b_path)

        with tools.environment_append({"PKG_CONFIG_PATH": os.getcwd()}):
            # CALL YOUR BUILD SYSTEM (configure, make etc)
            # E.g., self.run('g++ main.cpp $(pkg-config libB --libs --cflags) -o main')
```

Approach 2: Prepare and package *.pc* files before packaging them

With this approach you will patch the *.pc* files from B and C before packaging them. The goal is to replace the absolute path (the variable part of the path) with a variable placeholder. Then in the consumer package A, declare the variable using `--define-variable` when calling the **`pkg-config`** command.

This approach is cleaner than approach 1, because the packaged files are already prepared to be reused with or without Conan by declaring the needed variable. And it's unneeded to import the *.pc* files to the consumer package. However, you need B and C libraries to package the *.pc* files correctly.

Library B recipe (preparing the *.pc* file):

```
from conans import ConanFile, tools

class LibBConan(ConanFile):
    ....

    def build(self):
        ...
        tools.replace_prefix_in_pc_file("mypcfile.pc", "${package_root_path_lib_b}")

    def package(self):
        self.copy(pattern="*.pc", dst="", keep_path=False)
```

Library A recipe (importing and consuming *.pc* file):

```
class LibAConan(ConanFile):
    ....

    requires = "libB/1.0@conan/stable, libC/1.0@conan/stable"

    def build(self):

        args = '--define-variable package_root_path_lib_b=%s' % self.deps_cpp_info["libB
↪"].rootpath
        args += ' --define-variable package_root_path_lib_c=%s' % self.deps_cpp_info[
↪"libC"].rootpath
        pkgconfig_exec = 'pkg-config ' + args

        vars = {'PKG_CONFIG': pkgconfig_exec, # Used by autotools
                'PKG_CONFIG_PATH': "%s:%s" % (self.deps_cpp_info["libB"].rootpath,
```

(continues on next page)

(continued from previous page)

```
                                            self.deps_cpp_info["libC"].rootpath)}

        with tools.environment_append(vars):
            # Call autotools (./configure ./make, will read PKG_CONFIG)
            # Or directly declare the variables:
            self.run('g++ main.cpp $(pkg-config %s libB --libs --cflags) -o main' % args)
```

Approach 3: Use `--define-prefix`

If you have available **pkg-config** >= 0.29 and you have only one dependency, you can directly use the **--define-prefix** option to declare a custom `prefix` variable. With this approach you won't need to patch anything, just declare the correct variable.

Approach 4: Use `PKG_CONFIG_$PACKAGE_$VARIABLE`

If you have **pkg-config** >= 0.29.1 available, you can manage multiple dependencies declaring **N** variables with the prefixes:

```
class LibAConan(ConanFile):
    ....

    requires = "libB/1.0@conan/stable, libC/1.0@conan/stable"

    def build(self):

        vars = {'PKG_CONFIG_libB_PREFIX': self.deps_cpp_info["libB"].rootpath,
                'PKG_CONFIG_libC_PREFIX': self.deps_cpp_info["libC"].rootpath,
                'PKG_CONFIG_PATH': "%s:%s" % (self.deps_cpp_info["libB"].rootpath,
                                              self.deps_cpp_info["libC"].rootpath)}

        with tools.environment_append(vars):
            # Call the build system
```

Approach 5: Use the `pkg_config` generator

If you use `package_info()` in library B and library C, and specify all the library names and any other needed flag, you can use the `pkg_config` generator for **library A**. Those files doesn't need to be patched, because they are dynamically generated with the correct path.

So it can be a good solution in case you are building **library A** with a build system that manages *.pc* files like *Meson Build* or *AutoTools*:

Meson Build

```
from conans import ConanFile, tools, Meson
import os

class ConanFileToolsTest(ConanFile):
    generators = "pkg_config"
    requires = "lib_a/0.1@conan/stable"
```

(continues on next page)

(continued from previous page)

```
    settings = "os", "compiler", "build_type"

    def build(self):
        meson = Meson(self)
        meson.configure()
        meson.build()
```

Autotools

```
from conans import ConanFile, tools, AutoToolsBuildEnvironment
import os

class ConanFileToolsTest(ConanFile):
    generators = "pkg_config"
    requires = "lib_a/0.1@conan/stable"
    settings = "os", "compiler", "build_type"

    def build(self):
        autotools = AutoToolsBuildEnvironment(self)
        # When using the pkg_config generator, self.build_folder will be added to PKG_
↪CONFIG_PATH
        # so pkg_config will be able to locate the generated pc files from the requires␣
↪(LIB_A)
        autotools.configure()
        autotools.make()
```

See also:

Check the *tools.PkgConfig()*, a wrapper of the **pkg-config** tool that allows to extract flags, library paths, etc. for any *.pc* file.

15.2.6 Boost Build

Caution: This generator is deprecated in favor of the `b2` generator. See *generator b2*.

With this generator `boost-build` you can generate a `project-root.jam` file to be used with your Boost Build system.

Check the *generator boost-build*

15.2.7 B2 (Boost Build)

With this generator `b2` you can generate a `conanbuildinfo.jam` file to be used with your B2 system.

Check the *generator b2*

15.2.8 QMake

The `qmake` generator will generate a *conanbuildinfo.pri* file that can be used for your qmake builds.

```
$ conan install . -g qmake
```

Add `conan_basic_setup` to `CONFIG` and include the file in your existing project *.pro* file:

Listing 11: *yourproject.pro*

```
...

CONFIG += conan_basic_setup
include(conanbuildinfo.pri)
```

This will include all the statements in *conanbuildinfo.pri* in your project. Include paths, libraries, defines, etc. will be set up for all requirements you have defined as dependencies in a *conanfile.txt*.

If you'd prefer to manually add the variables for each dependency, you can do so by skipping the `CONFIG` statement and only including *conanbuildinfo.pri*:

Listing 12: *yourproject.pro*

```
# ...

include(conanbuildinfo.pri)

# you may now modify your variables manually for each library, such as
# INCLUDEPATH += CONAN_INCLUDEPATH_POCO
```

The `qmake` generator allows multi-configuration packages, i.e. packages that contains both Debug and Release artifacts.

Example

Tip: This complete example is stored in https://github.com/memsharded/qmake_example

This example project will depend on a multi-configuration (Debug/Release) "Hello World" package. It should be installed first:

```
$ git clone https://github.com/memsharded/hello_multi_config
$ cd hello_multi_config
```

(continues on next page)

(continued from previous page)

```
$ conan create . memsharded/testing
hello/0.1@memsharded/testing export: Copied 1 '.txt' file: CMakeLists.txt
hello/0.1@memsharded/testing export: Copied 1 '.cpp' file: hello.cpp
hello/0.1@memsharded/testing export: Copied 1 '.h' file: hello.h
hello/0.1@memsharded/testing: A new conanfile.py version was exported
```

This hello package is created with CMake, but that doesn't matter for this example, as it can be consumed from a qmake project with the configuration showed before.

Now let's get the qmake project and install its *hello/0.1@memsharded/testing* dependency:

```
$ git clone https://github.com/memsharded/qmake_example
$ cd qmake_example
$ conan install .
PROJECT: Installing C:\Users\memsharded\qmake_example\conanfile.txt
Requirements
    hello/0.1@memsharded/testing from local cache - Cache
Packages
    hello/0.1@memsharded/testing:15af85373a5688417675aa1e5065700263bf257e - Cache

hello/0.1@memsharded/testing: Already installed!
PROJECT: Generator qmake created conanbuildinfo.pri
PROJECT: Generator txt created conanbuildinfo.txt
PROJECT: Generated conaninfo.txt
```

As you can see, we got the dependency information in the *conanbuildinfo.pri* file. You can inspect the file to see the variables generated. Now let's build the project for Release and then for Debug:

```
$ qmake
$ make
$ ./helloworld
> Hello World Release!

# now let's build the Debug one
$ make clean
$ qmake CONFIG+=debug
$ make
$ ./helloworld
> Hello World Debug!
```

See also:

Check the complete reference of the *qmake generator*.

15.2.9  Premake

Since Conan 1.9.0 the `premake` generator is built-in and works with **`premake5`**, so the following should be enough to use it:

```
[generators]
premake
```

Example

We are going to use the same example from *Getting Started*, a MD5 hash calculator app.

This is the main source file for it:

Listing 13: main.cpp

```
#include "Poco/MD5Engine.h"
#include "Poco/DigestStream.h"

#include <iostream>


int main(int argc, char** argv)
{
    Poco::MD5Engine md5;
    Poco::DigestOutputStream ds(md5);
    ds << "abcdefghijklmnopqrstuvwxyz";
    ds.close();
    std::cout << Poco::DigestEngine::digestToHex(md5.digest()) << std::endl;
    return 0;
}
```

As this project relies on the Poco Libraries, we are going to create a *conanfile.txt* with our requirement and also declare the Premake generator:

Listing 14: conanfile.txt

```
[requires]
poco/1.9.4

[generators]
premake
```

In order to use the new generator within your project, use the following Premake script as a reference:

Listing 15: premake5.lua

```
-- premake5.lua

include("conanbuildinfo.premake.lua")
```

(continues on next page)

(continued from previous page)

```
workspace("ConanPremakeDemo")
    conan_basic_setup()

    project "ConanPremakeDemo"
        kind "ConsoleApp"
        language "C++"
        targetdir "bin/%{cfg.buildcfg}"

        linkoptions { conan_exelinkflags }

        files { "**.h", "**.cpp" }

        filter "configurations:Debug"
        defines { "DEBUG" }
        symbols "On"

        filter "configurations:Release"
        defines { "NDEBUG" }
        optimize "On"
```

Now we are going to let Conan retrieve the dependencies and generate the dependency information in a *conanbuildinfo.lua*:

```
$ conan install .
```

Then let's call **premake** to generate our project:

- Use this command for Windows Visual Studio:

```
$ premake5 vs2017  # Generates a .sln
```

- Use this command for Linux or macOS:

```
$ premake5 gmake  # Generates a makefile
```

Now you can build your project with Visual Studio or Make.

See also:

Check the complete reference of the *premake generator*.

15.2.10 XMake

Install third-party packages:

After version 2.2.5, xmake supports installing for dependency libraries of conan package manager.

Listing 16: xmake.lua

```
-- xmake.lua

add_requires("conan::zlib/1.2.11@conan/stable", {alias = "zlib", debug = true})
add_requires("conan::openssl/1.1.1g", {alias = "openssl",
    configs = {options = "OpenSSL:shared=True"}})

target("test")
    set_kind("binary")
    add_files("src/*.c")
    add_packages("openssl", "zlib")
```

After executing xmake to compile:

```
$ xmake
checking for the architecture ... x86_64
checking for the Xcode directory ... /Applications/Xcode.app
checking for the SDK version of Xcode ... 10.14
note: try installing these packages (pass -y to skip confirm)?
  -> conan::zlib/1.2.11@conan/stable  (debug)
  -> conan::openssl/1.1.1g
please input: y (y/n)

  => installing conan::zlib/1.2.11@conan/stable .. ok
  => installing conan::openssl/1.1.1g .. ok

[  0%]: ccache compiling.release src/main.c
[100%]: linking.release test
```

Find a conan package

XMake v2.2.6 and later versions also support finding the specified package in the Conan cache:

Listing 17: xmake.lua

```
...
find_packages("conan::openssl/1.1.1g")
```

Test command for finding package

We can also add a third-party package manager prefix to test:

```
xmake l find_packages conan::openssl/1.1.1g
```

Note: It should be noted that if the find_package command is executed in the project directory with xmake.lua, there will be a cache. If the search fails, the next lookup will also use the cached result. If you want to force a retest every time, Please switch to the non-project directory to execute the above command.

15.2.11 Make

Warning: This integration is to be deprecated in Conan 2.0. Check *the conan.tools.gnu Autotools* integration.

Conan provides the *Make generator* to integrate with plain Makefiles

The `make` generator outputs all the variables related to package dependencies into a file which is named *conanbuildinfo.mak*. The `make` toolchain outputs all the variables related to settings, options, and platform into a file which is named `conan_toolchain.mak`.

To use the generator, indicate it in your `conanfile` like this:

Listing 18: *conanfile.txt*

```
[generators]
make
```

Listing 19: *conanfile.py*

```
class MyConan(ConanFile):
    ...
    generators = "make"
```

Example

We are going to use the same example from *Getting Started*, a MD5 hash calculator app.

This is the main source file for it:

Listing 20: main.cpp

```
#include "Poco/MD5Engine.h"
#include "Poco/DigestStream.h"

#include <iostream>


int main(int argc, char** argv)
{
    Poco::MD5Engine md5;
    Poco::DigestOutputStream ds(md5);
    ds << "abcdefghijklmnopqrstuvwxyz";
    ds.close();
    std::cout << Poco::DigestEngine::digestToHex(md5.digest()) << std::endl;
    return 0;
}
```

In order to use this generator within your project, use the following Makefile as a reference:

Listing 21: Makefile

```
#-------------------------------------
#     Prepare flags from make generator
```

(continues on next page)

(continued from previous page)

```
#----------------------------------------

include conanbuildinfo.mak

CFLAGS              += $(CONAN_CFLAGS)
CXXFLAGS            += $(CONAN_CXXFLAGS)
CPPFLAGS            += $(addprefix -I, $(CONAN_INCLUDE_DIRS))
CPPFLAGS            += $(addprefix -D, $(CONAN_DEFINES))
LDFLAGS             += $(addprefix -L, $(CONAN_LIB_DIRS))
LDLIBS              += $(addprefix -l, $(CONAN_LIBS))
EXELINKFLAGS        += $(CONAN_EXELINKFLAGS)

#----------------------------------------
#     Make variables for a sample App
#----------------------------------------

SRCS          = main.cpp
OBJS          = main.o
EXE_FILENAME  = main

#----------------------------------------
#     Make Rules
#----------------------------------------

.PHONY                  :   exe
exe                     :   $(EXE_FILENAME)

$(EXE_FILENAME)         :   $(OBJS)
    g++ $(OBJS) $(CXXFLAGS) $(LDFLAGS) $(LDLIBS) -o $(EXE_FILENAME)

%.o                     :   $(SRCS)
    g++ -c $(CPPFLAGS) $(CXXFLAGS) $< -o $@
```

Now we are going to let Conan retrieve the dependencies, generate the dependency information in the file `conanbuildinfo.mak`, and generate the options and settings information in the file `conan_toolchain.mak`:

```
$ conan install .
```

Then let's call **make** to generate our project:

```
$ make exe
```

Now you can run your application with `./main`.

See also:

Complete reference for *Make generator*

15.2.12 qbs

Conan provides a **qbs** generator, which will generate a `conanbuildinfo.qbs` file that can be used for your qbs builds.

Add `conanbuildinfo.qbs` as a reference on the project level and a Depends item with the name `conanbuildinfo`:

yourproject.qbs

```
import qbs

Project {
   references: ["conanbuildinfo.qbs"]
   Product {
        type: "application"
        consoleApplication: true
        files: [
            "conanfile.txt",
            "main.cpp",
        ]
        Depends { name: "cpp" }
        Depends { name: "ConanBasicSetup" }
   }
}
```

This will include the product called `ConanBasicSetup` which holds all the necessary settings for all your dependencies.

If you'd prefer to manually add each dependency, just replace `ConanBasicSetup` with the dependency you would like to include. You may also specify multiple dependencies:

yourproject.qbs

```
import qbs

Project {
   references: ["conanbuildinfo.qbs"]
   Product {
        type: "application"
        consoleApplication: true
        files: [
            "conanfile.txt",
            "main.cpp",
        ]
        Depends { name: "cpp" }
        Depends { name: "catch" }
        Depends { name: "Poco" }
   }
}
```

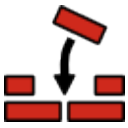
See also:

Check the *Reference/Generators/qbs* section for get more details.

15.2.13  Meson Build

If you are using **Meson Build** as your library build system, you can use the **Meson** build helper. This helper has `.configure()` and `.build()` methods available to ease the call to Meson build system. It also will automatically take the `pc files` of your dependencies when using the *pkg_config generator*.

Check *Building with Meson Build* for more info.

15.2.14  SCons

SCons can be used both to generate and consume Conan packages via the *scons* generator. The package recipe `build()` method could be similar to:

```
class PkgConan(ConanFile):
    settings = 'os', 'compiler', 'build_type', 'arch'
    requires = 'hello/1.0@user/stable'
    generators = "scons"
    ...

    def build(self):
        debug_opts = ['--debug-build'] if self.settings.build_type == 'Debug' else []
        os.makedirs("build")
        # FIXME: Compiler, version, arch are hardcoded, not parametrized
        with tools.chdir("build"):
            self.run(['scons', '-C', '{}/src'.format(self.source_folder)] + debug_opts)

    ...
```

The `SConscript` build script can load the generated `SConscript_conan` file that contains the information of the dependencies, and use it to build

```
conan = SConscript('{}/SConscript_conan'.format(build_path_relative_to_sconstruct))
if not conan:
    print("File `SConscript_conan` is missing.")
    print("It should be generated by running `conan install`.")
    sys.exit(1)

flags = conan["conan"]
version = flags.pop("VERSION")
env.MergeFlags(flags)
env.Library("hello", "hello.cpp")
```

A complete example with a *test_package* that uses SCons too is available in the following GitHub repository. Give it a try!

```
$ git clone https://github.com/memsharded/conan-scons-template
$ cd conan-scons-template
```

(continues on next page)

(continued from previous page)

```
$ conan create . demo/testing
> Hello World Release!
$ conan create . demo/testing -s build_type=Debug
> Hello World Debug!
```

15.2.15 Compilers on command line

The **compiler_args** generator creates a file named `conanbuildinfo.args` containing command line arguments to invoke `gcc`, `clang` or `cl` (Visual Studio) compiler.

Now we are going to compile the *getting started* example using **compiler_args** instead of the **cmake** generator.

Open `conanfile.txt` and change (or add) **compiler_args** generator:

```
[requires]
poco/1.9.4

[generators]
compiler_args
```

Install the requirements (from the mytimer/build folder):

```
$ conan install ..
```

Note: Remember, if you don't specify settings in the **install command** with **-s**, Conan will use the detected defaults. You can always change them by editing the `~/.conan/profiles/default` or override them with "-s" parameters.

The generated `conanbuildinfo.args` show:

```
-DPOCO_STATIC=ON -DPOCO_NO_AUTOMATIC_LIBS
-I/home/user/.conan/data/poco/1.9.4/_/_/package/58080bce1cc38259eb7c282aa95c25aecde8efe4/
↪include
-I/home/user/.conan/data/openssl/1.0.2t/_/_/package/
↪f99afdbf2a1cc98ba2029817b35103455b6a9b77/include
-I/home/user/.conan/data/zlib/1.2.11/_/_/package/
↪6af9cc7cb931c5ad942174fd7838eb655717c709/include
-m64 -O3 -s -DNDEBUG
-Wl,-rpath="/home/user/.conan/data/poco/1.9.4/_/_/package/
↪58080bce1cc38259eb7c282aa95c25aecde8efe4/lib"
-Wl,-rpath="/home/user/.conan/data/openssl/1.0.2t/_/_/package/
↪f99afdbf2a1cc98ba2029817b35103455b6a9b77/lib"
-Wl,-rpath="/home/user/.conan/data/zlib/1.2.11/_/_/package/
↪6af9cc7cb931c5ad942174fd7838eb655717c709/lib"
-L/home/user/.conan/data/poco/1.9.4/_/_/package/58080bce1cc38259eb7c282aa95c25aecde8efe4/
↪lib
-L/home/user/.conan/data/openssl/1.0.2t/_/_/package/
↪f99afdbf2a1cc98ba2029817b35103455b6a9b77/lib
-L/home/user/.conan/data/zlib/1.2.11/_/_/package/
↪6af9cc7cb931c5ad942174fd7838eb655717c709/lib
-lPocoMongoDB -lPocoNetSSL -lPocoNet -lPocoCrypto -lPocoDataSQLite -lPocoData -lPocoZip -
↪lPocoUtil
```

(continues on next page)

(continued from previous page)

```
-lPocoXML -lPocoJSON -lPocoRedis -lPocoFoundation
-lrt -lssl -lcrypto -ldl -lpthread -lz
-D_GLIBCXX_USE_CXX11_ABI=1
```

This is hard to read, but those are just the **compiler_args** parameters needed to compile our program:

- **-I** options with headers directories
- **-L** for libraries directories
- **-l** for library names
- and so on... see the *complete reference here*

It's almost the same information we can see in `conanbuildinfo.cmake` and many other generators' files.

Run:

```
$ mkdir bin
$ g++ ../timer.cpp @conanbuildinfo.args -std=c++14 -o bin/timer
```

Note: "@conanbuildinfo.args" appends all the file contents to g++ command parameters

```
$ ./bin/timer
 Callback called after 250 milliseconds.
 ...
```

To invoke `cl` (Visual Studio compiler):

```
$ cl /EHsc timer.cpp @conanbuildinfo.args
```

You can also use the generator within your `build()` method of your conanfile.py.

Check the *Reference, generators, compiler_args* section for more info.

15.2.16 Bazel

If you are using **Bazel** as your library build system, you can use the **Bazel** build helper. This helper has a `.build()` method available to ease the call to Bazel build system.

If you want to declare Conan dependencies in your project, you must do it, as usual, in the **conanfile.py** file. For example:

```
class BazelExampleConan(ConanFile):
    name = "bazel-example"
    ....
    requires = "boost/1.76.0"
```

Then, tell Bazel to use that dependencies by adding this to the **WORKSPACE** file:

```
load("@//conandeps:dependencies.bzl", "load_conan_dependencies")
load_conan_dependencies()
```

After that, just update the BUILD files where you need to use the new dependency:

```
cc_binary(
    name = "hello-world",
    srcs = ["hello-world.cc"],
    deps = [
        "@boost//:boost",
    ],
)
```

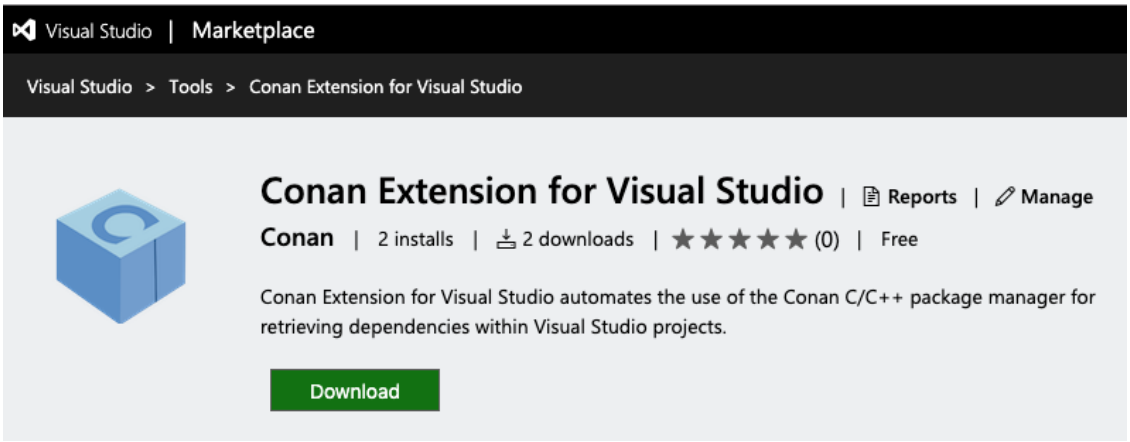
15.3 IDEs

You can develop both the recipes and your libraries using you IDE.

15.3.1 Visual Studio

Conan Extension for Visual Studio

Thanks to the invaluable help of our community we manage to develop and maintain a free extension for Visual Studio in the Microsoft Marketplace, it is called Conan Extension for Visual Studio and it provides integration with Conan using the *Visual Studio generators*.



You can install it into your IDE using the **Extensions manager** and start using it right away. This extension will look for a *conanfile.py* (or *conanfile.txt*) and retrieve the requirements declared in it that match your build configuration (it will build them from sources if no binaries are available).

Note: Location of the conanfile

In version `1.0` of the extension, the algorithm to look for the *conanfile.py* (preferred) or *conanfile.txt* is very naïve: It will start looking for those files in the directory where the **Visual Studio project file** is located and then it will walk recursively into parent directories to look for them.

The extension creates a property sheet file and adds it to the project, so all the information from the dependencies handled by Conan should be added (as inherited properties) to those already available in your projects.

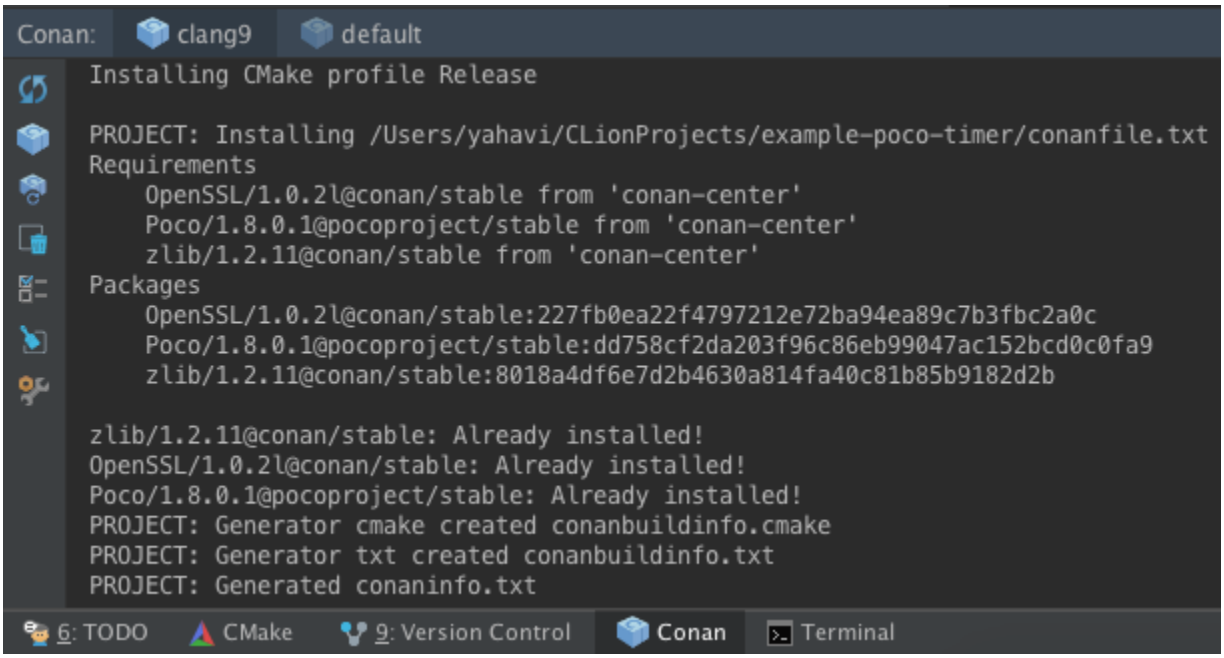
At this moment (release `v1.0.x`) the extension is under heavy development, some behaviors may change and new features will be added. You can subscribe to its repository to stay updated and, of course, any feedback about it will be more than welcome.

General Integration

Check the *MSBuild() integration*, that contains information about Build Helpers and generators to be used with Visual Studio.

15.3.2 CLion

There is an official Jetbrains plugin Conan plugin for CLion.



You can read how to use it in the following blog post

General Integration

CLion uses **CMake** as the build system of projects, so you can use the *CMake generator* to manage your requirements in your CLion project.

Just include the `conanbuildinfo.cmake` this way:

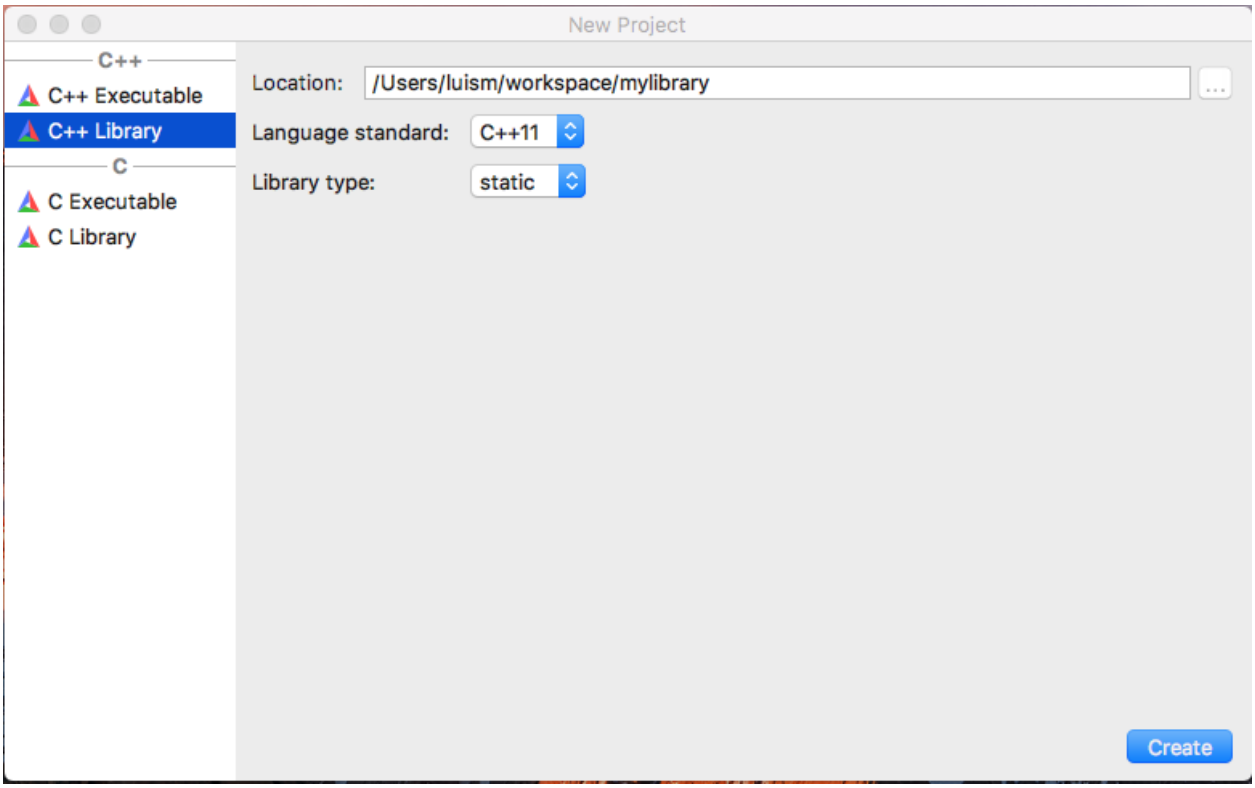
```
if(EXISTS ${CMAKE_BINARY_DIR}/conanbuildinfo.cmake)
    include(${CMAKE_BINARY_DIR}/conanbuildinfo.cmake)
    conan_basic_setup()
else()
    message(WARNING "The file conanbuildinfo.cmake doesn't exist, you have to run conan␣
↪install first")
endif()
```

If the `conanbuildinfo.cmake` file is not found, it will print a warning message in the `Messages` console of your CLion IDE.

Using packages in a CLion project

Let see an example of how to consume Conan packages in a CLion project. We are going to require and use the `zlib` conan package.

1. Create a new CLion project



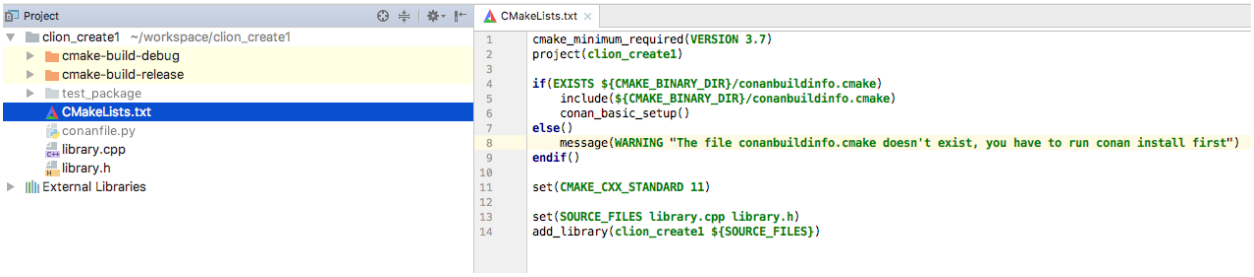
2. Edit the `CMakeLists.txt` file and add the following lines:

```
if(EXISTS ${CMAKE_BINARY_DIR}/conanbuildinfo.cmake)
    include(${CMAKE_BINARY_DIR}/conanbuildinfo.cmake)
```

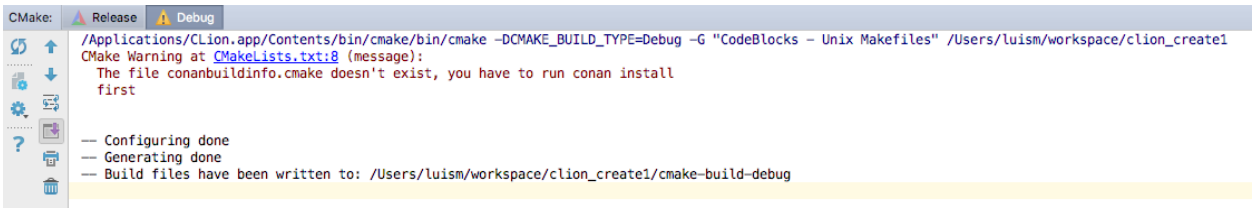
(continues on next page)

(continued from previous page)

```
    conan_basic_setup()
else()
    message(WARNING "The file conanbuildinfo.cmake doesn't exist, you have to run conan␣
↪install first")
endif()
```

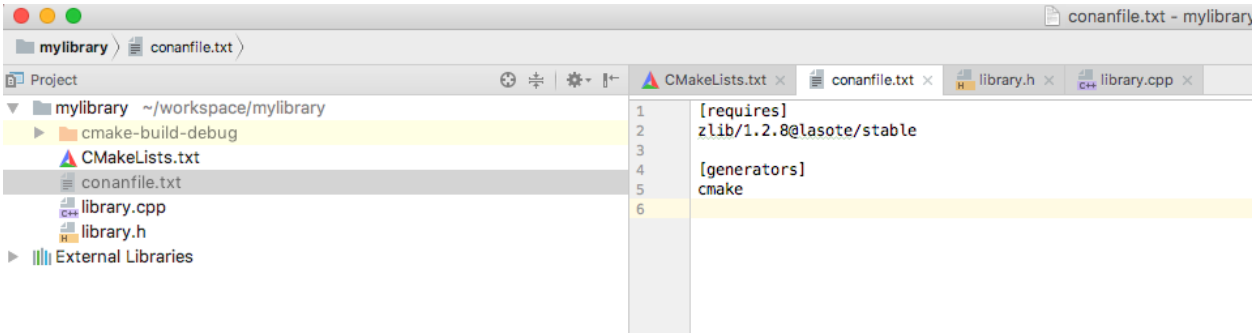


3. CLion will reload your CMake project and you will be able to see a Warning in the console, because the `conanbuildinfo.cmake` file still doesn't exist:



4. Create a `conanfile.txt` with all your requirements and use the `cmake` generator. In this case we only require the zlib library from a Conan package:

```
[requires]
zlib/1.2.11

[generators]
cmake
```



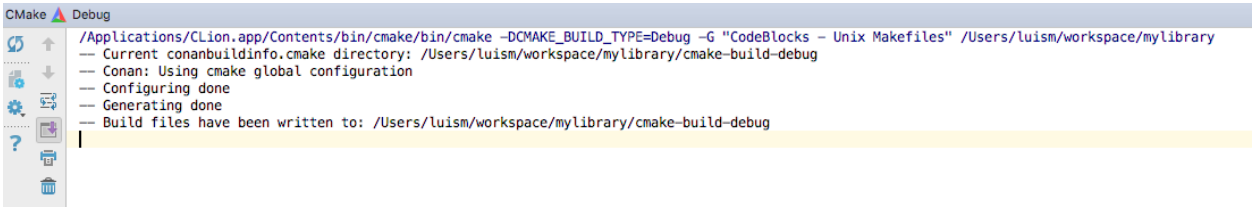
5. Now you can run **`conan install`** for debug in the `cmake-build-debug` folder to install your requirements and generate the `conanbuildinfo.cmake` file there:

```
$ conan install . -s build_type=Debug --install-folder=cmake-build-debug
```

6. Repeat the last step if you have the release build types configured in your CLion IDE, but change the build_type setting accordingly:

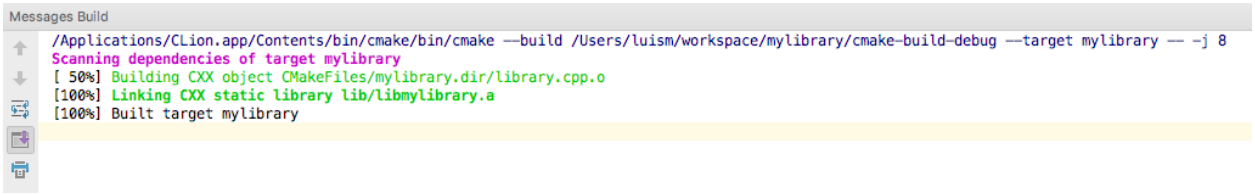
```
$ conan install . -s build_type=Release --install-folder=cmake-build-release
```

7. Now reconfigure your CLion project. The Warning message is not shown anymore:



8. Open the `library.cpp` file and include `zlib.h`. If you follow the link, you can see that CLion automatically detects the `zlib.h` header file from the local Conan cache.

9. Build your project normally using your CLion IDE:

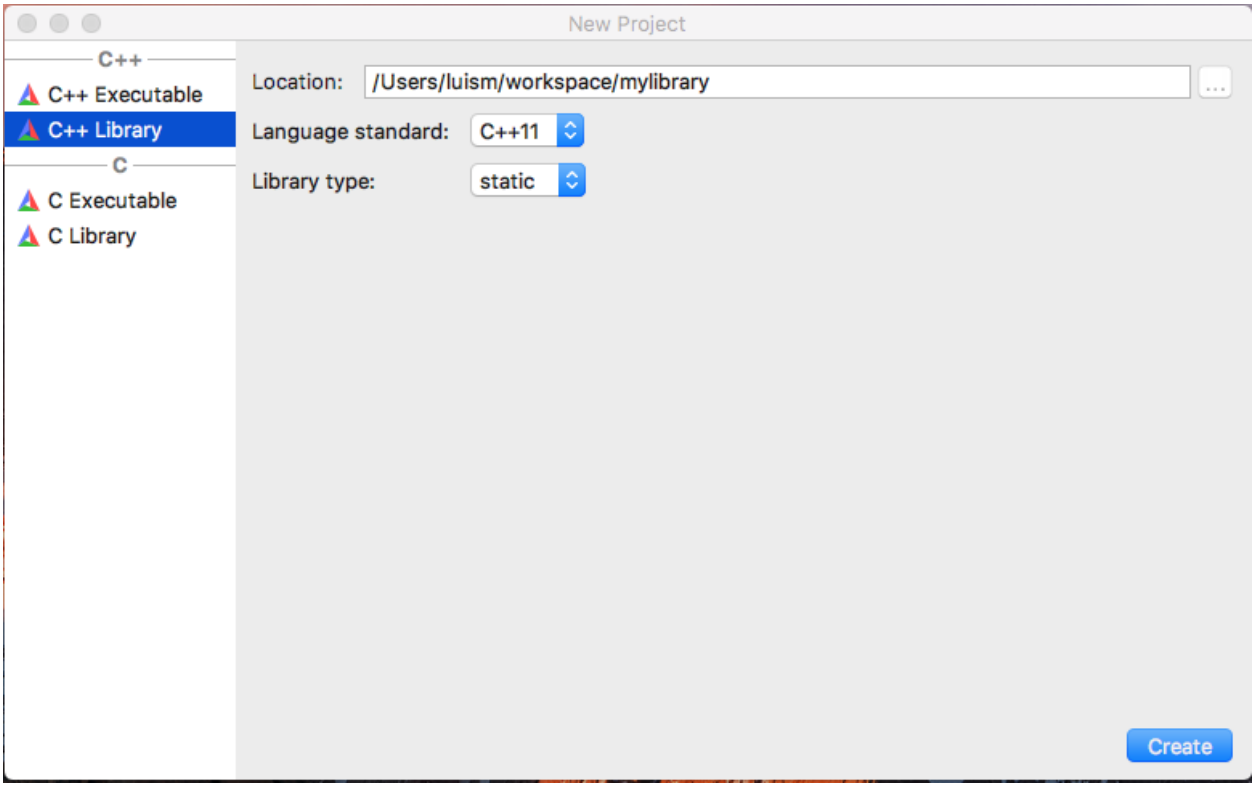


You can check a complete example of a CLion project reusing conan packages in this github repository: lasote/clion-conan-consumer.

Creating Conan packages in a CLion project

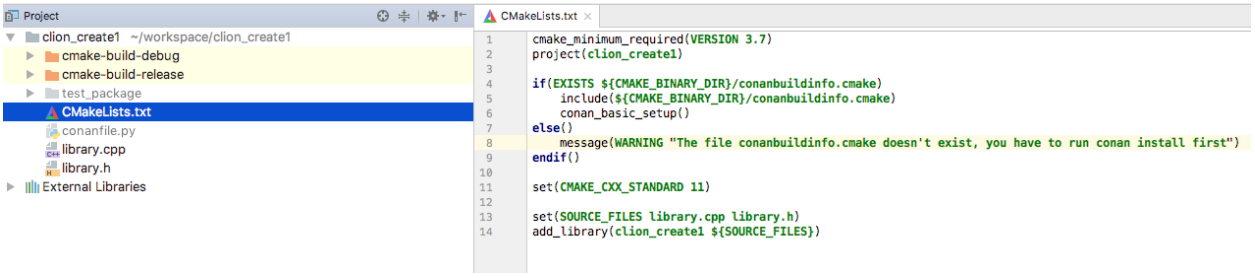
Now we are going to see how to create a Conan package from the previous library.

1. Create a new CLion project



2. Edit the `CMakeLists.txt` file and add the following lines:

```
if(EXISTS ${CMAKE_BINARY_DIR}/conanbuildinfo.cmake)
    include(${CMAKE_BINARY_DIR}/conanbuildinfo.cmake)
    conan_basic_setup()
else()
    message(WARNING "The file conanbuildinfo.cmake doesn't exist, you have to run conan␣
↪install first")
endif()
```



3. Create a `conanfile.py` file. It's recommended to use the **`conan new`** command.

```
$ conan new mylibrary/1.0@myuser/channel
```

Edit the `conanfile.py`:

- We are removing the `source` method because we have the sources in the same project; so we can use the `exports_sources`.
- In the `package_info` method, adjust the library name. In this case our `CMakeLists.txt` creates a target library called `mylibrary`.

- Adjust the CMake helper in the `build()` method. The `cmake.configure()` doesn't need to specify the `source_folder`, because we have the `library.*` files in the root directory.
- Adjust the `copy` function calls in the `package` method to ensure that all your headers and libraries are copied to the Conan package.

```
from conans import ConanFile, CMake, tools


class MylibraryConan(ConanFile):
    name = "mylibrary"
    version = "1.0"
    license = "<Put the package license here>"
    url = "<Package recipe repository url here, for issues about the package>"
    description = "<Description of Mylibrary here>"
    settings = "os", "compiler", "build_type", "arch"
    options = {"shared": [True, False]}
    default_options = {"shared": False}
    generators = "cmake"
    requires = "zlib/1.2.11"

    def build(self):
        cmake = CMake(self)
        cmake.configure()
        cmake.build()

        # Explicit way:
        # self.run('cmake "%s" %s' % (self.source_folder, cmake.command_line))
        # self.run("cmake --build . %s" % cmake.build_config)

    def package(self):
        self.copy("*.h", dst="include", src="hello")
        self.copy("*.lib", dst="lib", keep_path=False)
        self.copy("*.dll", dst="bin", keep_path=False)
        self.copy("*.so", dst="lib", keep_path=False)
        self.copy("*.dylib", dst="lib", keep_path=False)
        self.copy("*.a", dst="lib", keep_path=False)

    def package_info(self):
        self.cpp_info.libs = ["mylibrary"]
```

4. To build your library with CLion, follow the guide of *Using packages from step 5*.

5. To package your library, use the **conan export-pkg** command passing the used build-folder. It will call your `package()` method to extract the artifacts and push the Conan package to the local cache:

```
$ conan export-pkg . mylibrary/1.0@myuser/channel --build-folder cmake-build-debug -
↪pr=myprofile
```

7. Now you can upload it to a Conan server if needed:

```
$ conan upload mylibrary/1.0@myuser/channel # This will upload only the recipe, use --
↪all to upload all the generated binary packages.
```

8. If you would like to see how the package looks like before exporting it to the local cache (**conan export-pkg**) you can use the **conan package** command to create the package in a local directory:

```
$ conan package . --build-folder cmake-build-debug --package-folder=mypackage
```

If we list the `mypackage` folder we can see:

- A `lib` folder containing our library
- A `include` folder containing our header files
- A `conaninfo.txt` and `conanmanifest.txt` conan files, always present in all packages.

You can check a full example of a CLion project for creating a Conan package in this github repository: lasote/clion-conan-package.

15.3.3 Apple/Xcode

Conan can be integrated with **Apple's XCode** in two different ways:

- Using the **cmake** generator to create a **conanbuildinfo.cmake** file.
- Using the **xcode** generator to create a **conanbuildinfo.xcconfig** file.

With CMake

Check the *Integrations/cmake* section to read about the **cmake** generator. Check the official CMake docs to find out more about generating Xcode projects with CMake.

With the *xcode* generator

You can use the **xcode** generator to integrate your requirements with your *Xcode* project. This generator creates an `xcconfig` file, with all the *include paths*, *lib paths*, *libs*, *flags* etc, that can be imported in your project.

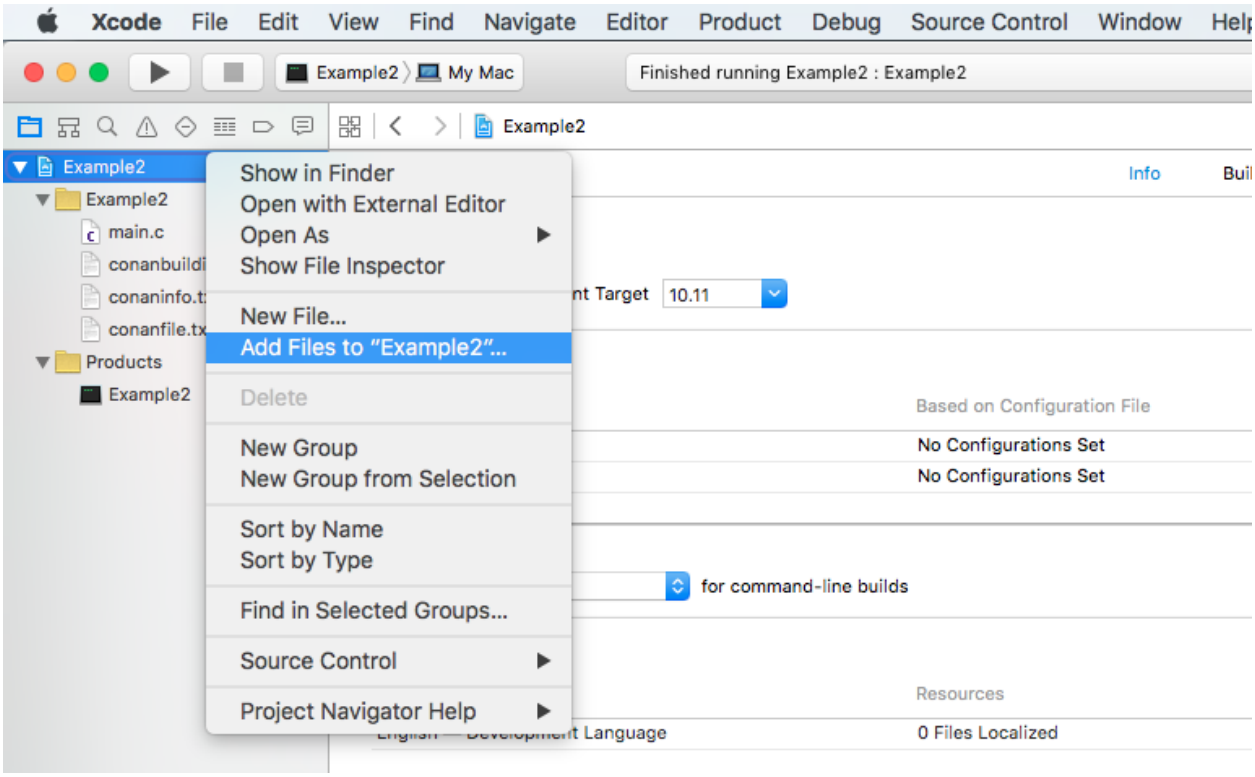
Open `conanfile.txt` and change (or add) the **xcode** generator:

```
[requires]
poco/1.9.4

[generators]
xcode
```

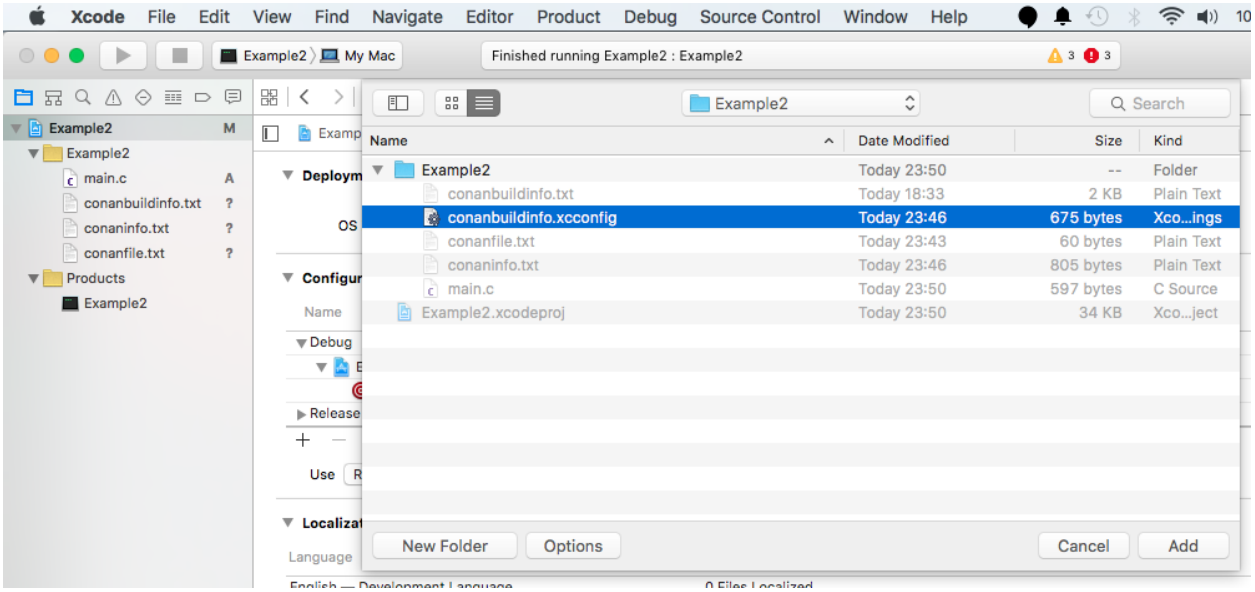
Install the requirements:

```
$ conan install .
```

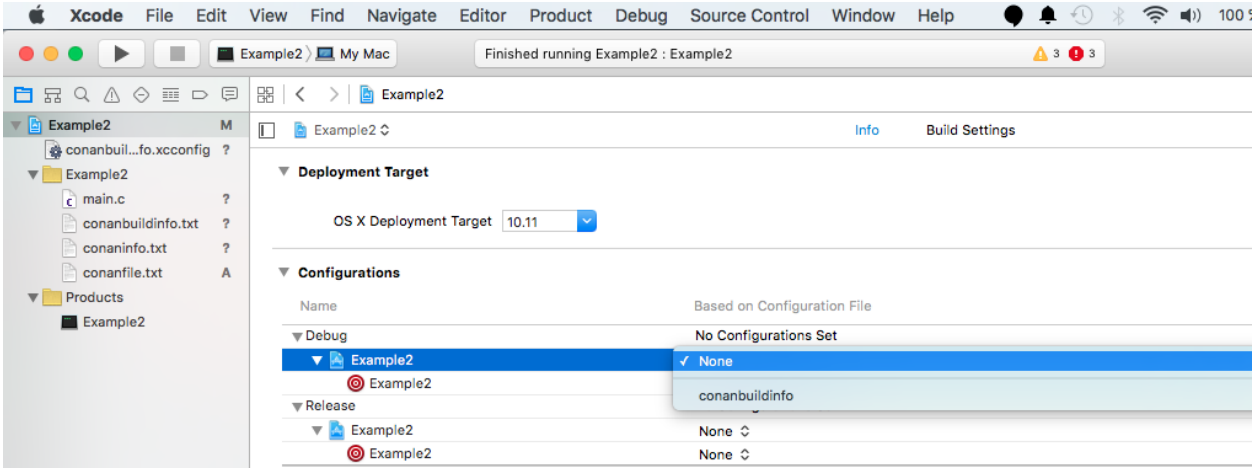
Go to your **Xcode** project, click on the project and select **Add files to**...



Choose `conanbuildinfo.xcconfig` generated.



Click on the project again. In the **info/configurations** section, choose **conanbuildinfo** for *release* and *debug*.



Build your project as usual.

See also:

Check the *Reference/Generators/xcode* for the complete reference.

See also:

Check the *Tools section about Apple tools* to ease the integration with the Apple development tools in your recipes using the toolchain as a *tool require*.

See also:

Check the *Darwin Toolchain package* section to learn how to **cross build** for `iOS`, `watchOS` and `tvOS`.

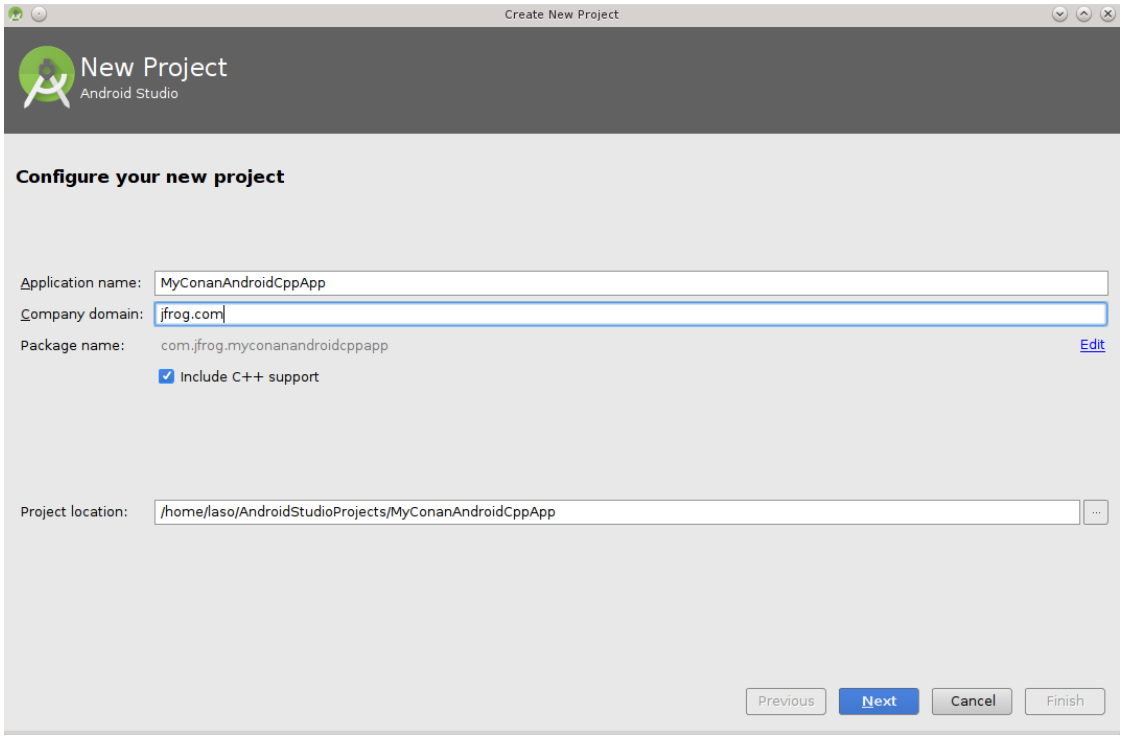
15.3.4 Android Studio

You can use Conan to *cross-build your libraries for Android* with different architectures. If you are using Android Studio for your Android application development, you can integrate Conan to automate the library building for the different architectures that you want to support in your project.

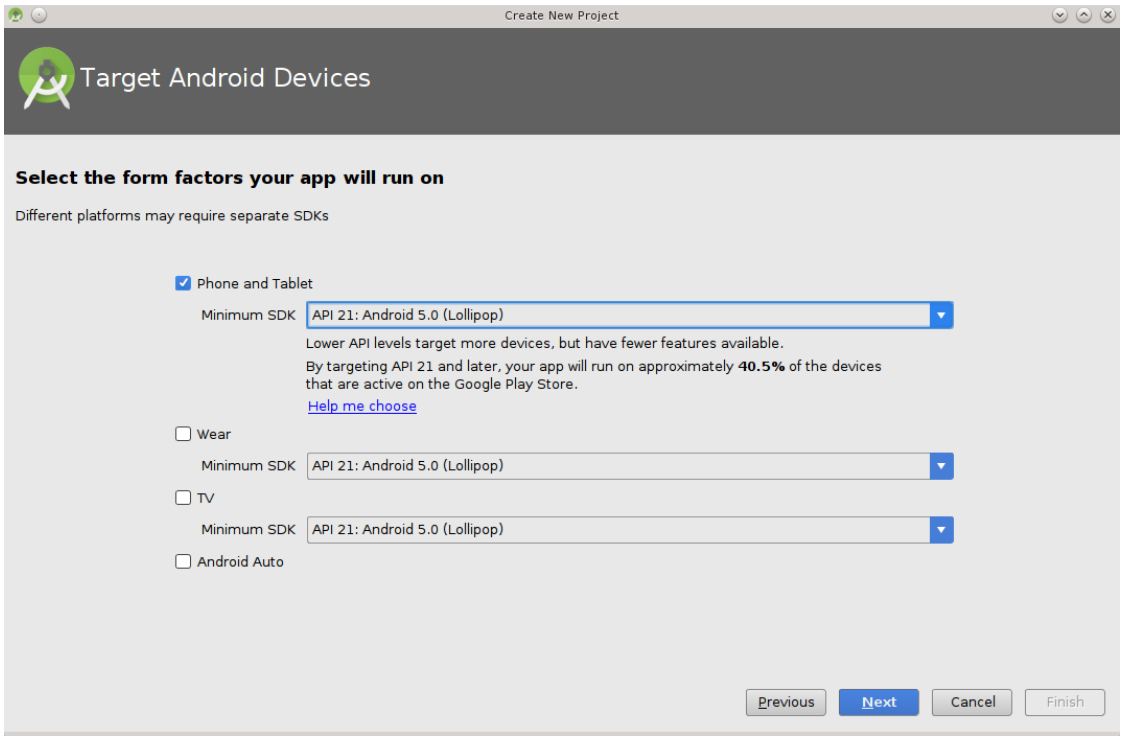
Here is an example of how to integrate the `libpng` Conan package library in an Android application, but any library that can be cross-compiled to Android could be used using the same procedure.

We are going to start from the "Hello World" wizard application and then will add it the `libpng` C library:

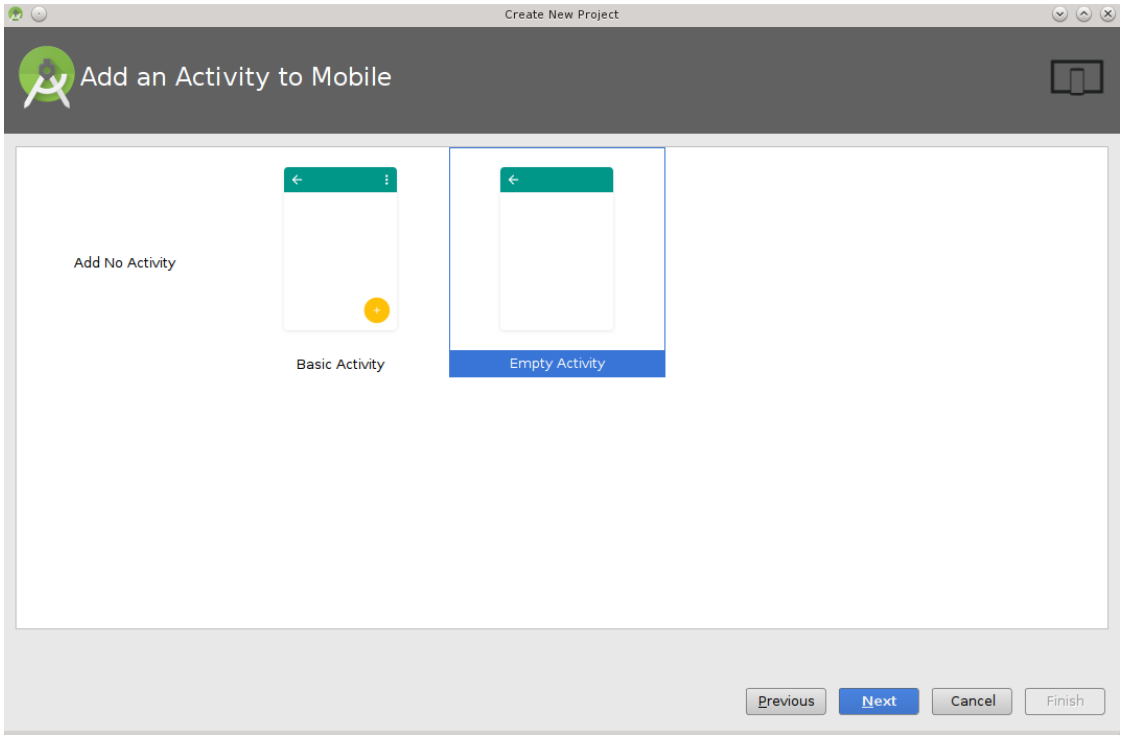
1. Follow the *cross-build your libraries for Android* guide to create a standalone toolchain and create a profile named `android_21_arm_clang` for Android. You can also use the NDK that the Android Studio installs.
2. Create a new Android Studio project and include C++ support.

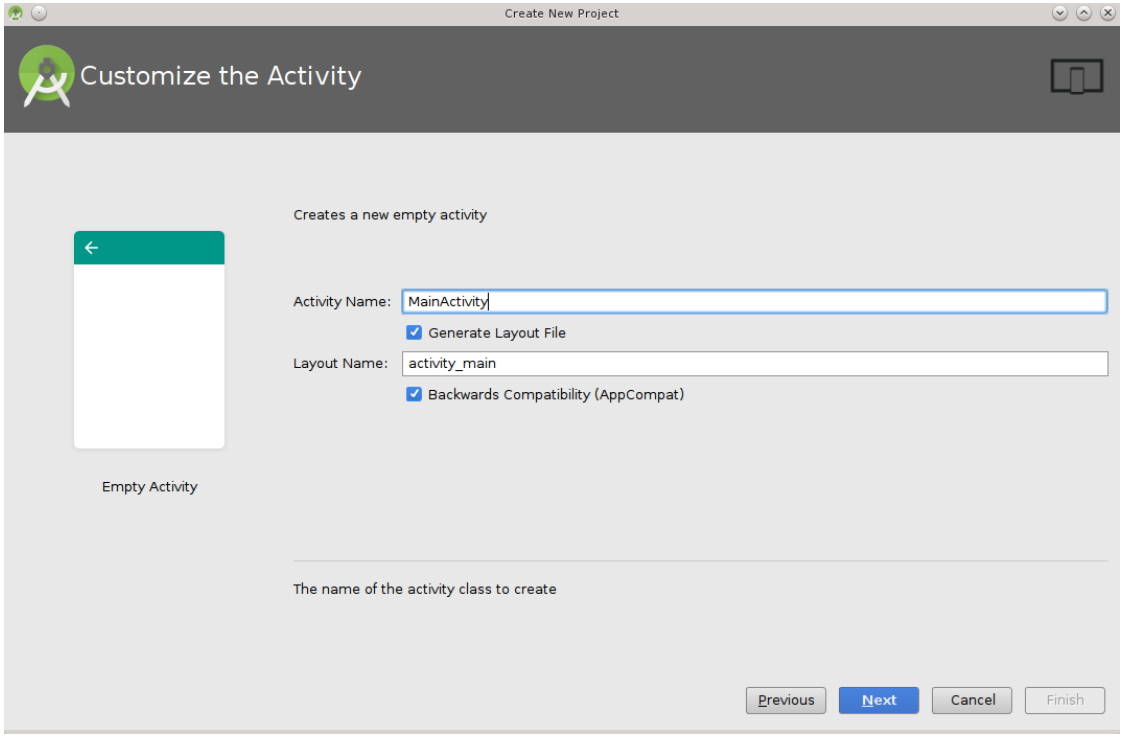


3. Select your API level and target. The arch and api level have to match with the standalone toolchain created in step 1.

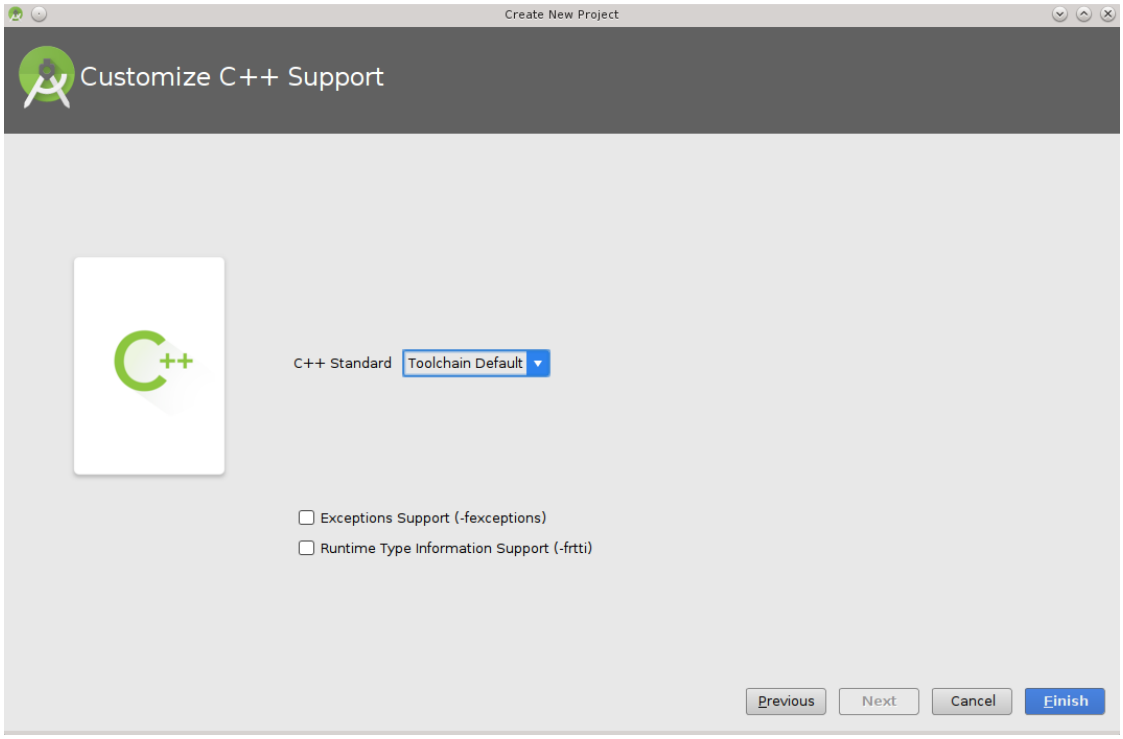


4. Add an empty Activity and name it.





5. Select the C++ standard



6. Change to the *project view* and in the *app* folder create a `conanfile.txt` with the following contents:

conanfile.txt

```
[requires]
libpng/1.6.23@lasote/stable

[generators]
cmake
```

7. Open the `CMakeLists.txt` file from the app folder and replace the contents with:

```
cmake_minimum_required(VERSION 3.4.1)

include(${CMAKE_CURRENT_SOURCE_DIR}/conan_build/conanbuildinfo.cmake)
set(CMAKE_CXX_COMPILER_VERSION "5.0") # Unknown miss-detection of the compiler by CMake
conan_basic_setup(TARGETS)

add_library(native-lib SHARED src/main/cpp/native-lib.cpp)
target_link_libraries(native-lib CONAN_PKG::libpng)
```

8. Open the *app/build.gradle* file. We are configuring the architectures we want to build, specifying adding a new task `conanInstall` that will call **`conan install`** to install the requirements:

- In the defaultConfig section, append:

```
ndk {
   // Specifies the ABI configurations of your native
   // libraries Gradle should build and package with your APK.
   abiFilters 'armeabi-v7a'
}
```

- After the android block:

```
task conanInstall {
    def buildDir = new File("app/conan_build")
    buildDir.mkdirs()
    // if you have problems running the command try to specify the absolute
    // path to conan (Known problem in MacOSX) /usr/local/bin/conan
    def cmmd = "conan install ../conanfile.txt --profile android_21_arm_clang --build␣
↪missing "
    print(">> ${cmmd} \n")

    def sout = new StringBuilder(), serr = new StringBuilder()
    def proc = cmmd.execute(null, buildDir)
    proc.consumeProcessOutput(sout, serr)
    proc.waitFor()
    println "$sout $serr"
    if(proc.exitValue() != 0){
        throw new Exception("out> $sout err> $serr" + "\nCommand: ${cmmd}")
    }
}
```

9. Finally open the default example cpp library in `app/src/main/cpp/native-lib.cpp` and include some lines using your library. Be careful with the JNICALL name if you used another app name in the wizard:

```
#include <jni.h>
#include <string>
#include "png.h"
#include "zlib.h"
#include <sstream>
#include <iostream>

extern "C"
JNIEXPORT jstring JNICALL
Java_com_jfrog_myconanandroidcppapp_MainActivity_stringFromJNI(
        JNIEnv *env,
        jobject /* this */) {
   std::ostringstream oss;
   oss << "Compiled with libpng: " << PNG_LIBPNG_VER_STRING << std::endl;
   oss << "Running with libpng: " << png_libpng_ver << std::endl;
   oss << "Compiled with zlib: " << ZLIB_VERSION << std::endl;
   oss << "Running with zlib: " << zlib_version << std::endl;

   return env->NewStringUTF(oss.str().c_str());
}
```

Build your project normally. Conan will create a `conan` folder with a folder for each different architecture you have specified in the abiFilters with a `conanbuildinfo.cmake` file.

Then run the app using an x86 emulator for best performance:

See also:

Check the section *Linux/Windows/macOS to Android* to read more about cross-building for Android.

15.3.5 YouCompleteMe (vim)

If you are a vim user, you may also be a user of YouCompleteMe.

With this generator, you can create the necessary files for your project dependencies, so YouCompleteMe will show symbols from your Conan installed dependencies for your project. You only have to add the `ycm` generator to your `conanfile`:

Listing 22: *conanfile.txt*

```
[generators]
ycm
```

It will generate a *conan_ycm_extra_conf.py* and a *conan_ycm_flags.json* file in your folder. Those files will be overwritten each time you run **`conan install`**.

In order to make YouCompleteMe work, copy/move *conan_ycm_extra_conf.py* to your project base folder (usually the one containing your `conanfile`) and rename it to *.ycm_extra_conf.py*.

You can (and probably should) edit this file to add your project specific configuration. If your base folder is different from your build folder, link the *conan_ycm_flags.json* from your build folder to your base folder.

```
# from your base folder
$ cp build/conan_ycm_extra_conf.py .ycm_extra_conf.py
$ ln -s build/conan_ycm_flags.json conan_ycm_flags.json
```

15.4 CI Platforms

You can use any CI platform to build your libraries and generate your Conan packages.

15.4.1 Jenkins

You can use *Jenkins CI* both for:

- Building and testing your project, which manages dependencies with Conan, and probably a conanfile.txt file
- Building and testing conan binary packages for a given Conan package recipe (with a conanfile.py) and uploading to a Conan remote (Artifactory or conan_server)

There is no need for any special setup for it, just install Conan and your build tools in the Jenkins machine and call the needed Conan commands.

Note: As reported in https://github.com/conan-io/conan/issues/6400, running Conan under Jenkins could have some unexpected issues running `git clone` of repositories requiring authentication. If that is the case, consider to use `ssh` protocol instead of `https`.

Artifactory and Jenkins integration

If you are using Artifactory you can take advantage of the Jenkins Artifactory Plugin. Check here how to install the plugin and here you can check the full documentation about the DSL.

The Artifactory Jenkins plugin provides a powerful DSL (Domain Specific Language) to call Conan, connect with your Artifactory instance, upload and download your packages from Artifactory and manage your build information.

Example: Test your project getting requirements from Artifactory

This is a template to use Jenkins with an Artifactory plugin and Conan to retrieve your package from Artifactory server and publish the build information about the downloaded packages to Artifactory.

In this script we assume that we already have all our dependencies in the Artifactory server, and we are building our project that uses **Boost** and **Poco** libraries.

Create a new Jenkins Pipeline task using this script:

```
//Adjust your artifactory instance name/repository and your source code repository
def artifactory_name = "artifactory"
def artifactory_repo = "conan-local"
def repo_url = 'https://github.com/memsharded/example-boost-poco.git'
def repo_branch = 'master'

node {
    def server = Artifactory.server artifactory_name
```

(continues on next page)

(continued from previous page)

```
    def client = Artifactory.newConanClient()

    stage("Get project"){
        git branch: repo_branch, url: repo_url
    }

    stage("Get dependencies and publish build info"){
        sh "mkdir -p build"
        dir ('build') {
          def b = client.run(command: "install ..")
          server.publishBuildInfo b
        }
    }

    stage("Build/Test project"){
        dir ('build') {
          sh "cmake ../ && cmake --build ."
        }
    }
}
```

Example: Build a Conan package and upload it to Artifactory

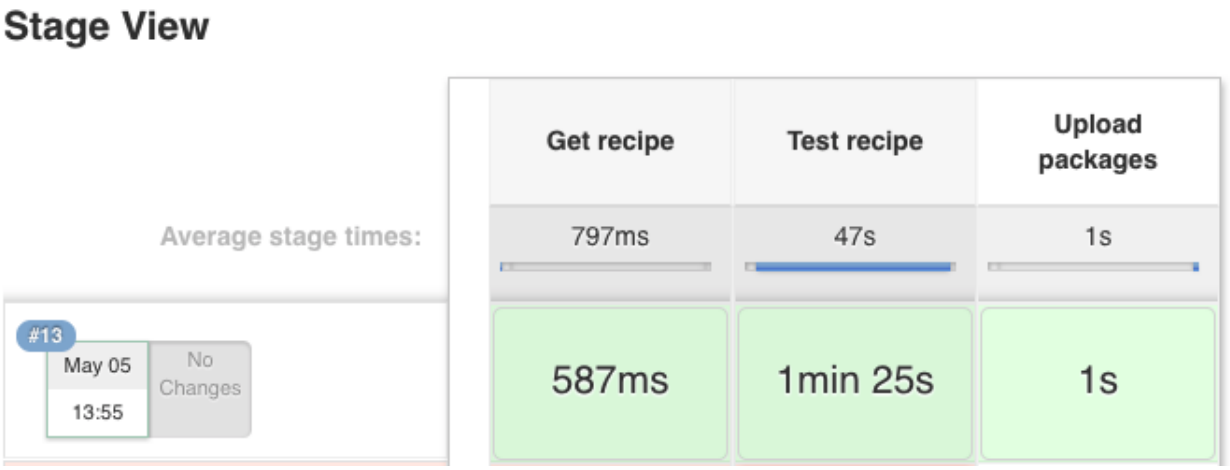
In this example we will call Conan `create` command to create a binary packages and then upload it to Artifactory. We also upload the build information:

```
def artifactory_name = "artifactory"
def artifactory_repo = "conan-local"
def repo_url = 'https://github.com/conan-io/conan-center-index.git'
def repo_branch = "master"
def recipe_folder = "recipes/zlib/1.2.11"
def recipe_version = "1.2.11"

node {
    def server = Artifactory.server artifactory_name
    def client = Artifactory.newConanClient()
    def serverName = client.remote.add server: server, repo: artifactory_repo

    stage("Get recipe"){
```

(continues on next page)

(continued from previous page)

```
        git branch: repo_branch, url: repo_url
    }

    stage("Test recipe"){
        dir (recipe_folder) {
          client.run(command: "create . ${recipe_version}@")
        }
    }

    stage("Upload packages"){
        String command = "upload \"*\" --all -r ${serverName} --confirm"
        def b = client.run(command: command)
        server.publishBuildInfo b
    }
}
```



15.4.2 Travis CI

You can use the Travis CI cloud service to automatically build and test your project in Linux/MacOS environments in the cloud. It is free for OSS projects, and offers an easy integration with GitHub, so builds can be automatically fired in Travis-CI after a **`git push`** to GitHub.

You can use Travis-CI both for:

- Building and testing your project, which manages dependencies with Conan, and probably a *conanfile.txt* file.
- Building and testing Conan binary packages for a given Conan package recipe (with a *conanfile.py*).

Installing dependencies and building your project

A very common use case is to build your project after Conan takes care of installing your dependencies. Doing this process in Travis CI is quite convenient as you can do it with **conan install**.

To enable **Travis CI** support, you need to create a *.travis.yml* file and paste this code in it:

```
os: linux
language: python
python: "3.7"
dist: xenial
compiler:
  - gcc
install:
# Install conan
  - pip install conan
# Automatic detection of your arch, compiler, etc.
  - conan user
script:
# Download dependencies and build project
  - conan install .
# Call your build system
  - cmake . -G "Unix Makefiles"
  - cmake --build .
# Run your tests
  - ctest .
```

Travis will install the gcc compiler and the **conan** client and will execute the **conan install** command using the requirements and generators indicated in your *conanfile.py* or *conanfile.txt*. Then, the **script** section installs the requirements and then you can use your build system to compile the project (using **make** in this example).

Creating, testing and uploading Conan binary packages

You can also use Travis CI to automate building new Conan binary packages with every change you push to GitHub. You can probably set up your own way, but Conan has some utilities to help in the process.

The command **conan new** has arguments to create a default working *.travis.yml* file. Other setups might be possible, but for this example we are assuming that you are using GitHub and also uploading your final packages to your Artifactory CE server.

You could follow these steps:

1. First, create an empty GitHub repository. Let's call it "hello", for creating a "hello world" package. GitHub allows creating it with a Readme and .gitignore.
2. Create a Conan repository in your Artifactory CE, and get its URL ("Set me up"). We will call it `UPLOAD_URL`
3. Activate the repo in your Travis account, so it is built when we push changes to it.
4. Under *Travis More Options -> Settings->Environment Variables*, add the `CONAN_LOGIN_USERNAME` and `CONAN_PASSWORD` environment variables with the Artifactory CE user and password.
5. Clone the repo: **git clone <your_repo/hello> && cd hello**.
6. Create the package: **conan new hello/0.1@myteam/testing -t -s -cilg -cis -ciu=UPLOAD_URL**

7. You can inspect the created files: both *.travis.yml*, *.travis/run.sh*, and `.travis/install.sh` and the *build.py* script, that is used by **conan-package-tools** utility to split different builds with different configurations in different Travis CI jobs.
8. You can test locally, before pushing, with **`conan test`**.
9. Add the changes, commit and push: **`git add . && git commit -m "first commit" && git push`**.
10. Go to Travis and see the build, with the different jobs.
11. When it has finished, go to your Artifactory CE repository, you should see there the uploaded packages for different configurations.
12. Check locally, searching in Artifactory CE: **`conan search hello/0.1@myteam/testing -r=myremote`**.

If something fails, please report an issue in the `conan-package-tools` GitHub repository: https://github.com/conan-io/conan-package-tools

15.4.3 Appveyor

You can use the AppVeyor cloud service to automatically build and test your project in a Windows environment in the cloud. It is free for OSS projects, and offers an easy integration with Github, so builds can be automatically fired in Appveyor after a **`git push`** to Github.

You can use Appveyor both for:

- Building and testing your project, which manages dependencies with Conan, and probably a conanfile.txt file
- Building and testing Conan binary packages for a given Conan package recipe (with a conanfile.py)

Building and testing your project

We are going to use an example with GTest package, with **AppVeyor** support to run the tests.

Clone the project from github:

```
$ git clone https://github.com/lasote/conan-gtest-example
```

Create an `appveyor.yml` file and paste this code in it:

```
version: 1.0.{build}
    platform:
      - x64

    install:
      - cmd: echo "Downloading conan..."
      - cmd: set PATH=%PATH%;%PYTHON%/Scripts/
      - cmd: pip.exe install conan
      - cmd: conan user # Create the conan data directory
      - cmd: conan --version
```

(continues on next page)

(continued from previous page)

```
build_script:
  - cmd: mkdir build
  - cmd: conan install . -o gtest:shared=True
  - cmd: cd build
  - cmd: cmake ../ -DBUILD_TEST=TRUE  -G "Visual Studio 14 2015 Win64"
  - cmd: cmake --build . --config Release

test_script:
  - cmd: cd bin
  - cmd: encryption_test.exe
```

Appveyor will install the **Conan** tool and will execute the **conan install** command. Then, the **build_script** section creates the build folder, compiles the project with **cmake** and the section **test_script** runs the **tests**.

Creating, testing and uploading Conan binary packages

You can use Appveyor to automate the building of binary packages, which will be created in the cloud after pushing to Github. You can probably set up your own way, but Conan has some utilities to help in the process.

The command **conan new** has arguments to create a default working *appveyor.yml* file. Other setups might be possible, but for this example we are assuming that you are using GitHub and also uploading your final packages to your own free Artifactory CE repository. You could follow these steps:

1. First, create an empty github repository. Let's call it "hello", for creating a "hello world" package. Github allows to create it with a Readme and .gitignore.
2. Create a Conan repository in your Artifactory CE, and get its URL ("Set me up"). We will call it `UPLOAD_URL`
3. Activate the repo in your Appveyor account, so it is built when we push changes to it.
4. Under *Appveyor Settings->Environment*, add the `CONAN_LOGIN_USERNAME` and `CONAN_PASSWORD` environment variables with the Artifactory CE user and password.
5. Clone the repo: `$ git clone <your_repo/hello> && cd hello`
6. Create the package: **conan new hello/0.1@myteam/testing -t -s -ciw -cis -ciu=UPLOAD_URL**
7. You can inspect the created files: both *appveyor.yml* and the *build.py* script, that is used by **conan-package-tools** utility to split different builds with different configurations in different appveyor jobs.
8. You can test locally, before pushing, with **conan create**
9. Add the changes, commit and push: **git add . && git commit -m "first commit" && git push**
10. Go to Appveyor and see the build, with the different jobs.
11. When it finish, go to your Artifactory CE repository, you should see there the uploaded packages for different configurations
12. Check locally, searching in Artifactory CE: **conan search hello/0.1@myteam/testing -r=myrepo**

If something fails, please report an issue in the `conan-package-tools` github repository: https://github.com/conan-io/conan-package-tools

15.4.4 Gitlab

You can use the Gitlab CI cloud or local service to automatically build and test your project in Linux/MacOS/Windows environments. It is free for OSS projects, and offers an easy integration with Gitlab, so builds can be automatically fired in Gitlab CI after a **`git push`** to Gitlab.

You can use Gitlab CI both for:

- Building and testing your project, which manages dependencies with Conan, and probably a conanfile.txt file
- Building and testing Conan binary packages for a given Conan package recipe (with a conanfile.py)

Building and testing your project

We are going to use an example with GTest package, with **Gitlab CI** support to run the tests.

Clone the project from github:

```
$ git clone https://github.com/lasote/conan-gtest-example
```

Create a `.gitlab-ci.yml` file and paste this code in it:

```
image: conanio/gcc63

build:
  before_script:
    # Upgrade Conan version
    - sudo pip install --upgrade conan
    # Automatic detection of your arch, compiler, etc.
    - conan user

  script:
    # Download dependencies, build, test and create package
    - conan create . user/channel
```

Gitlab CI will install the **conan** tool and will execute the **conan install** command. Then, the **script** section creates the build folder, compiles the project with **cmake** and runs the **tests**.

On Windows the Gitlab runner may be running as a service and not have a home directory, in which case you need to set a custom value for `CONAN_USER_HOME`.

Creating, testing and uploading Conan binary packages

You can use Gitlab CI to automate the building of binary packages, which will be created in the cloud after pushing to Gitlab. You can probably setup your own way, but Conan has some utilities to help in the process. The command **conan new** has arguments to create a default working `.gitlab-ci.yml` file. Other setups might be possible, but for this example we are assuming that you are using github and also uploading your final packages to your own Artifactory CE server. You could follow these steps:

1. First, create an empty gitlab repository, let's call it "hello", for creating a "hello world" package. Gitlab allows to create it with a Readme, license and .gitignore.
2. Create a Conan repository in Artifactory CE, and get its URL ("Set me up"). We will call it `UPLOAD_URL`
3. Under your project page, *Settings -> Pipelines -> Add a variable*, add the `CONAN_LOGIN_USERNAME` and `CONAN_PASSWORD` environment variables with the Artifactory CE user and password.
4. Clone the repo: **git clone <your_repo/hello> && cd hello**.
5. Create the package: **conan new hello/0.1@myteam/testing -t -s -ciglg -ciglc -cis -ciu=UPLOAD_URL**
6. You can inspect the created files: both *.gitlab-ci.yml* and the *build.py* script, that is used by **conan-package-tools** utility to split different builds with different configurations in different GitLab CI jobs.
7. You can test locally, before pushing, with **conan create** or by GitLab Runner.
8. Add the changes, commit and push: **git add . && git commit -m "first commit" && git push**.
9. Go to Pipelines page and see the pipeline, with the different jobs.
10. When it has finished, go to your Artifactory CE repository, you should see there the uploaded packages for different configurations.
11. Check locally, searching in Artifactory CE: **conan search hello/0.1@myteam/testing -r=myremote**.

If something fails, please report an issue in the **conan-package-tools** github repository: https://github.com/conan-io/conan-package-tools

15.4.5 Circle CI

You can use the Circle CI cloud to automatically build and test your project in Linux/MacOS environments. It is free for OSS projects, and offers an easy integration with Github, so builds can be automatically fired in CircleCI after a `git push` to Github.

You can use CircleCI both for:

- Building and testing your project, which manages dependencies with Conan, and probably a conanfile.txt file
- Building and testing Conan binary packages for a given Conan package recipe (with a conanfile.py)

Building and testing your project

We are going to use an example with GTest package, with **CircleCI** support to run the tests.

Clone the project from github:

```
$ git clone https://github.com/lasote/conan-gtest-example
```

Create a `.circleci/config.yml` file and paste this code in it:

```
version: 2
gcc-6:
  docker:
    - image: conanio/gcc6
  steps:
    - checkout
    - run:
        name: Build Conan package
        command: |
          sudo pip install --upgrade conan
          conan user
          conan create . user/channel
workflows:
  version: 2
  build_and_test:
    jobs:
    - gcc-6
```

CircleCI will install the **Conan** tool and will execute the **conan create** command. Then, the **script** section creates the build folder, compiles the project with **cmake** and runs the **tests**.

Creating, testing and uploading Conan package binaries

You can use CircleCI to automate the building of binary packages, which will be created in the cloud after pushing to Github. You can probably set up your own way, but Conan has some utilities to help in the process.

The command `conan new` has arguments to create a default working `.circleci/config.yml` file. Other setups might be possible, but for this example we are assuming that you are using github and also uploading your final packages to your own Artifactory CE server. You could follow these steps:

1. First, create an empty Github repository (let's call it "hello") for creating a "hello world" package. Github allows to create it with a Readme, license and .gitignore.
2. Create a Conan repository in your Artifactory CE and get its URL ("Set me up"). We will call it `UPLOAD_URL`
3. Under your project page, *Settings -> Pipelines -> Add a variable*, add the `CONAN_LOGIN_USERNAME` and `CONAN_PASSWORD` environment variables with the Artifactory CE user and password.
4. Clone the repo: `$ git clone <your_repo/hello> && cd hello`
5. Create the package: `$ conan new hello/0.1@myteam/testing -t -s -ciccg -ciccc -cicco -cis -ciu=UPLOAD_URL`
6. You can inspect the created files: both `.circleci/config.yml` and the `build.py` script, that is used by `conan-package-tools` utility to split different builds with different configurations in different GitLab CI jobs.
7. You can test locally, before pushing, with `$ conan create`
8. Add the changes, commit and push: `$ git add . && git commit -m "first commit" && git push`

9. Go to Pipelines page and see the pipeline, with the different jobs.
10. When it has finished, go to your Artifactory CE repository, you should see there the uploaded packages for different configurations
11. Check locally, searching in Artifactory CE: `$ conan search hello/0.1@myteam/testing -r=myremote`

If something fails, please report an issue in the `conan-package-tools` github repository: https://github.com/conan-io/conan-package-tools

15.4.6  Microsoft's Azure DevOps (TFS, VSTS)

Thanks to the JFrog Artifactory Extension for Azure DevOps and TFS it is possible to support Conan tasks and integrate it with the CI development platform provided by Microsoft's Azure DevOps and the Artifactory binary repository manager.

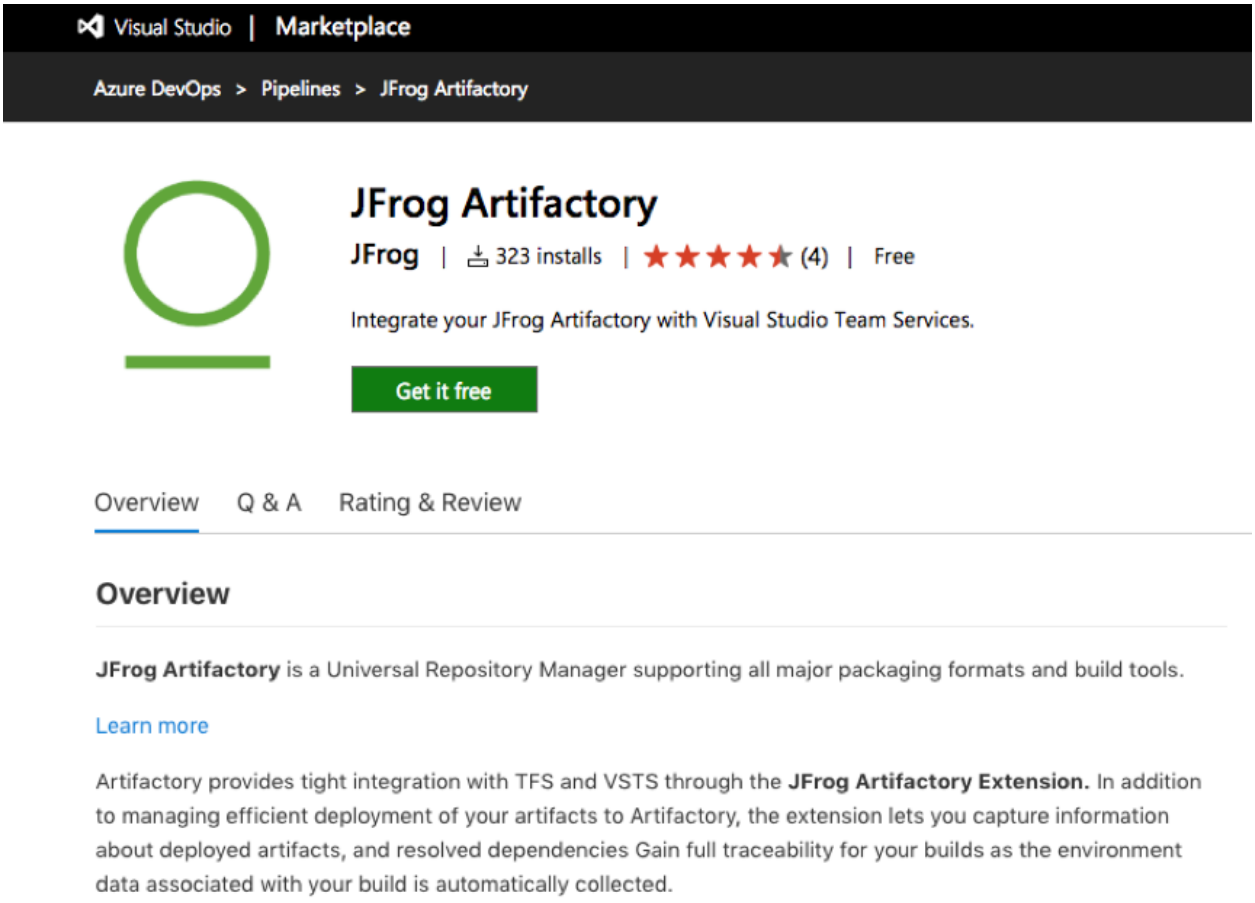
The support for Conan now in the JFrog Artifactory Extension helps you perform the following tasks in Azure DevOps or TFS:

- Run Conan commands
- Resolve Conan dependencies from remote Artifactory servers
- Push Conan packages to Artifactory
- Publish BuildInfo metadata
- Import a Conan configuration

In this section we will show you how to add Conan tasks to your pipelines using the Artifactory/Conan Extension and push the generated buildinfo metadata to Artifactory where it can be used to track and automate your builds.

Configuring DevOps Azure to use Artifactory with Conan

To use the Conan support provided by the JFrog Artifactory Extension you must configure a self-hosted agent that will enable Conan builds for your Azure Pipelines environment. Afterwards you can install the JFrog Artifactory Extension from the Visual Studio Marketplace and follow the installation instructions in the Overview.



When completed, proceed to create builds and access buildinfo from within Azure DevOps or TFS.

Steps to follow

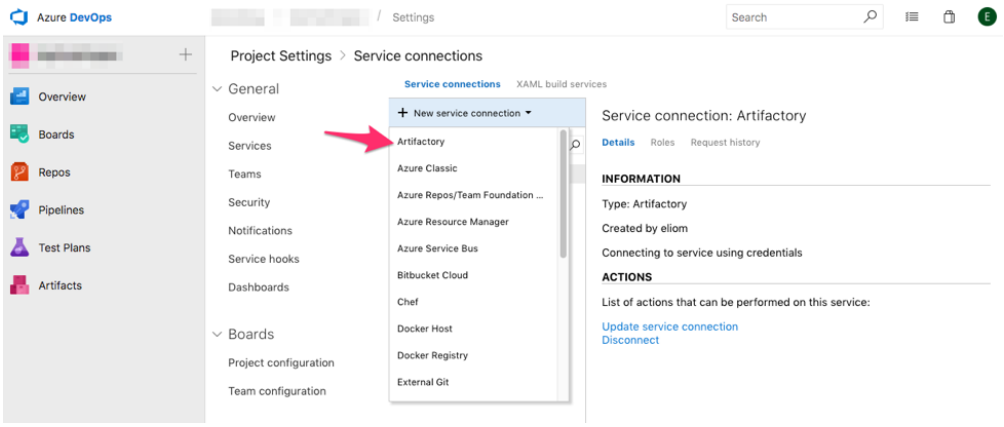
In these steps, you will set up Azure DevOps to use Artifactory and add Conan tasks to your build pipeline. Then you can set up to push the buildinfo from the Conan task to Artifactory.

STEP 1: Configure the Artifactory instance

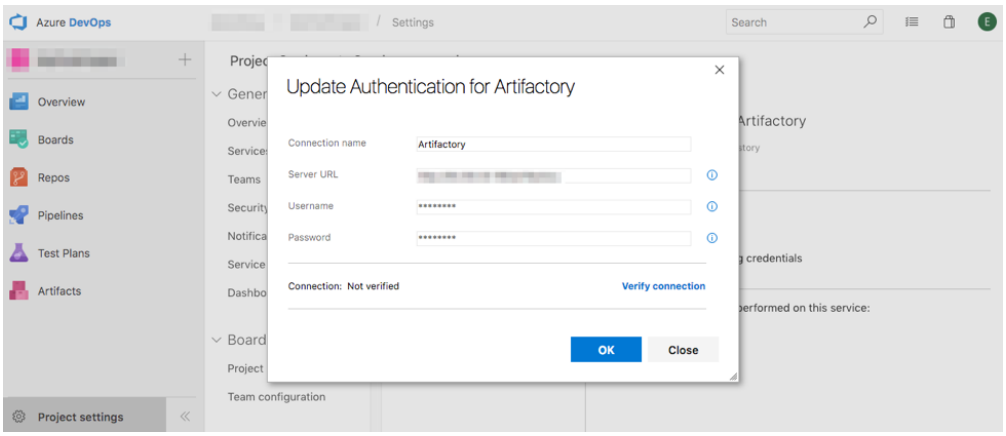
Once the Artifactory Extension is installed, you must configure Azure DevOps to access the Artifactory instance.

To add Artifactory to Azure DevOps:

1. In Azure DevOps, go to Project Settings > Service connections.
2. Click + New service connection to display the list control, and select Artifactory.



3. In the resulting Update Authentication for Artifactory dialog, enter the required server and credential information, and click OK.

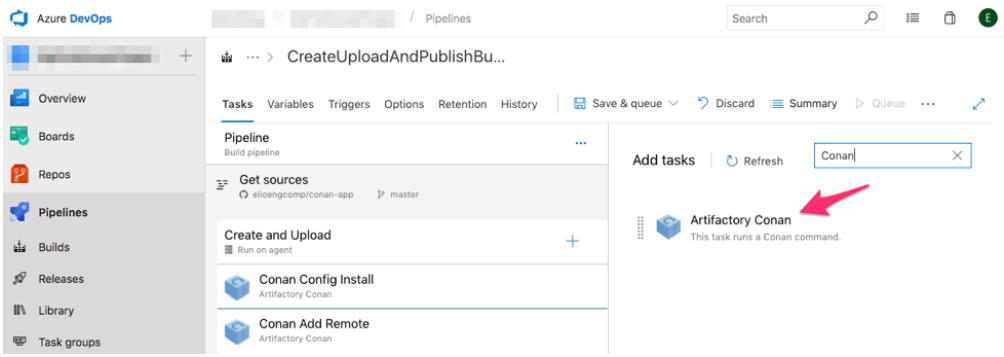


STEP 2: Add a Conan task

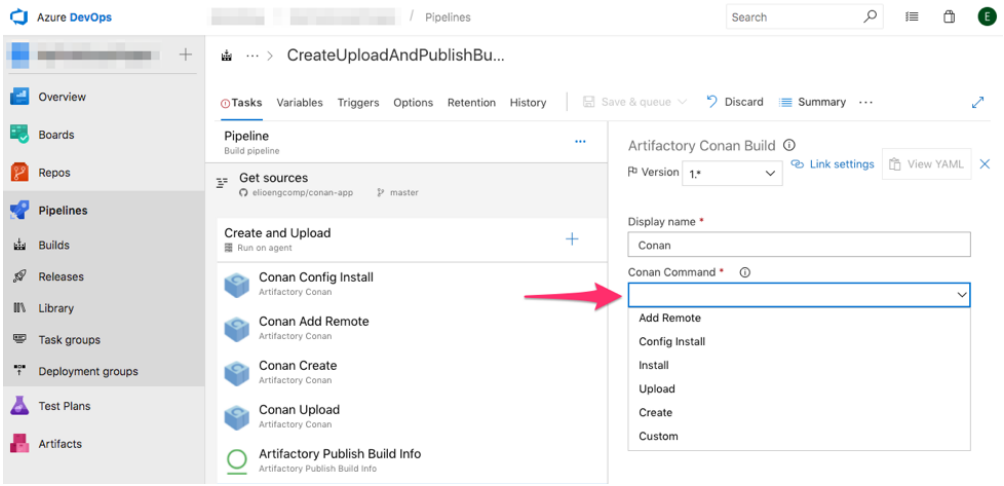
Once your Artifactory connection is configured, you may add Conan tasks to your Build or Release pipelines.

To add a Conan task:

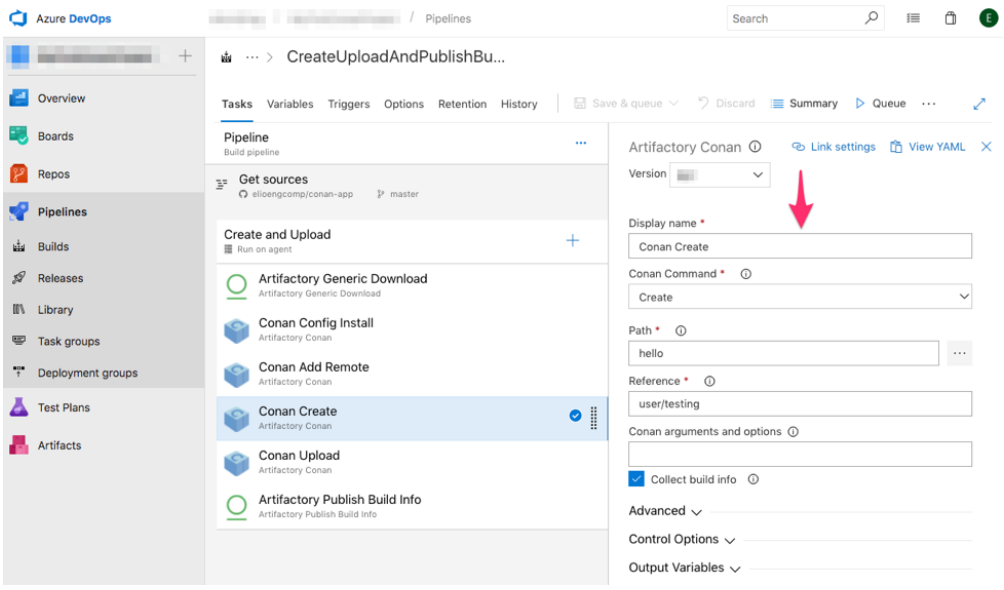
1. Go to the Pipeline Tasks setup screen.
2. In the Add tasks section, search for “Conan” in the task selection list.
3. Select the Artifactory Conan task to add it to your pipeline.



4. In the new task, select which Conan command to run.



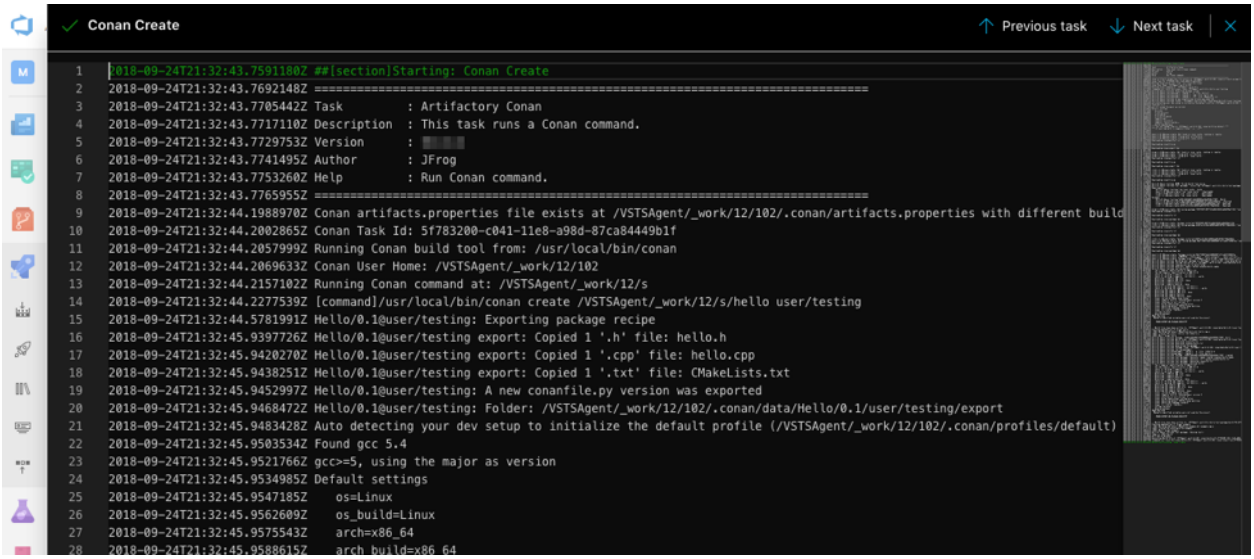
5. Configure the Conan command for the task.



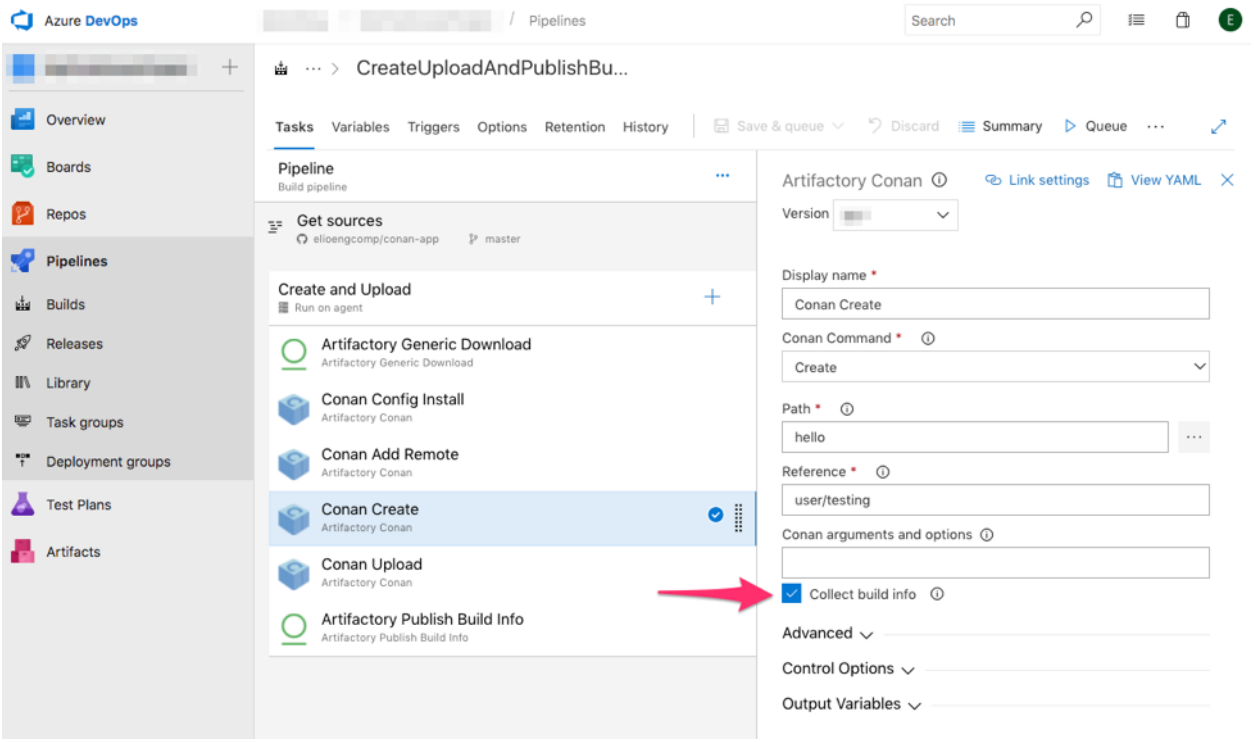
Continue to add Conan tasks as you need for each pipeline.

STEP 3: Configure the Push task buildinfo to Artifactory

When the pipeline containing the Conan task executes, the task log shows all the information about the executed Conan command.

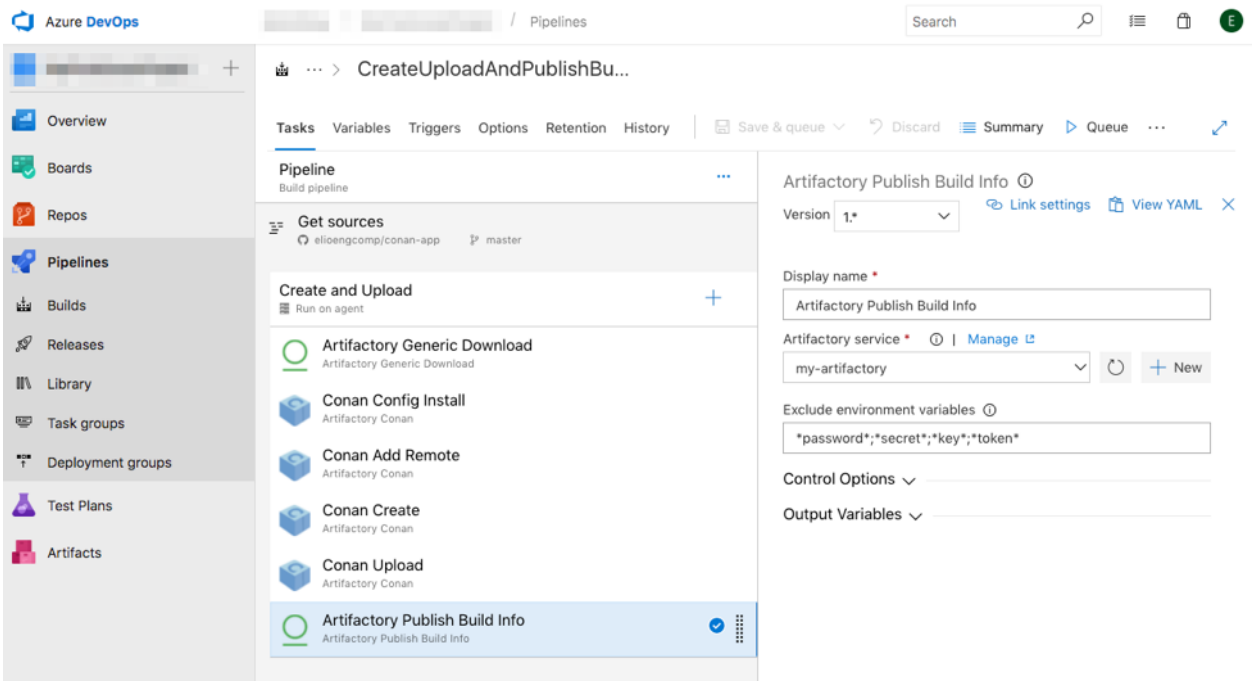


You can configure your Conan task to collect the buildinfo by selecting the Collect buildinfo checkbox when you create the task.

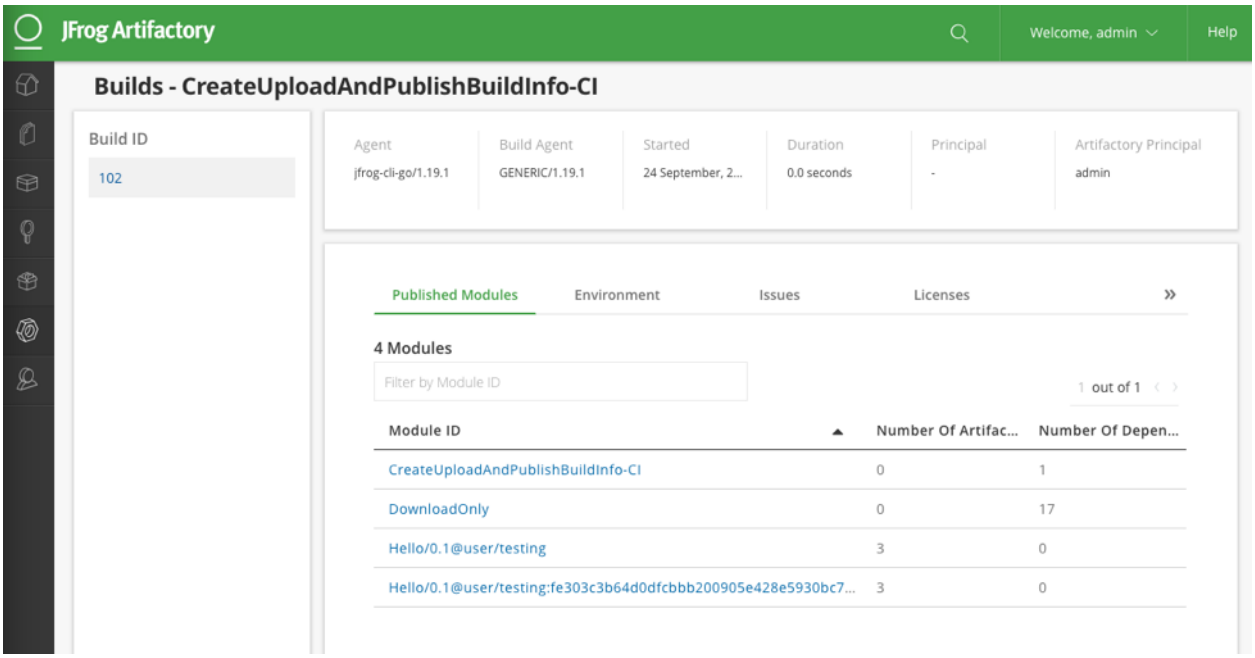


Once collected, the buildinfo can then be pushed as metadata to Artifactory.

To perform this, create an Artifactory Publish Build Info task to push the metadata to your Artifactory instance.



After you run the pipeline, you will be able to see the build information for the Conan task in Artifactory.



See also:

The documentation for this integration is taken from the JFrog blog.

15.5 Other Systems

You can run Conan on any platform supporting Python and also cross-build Conan packages for different platforms.

15.5.1 Buildroot

The Buildroot Project is a tool for automating the creation of Embedded Linux distributions. It builds the code for the architecture of the board so it was set up, all through an overview of Makefiles. In addition to being open-source, it is licensed under GPL-2.0-or-later.

Integration with Conan

Let's create a new file called **pkg-conan.mk** in the *package/* directory. At the same time, we need to add it in *package/Makefile.in* file in order to Buildroot be able to list it.

```
echo 'include package/pkg-conan.mk' >> package/Makefile.in
```

For this development we will break it down into a few steps. Because it is a large file, we will only portray parts of it in this post, but the full version can be found in pkg-conan.mk.

Buildroot defines its settings, including processor, compiler version, and build type through variables. However, these variables do not have directly valid values for Conan, so we need to parse most of them. Let's start with the compiler version, by default Buildroot uses a GCC-based toolchain, so we will only filter on its possible versions:

```
CONAN_SETTING_COMPILER_VERSION  ?=
ifeq ($(BR2_GCC_VERSION_8_X),y)
CONAN_SETTING_COMPILER_VERSION = 8
else ifeq ($(BR2_GCC_VERSION_7_X),y)
CONAN_SETTING_COMPILER_VERSION = 7
else ifeq ($(BR2_GCC_VERSION_6_X),y)
CONAN_SETTING_COMPILER_VERSION = 6
else ifeq ($(BR2_GCC_VERSION_5_X),y)
CONAN_SETTING_COMPILER_VERSION = 5
else ifeq ($(BR2_GCC_VERSION_4_9_X),y)
CONAN_SETTING_COMPILER_VERSION = 4.9
endif
```

This same process should be repeated for build_type, arch, and so on. For the Conan package installation step we will have the following routine:

```
define $(2)_BUILD_CMDS
    $$(TARGET_MAKE_ENV) $$(CONAN_ENV) $$($$(PKG)_CONAN_ENV) \
        CC=$$(TARGET_CC) CXX=$$(TARGET_CXX) \
        $$(CONAN) install $$(CONAN_OPTS) $$($$(PKG)_CONAN_OPTS) \
        $$($$(PKG)_REFERENCE) \
        -s build_type=$$(CONAN_SETTING_BUILD_TYPE) \
        -s arch=$$(CONAN_SETTING_ARCH) \
```

(continues on next page)

(continued from previous page)

```
        -s compiler=$$(CONAN_SETTING_COMPILER) \
        -s compiler.version=$$(CONAN_SETTING_COMPILER_VERSION) \
        -g deploy \
        --build $$(CONAN_BUILD_POLICY)
endef
```

The **conan install** command will be executed as usual, but the settings and options are configured through what was previously collected from Buildroot, and accept new ones through the Buildroot package recipe. Because it was a scenario where previously all sources were compiled in the first moment, we will set Conan build policy to `missing`, so any package will be built if not available.

Also, note that we are using the generator *deploy*, as we will need to copy all the artifacts into the Buildroot internal structure. Once built, we will copy the libraries, binaries and headers through the following routine:

```
define $(2)_INSTALL_CMDS
    cp -f -a $$($$(PKG)_BUILDDIR)/bin/. /usr/bin 2>/dev/null || :
    cp -f -a $$($$(PKG)_BUILDDIR)/lib/. /usr/lib 2>/dev/null || :
    cp -f -a $$($$(PKG)_BUILDDIR)/include/. /usr/include 2>/dev/null || :
endef
```

With this script we will be able to install the vast majority of Conan packages, using only simpler information for each Buildroot recipe.

Creating Conan packages with Buildroot

Installing Conan Zlib

Once we have our script for installing Conan packages, now let's install a fairly simple and well-known project: zlib. For this case we will create a new recipe in the package directory. Let's start with the package configuration file:

```
mkdir package/conan-zlib
touch package/conan-zlib/Config.in
touch package/conan-zlib/conan-zlib.mk
```

The contents of the file *Config.in* should be as follows:

```
config BR2_PACKAGE_CONAN_ZLIB
 bool "conan-zlib"
 help
   Standard (de)compression library. Used by things like
   gzip and libpng.

   http://www.zlib.net
```

Now let's go to the *conan-zlib.mk* that contains the Zlib data:

```
# conan-zlib.mk
CONAN_ZLIB_VERSION = 1.2.11
CONAN_ZLIB_LICENSE = Zlib
CONAN_ZLIB_LICENSE_FILES = licenses/LICENSE
CONAN_ZLIB_SITE = $(call github,conan-io,conan-center-index,
↪134dd3b84d629d27ba3474e01b688e9c0f25b9c8)
```

(continues on next page)

(continued from previous page)

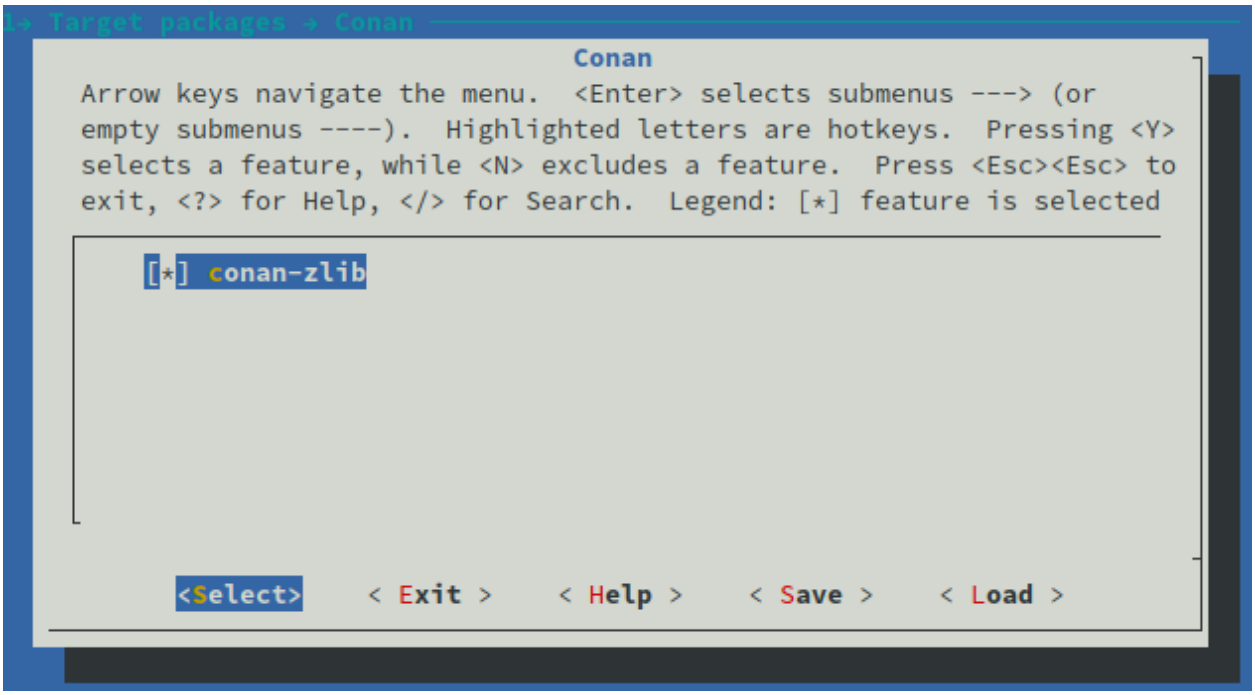
```
CONAN_ZLIB_REFERENCE = zlib/$(CONAN_ZLIB_VERSION)@
CONAN_ZLIB_SUBDIR = recipes/zlib/1.2.11

$(eval $(conan-package))
```

An important note here is the fact that `CONAN_ZLIB_SITE` is required even if not used for our purpose. If it is not present, Buildroot will raise an error during its execution. The other variables are simple, just expressing the package reference, name, version and license. Note that in the end we are calling our script which should execute Conan.

Once created, we still need to add it to the Buildroot configuration list. To do so, let's update the list with a new menu named *Conan*. In *package/Config.in* file, let's add the following section:

```
menu "Conan"
    source "package/conan-zlib/Config.in"
endmenu
```

Now just select the package through **menuconfig**: *Target Packages -> Conan -> conan-zlib*



Once configured and saved, simply run **make** again to install the package.

As you can see, Conan is following the same profile used by Buildroot, which gives us the advantage of not having to create a profile manually.

At the end of the installation it will be copied to the output directory.

Customizing Conan remote

Let's say we have an *Artifactory* instance where all packages are available for download. How could we customize the remote used by Buildroot? We need to introduce a new option, where we can write the remote name and Conan will be able to consume such variable. First we need to create a new configuration file to insert new options in Conan's menu:

```
mkdir package/conan
touch package/conan/Config.in
```

The file *Config.in* should contain:

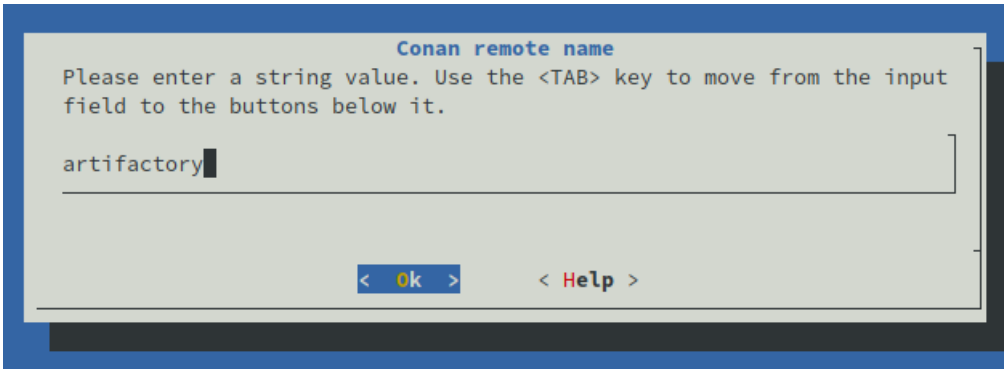
```
config CONAN_REMOTE_NAME
        string "Conan remote name"
    help
          Look in the specified remote server.
```

Also, we need to parse the option `CONAN_REMOTE_NAME` in *pkg-conan.mk* and add it to Conan command line:

```
ifneq ($(CONAN_REMOTE_NAME),"")
CONAN_REMOTE = -r $$(CONAN_REMOTE_NAME)
endif

define $(2)_BUILD_CMDS
    $$(TARGET_MAKE_ENV) $$(CONAN_ENV) $$($$(PKG)_CONAN_ENV) \
        CC=$$(TARGET_CC) CXX=$$(TARGET_CXX) \
        $$(CONAN) install $$(CONAN_OPTS) $$($$(PKG)_CONAN_OPTS) \
        $$($$(PKG)_REFERENCE) \
        -s build_type=$$(CONAN_SETTING_BUILD_TYPE) \
        -s arch=$$(CONAN_SETTING_ARCH) \
        -s compiler=$$(CONAN_SETTING_COMPILER) \
        -s compiler.version=$$(CONAN_SETTING_COMPILER_VERSION) \
        -g deploy \
        --build $$(CONAN_BUILD_POLICY) \
        $$(CONAN_REMOTE)
endef
```

Now we are ready to set our specific remote name. We only need to run **`make menuconfig`** and follow the path: *Target Packages -> Libraries -> Conan -> Conan remote name*

And we will see:



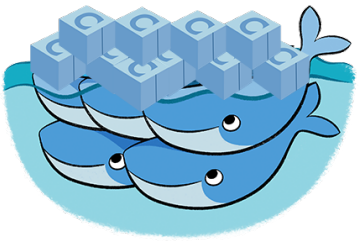
Now Conan is configured to search for packages in the remote named *artifactory*. But we need to run **`make`** again. Note that it will cost less time to build, since now we are using pre-built packages provided by Conan.

If no errors have occurred during the process we will have the following output folder:

```
ls output/images/
    bcm2710-rpi-3-b.dtb bcm2710-rpi-3-b-plus.dtb bcm2710-rpi-cm3.dtb boot.vfat rootfs.
↪ext2 rootfs.ext4 rpi-firmware sdcard.img zImage

ls -lh output/images/sdcard.img
    -rw-r--r-- 1 conan conan 153M ago  6 11:43 output/images/sdcard.img
```

These artifacts are the final compilation of everything that was generated during the build process, here we will be interested in the *sdcard.img* file. This is the final image that we will use on our *RaspberryPi3* and it is only 153MB. Compared to other embedded distributions like *Raspbian*, it is much smaller.

If you are interested in knowing more, we have a complete blog post about Buildroot integration.



15.5.2 Docker

You can easily run Conan in a Docker container to build and cross-build conan packages.

Check the *'How to use docker to create and cross build C and C++ conan packages'* section to know more.

15.5.3 Emscripten

It should be possible to build packages for Emscripten (asm.js) via the following conan profile:

```
include(default)
[settings]
os=Emscripten
arch=asm.js
compiler=clang
compiler.version=6.0
compiler.libcxx=libc++
[options]
[tool_requires]
emsdk_installer/1.38.29@bincrafters/stable
[env]
```

And the following conan profile is required for the WASM (Web Assembly):

```
include(default)
[settings]
os=Emscripten
arch=wasm
```

(continues on next page)

(continued from previous page)

```
compiler=clang
compiler.version=6.0
compiler.libcxx=libc++
[options]
[tool_requires]
emsdk_installer/1.38.29@bincrafters/stable
[env]
```

These profile above are using the emsdk_installer/1.38.29@bincrafters/stable conan package. It will automatically download the Emscripten SDK and set up required environment variables (like `CC`, `CXX`, etc.).

Note: In order to use `emsdk_installer` package, you need to add it to the remotes:

```
$ conan remote add bincrafters https://bincrafters.jfrog.io/artifactory/api/conan/public-
↪conan
```

Note: Alternatively, it's always possible to use an existing `emsdk` installation and manually specify required environment variables within the *[env]* section of the conan profile.

Note: In addition to the above, Windows users may need to specify `CONAN_MAKE_PROGRAM`, for instance from the existing MinGW installation (e.g. `C:\MinGW\bin\mingw32-make.exe`), or use make from the `mingw_installer/1.0@conan/stable`.

Note: In addition to the above, Windows users may need to specify `CONAN_CMAKE_GENERATOR`, e.g. to `MinGW Makefiles`, because default one is Visual Studio. Other options (e.g. Ninja) work as well.

As specified, `os` has been set to the `Emscripten`, and `arch` has been set to either `asm.js` or `wasm` (only these two are currently supported). And `compiler` setting has been set to match the one used by `Emscripten` - Clang 6.0 with libc++ standard library.

Running the code inside the browser

Note: Emscripten requires Python 2.7.12 or above, make sure that you have an up-to-date Python version installed.

Note: Running demo on Windows may require pywin32 module. Install it by running `pip install pywin32`.

In order to demonstrate how to use conan with Emscripten, let's check out the example project:

```
$ git clone --depth 1 git@github.com:conan-io/examples.git
```

Change the directory to the Emscripten demo:

```
$ cd features
$ cd emscripten
```

This is an extremely simple demo, which just imports the famous zlib library and outputs its version into the browser.

In order to build it for the Emscripten run:

```
$ ./build.sh
```

or (on Windows):

```
$ ./build.cmd
```

Please note that running the above command may take a while to download and tool required dependencies. This script will execute several conan commands:

```
$ conan remove conan-hello-emscripten/* -f
$ conan create . conan/testing  -k -p emscripten.profile --build missing
$ conan install conanfile.txt  -pr emscripten.profile
```

First one removes any traces of previous demo installations, just to ensure that environment is clean. Then, it builds the simple demo (it uses `CMakeLists.txt` and `main.cpp` files from the current directory). The following local profile is used (file `emscripten.profile` within the current directory):

```
include(default)
[settings]
os=Emscripten
arch=wasm
compiler=clang
compiler.version=6.0
compiler.libcxx=libc++
[options]
[tool_requires]
emsdk_installer/1.38.29@bincrafters/stable
ninja/1.9.0
[env]
```

Finally, it installs the demo importing ithe required files (`.html`, `.js` and `.wasm`) into the `bin` subdirectory.

Then we can run the code inside the browser via emrun helper:

```
$ ./run.sh
```

or (on Windows):

```
$ ./run.cmd
```

The command above uses *virtualenv generator* generator in order to get `emrun` command available in the `PATH`. And as the result, Web Browser should be opened (or new tab in Web Browser will be opened, if it was already run), and the following output should be displayed:

```
$ Using zlib version: 1.2.11
```

It confirms the fact we have just built `zlib` into JavaScript and run it inside the Web Browser.

15.5.4 QNX SOFTWARE SYSTEMS QNX Neutrino

It's possible to cross-compile packages for QNX Neutrino operating with Conan.

Conan has support for QNX Neutrino 6.x and 7.x. The following architectures are supported:

- armv7
- armv8
- sh4le
- ppc32be

The following C++ standard library implementations are supported for QCC:

- cxx (LLVM C++)
- gpp (GNU C++)
- cpp (Dinkum C++)
- cpp-ne (Dinkum C++ without exceptions)
- acpp (Dinkum Abridged C++)
- acpp-ne (Dinkum Abridged C++ without exceptions)
- ecpp (Dinkum Embedded C++)
- ecpp-ne (Dinkum Embedded C++ without exceptions)

Conan automatically sets up corresponding compiler flags for the given standard library (e.g. **-Y cxx** for the LLVM C++).

With QNX SDK set up on the machine, the following conan profile might be used for the cross-compiling (assuming **qcc** in the PATH):

```
include(default)
[settings]
os=Neutrino
os.version=6.5
arch=sh4le
compiler=qcc
compiler.version=4.4
compiler.libcxx=cxx
[options]
[tool_requires]
[env]
CC=qcc
CXX=QCC
```

15.5.5 Yocto

The Yocto Project is an open-source project that delivers a set of tools that create operating system images for embedded Linux systems. The Yocto Project tools are based on the OpenEmbedded project, which uses the BitBake build tool, to construct complete Linux images.

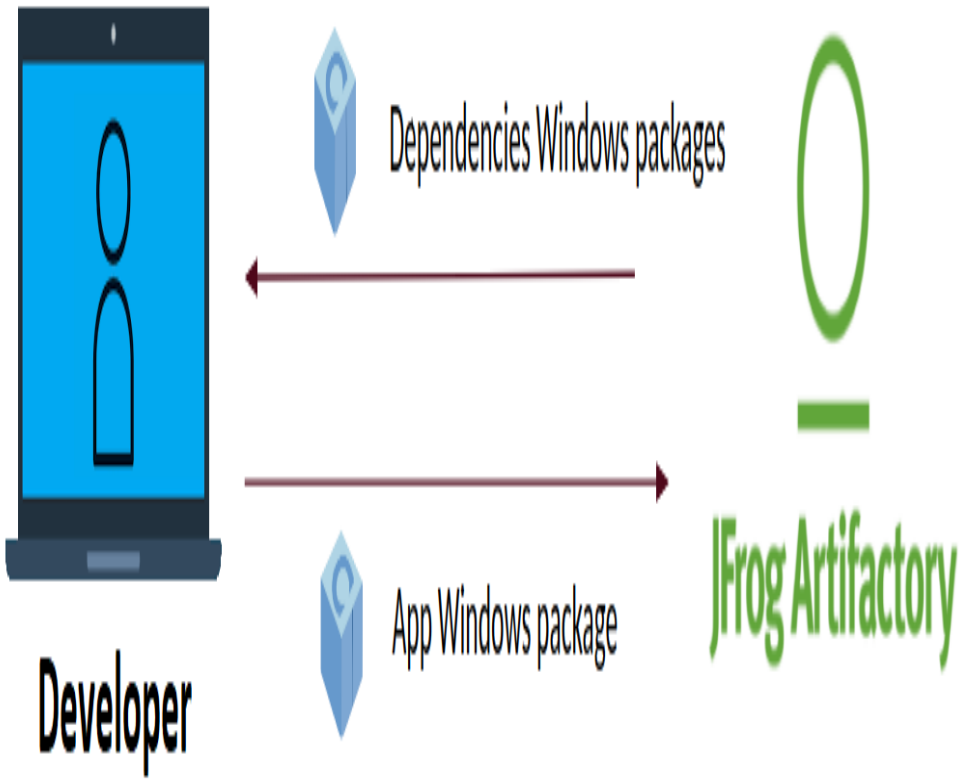
Yocto supports several Linux host distributions and it also provides a way to install the correct version of these tools by either downloading a buildtools-tarball or building one on a supported machine. This allows virtually any Linux distribution to be able to run Yocto, and also makes sure that it will be possible to replicate your Yocto build system in the future. The Yocto Project build system also isolates itself from the host distribution's C library, which makes it possible to share build caches between different distributions and also helps in future-proofing the build system.

Integration with Conan

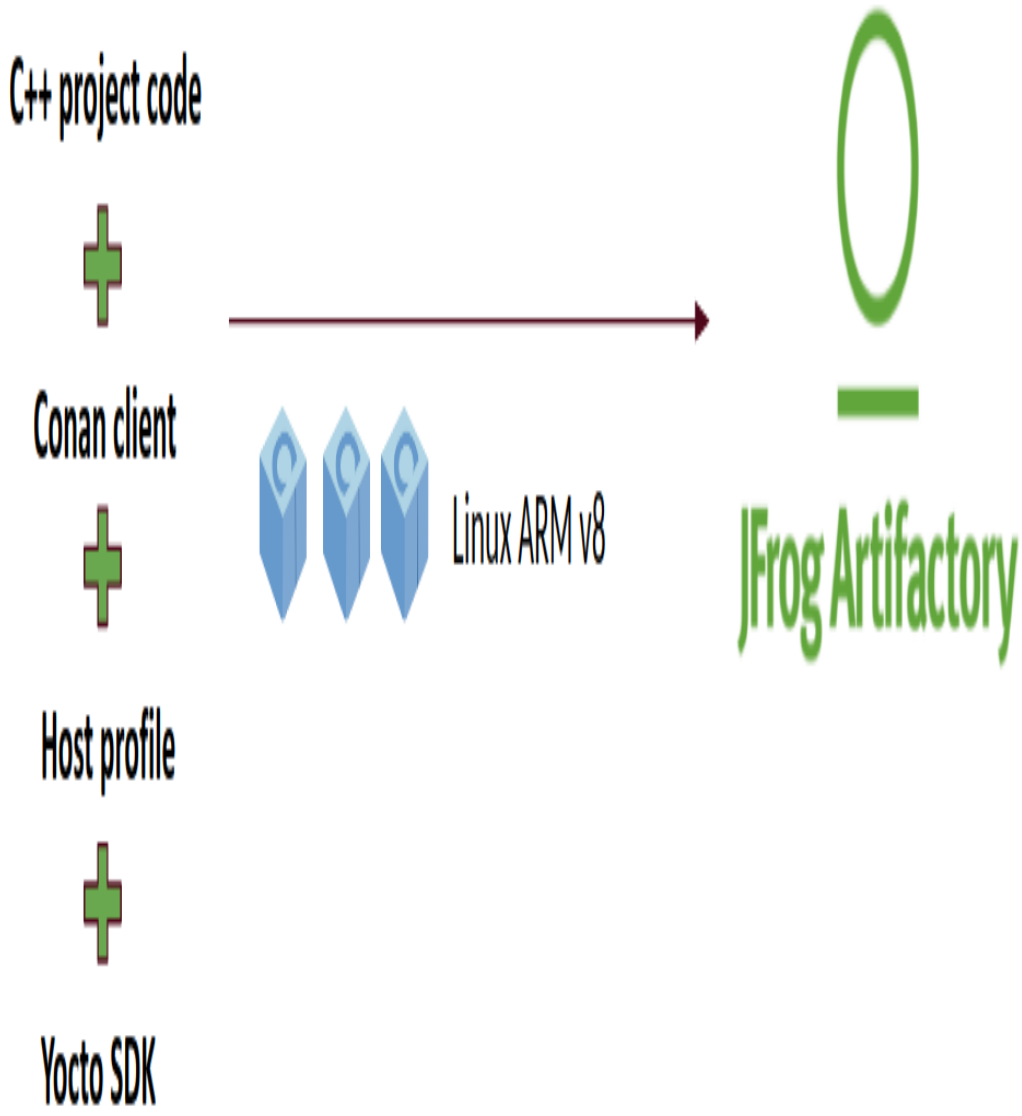
You can create Conan packages building with the Yocto SDK as any other package for other configuration. Those packages can be integrated into a Yocto build installing them from a remote and without compiling them again.

Three stages can be differentiated in the proposed flow:

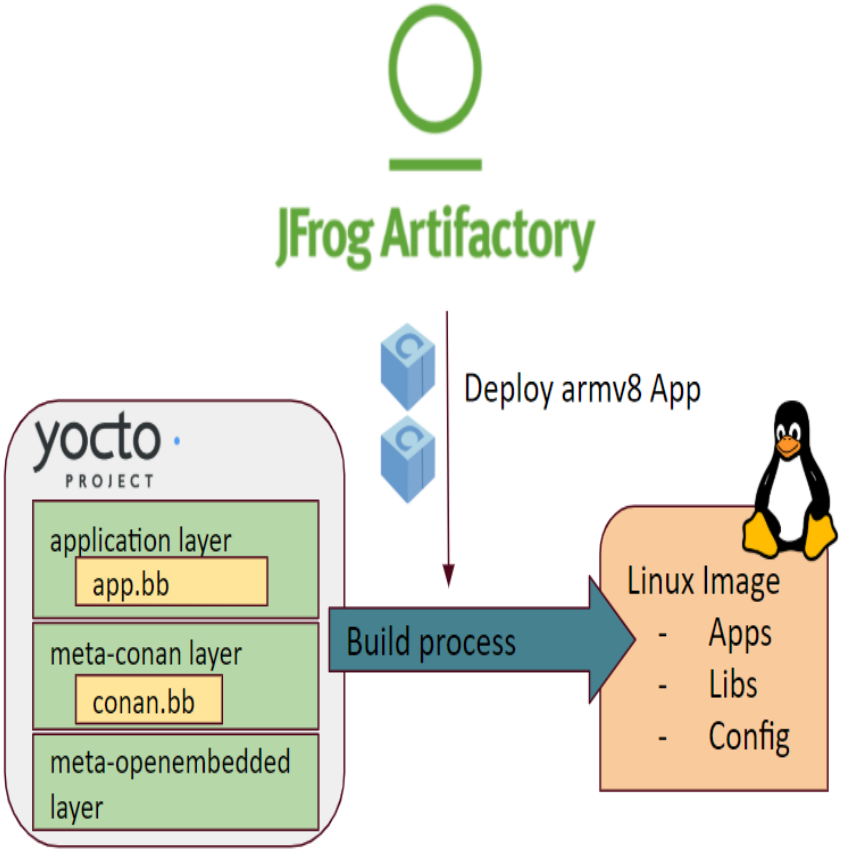
1. Developers can create an application with the native tools in their desktop platform of choice using their usual IDE, compiler or debugger and test the application.



2. Packages can be cross-built for the target device using the Yocto SDK and uploaded to Artifactory, even automated in a CI process.



3. Once the cross-built packages are available in Artifactory, the application can be directly deployed to the Yocto image without building it from sources again.



Creating Conan packages with Yocto's SDK

Prepare your recipes

First of all, the recipe of the application to be deployed to the final image should have a deploy() method. There you can specify the files of the application needed in the image as well as any other from its dependencies (like shared libraries or assets):

```
conan install
:caption: *conanfile.py*
:emphasize-lines: 28-31

 from conans import ConanFile


 class FoobarConan(ConanFile):
     name = "foobar"
     ...

 def package(self):
     self.copy("*.h", dst="include", src="hello")
     self.copy("*.so", dst="lib", keep_path=False)
     self.copy("*.a", dst="lib", keep_path=False)
```

(continues on next page)

(continued from previous page)

```
    self.copy("foobar", dst="bin", keep_path=False)

def deploy(self):
    # Deploy the executables from this eclipse/mosquitto package
    self.copy("*", src="bin", dst="bin")
    # Deploy the shared libs from this eclipse/mosquitto package
    self.copy("*.so*", src="lib", dst="bin")
   # Deploy all the shared libs from the transitive deps
    self.copy_deps("*.so*", src="lib", dst="bin")
```

Another option is using the deploy generator generator,
which will copy all artifacts, including package dependencies to your installation folder.

Setting up a Yocto SDK

Yocto SDKs are completely self-contained, there is no dependency on libraries of the build machine or tools installed in it. The SDK is a cross-building toolchain matching the target and it is generated from that specific configuration. This means that you will have to use a different SDK toolchain to build for a different target architecture or that some SDK's may have specific settings to enable some system dependency of the final target and those libraries will be available in the SDK.

You can create your own Yocto SDKs or download and use the prebuilt ones.

In the case that you are using CMake to create the Conan packages, Yocto injects a toolchain that configures CMake to only search for libraries in the rootpath of the SDK with CMAKE_FIND_ROOT_PATH. This is something that has to be patched to allow CMake to find libraries in the Conan cache as well:

Listing 23: *sdk/sysroots/x86_64-pokysdk-linux/usr/share/cmake/OEToolchainConfig.cmake*

```
set( CMAKE_FIND_ROOT_PATH $ENV{OECORE_TARGET_SYSROOT} $ENV{OECORE_NATIVE_SYSROOT} )
set( CMAKE_FIND_ROOT_PATH_MODE_PROGRAM NEVER )
# COMMENT THIS: set( CMAKE_FIND_ROOT_PATH_MODE_LIBRARY ONLY )
# COMMENT THIS: set( CMAKE_FIND_ROOT_PATH_MODE_INCLUDE ONLY )
# COMMENT THIS: set( CMAKE_FIND_ROOT_PATH_MODE_PACKAGE ONLY )
```

You can read more about those variables here:

- CMAKE_FIND_ROOT_PATH_MODE_LIBRARY
- CMAKE_FIND_ROOT_PATH_MODE_INCLUDE
- CMAKE_FIND_ROOT_PATH_MODE_PACKAGE

Cross-building Conan packages with the SDK toolchain

After setting up your desired SDK, you can start creating Conan packages setting up the environment of the Yocto SDK and running a **`conan create`** command with a suitable profile with the specific architecture of the toolchain.

For example, creating packages for *arch=armv8*:

The profile will be:

Listing 24: *armv8*

```
[settings]
os_build=Linux
arch_build=x86_64
os=Linux
arch=armv8
compiler=gcc
compiler.version=8
compiler.libcxx=libstdc++11
build_type=Release
```

Activate the SDK environment and execute the create command if you have a specific recipe:

```
$ source oe-environment-setup-aarch64-poky-linux
$ conan create . user/channel --profile armv8
```

However, if you wish an official Conan package from Conan Center, you can install it directly:

```
$ source oe-environment-setup-aarch64-poky-linux
$ conan install mosquitto/2.0.14@ -g deploy --profile:build=default --profile:host=armv8
```

This will generate the packages using the Yocto toolchain from the environment variables such as `CC`, `CXX`, `LD`... Now you can *upload the binaries* to an Artifactory server to share and reuse in your Yocto builds.

```
$ conan upload mosquitto/2.0.14@ --all --remote my_repo
```

Important: We strongly recommend using the Yocto's SDK toolchain to create packages as they will be built with the optimization flags suitable to be deployed later to an image generated in a Yocto build.

Deploying an application to a Yocto image

Now that you have your cross-built Conan packages in Artifactory, you can deploy them in a Yocto build.

Set up the Conan layer

We have created a meta-conan layer that includes all the configuration, the Conan client and a generic BitBake recipe. To add the layer you will have to clone the repository and the dependency layers of `meta-openembedded`:

```
$ cd poky
$ git clone https://github.com/conan-io/meta-conan.git
$ git clone --branch thud https://github.com/openembedded/meta-openembedded.git
```

You would also have to activate the layers in the *bblayers.conf* file of your build folder:

Listing 25: *conf/bblayers.conf*

```
POKY_BBLAYERS_CONF_VERSION = "2"

BBPATH = "${TOPDIR}"
```

(continues on next page)

(continued from previous page)

```
BBFILES ?= ""

BBLAYERS ?= " \
/home/username/poky/meta \
/home/username/poky/meta-poky \
/home/username/poky/meta-yocto-bsp \
/home/username/poky/meta-openembedded/meta-oe \
/home/username/poky/meta-openembedded/meta-python \
/home/username/poky/meta-conan \
"
```

Or, if you are not confident editing the configuration file or just to automate all process, you can use bitbake commands:

```
$ cd build/
$ bitbake-layers add-layer ${PWD}/../poky/meta-openembedded/meta-oe
$ bitbake-layers add-layer ${PWD}/../poky/meta-openembedded/meta-python
$ bitbake-layers add-layer ${PWD}/../poky/meta-conan
```

Note: Please report any question, feature request or issue related to the `meta-conan` layer in its GitHub issue tracker.

Write the Bitbake recipe for the Conan package

With the `meta-conan` layer, a Conan recipe to deploy a Conan package should look as easy as this recipe:

Listing 26: *conan-mosquitto_2.0.14.bb*

```
inherit conan

DESCRIPTION = "An open source MQTT broker"
LICENSE = "EPL-1.0"

CONAN_PKG = "mosquitto/2.0.14@"
```

This recipe will be placed inside your application layer that should be also added to the *conf/bblayers.conf* file.

Configure Conan variables for the build

Additionally to the recipe, you will need to provide the information about the credentials for Artifactory or the profile to be used to retrieve the packages in the *local.conf* file of your build folder.

Listing 27: *poky_build_folder/conf/local.conf*

```
IMAGE_INSTALL_append = " conan-mosquitto"

# Profile for installation
CONAN_PROFILE_PATH = "${TOPDIR}/conf/armv8"
# Artifactory repository
CONAN_REMOTE_URL = "https://localhost:8081/artifactory/api/conan/<repository>"
# Artifactory Credentials
```

(continues on next page)

(continued from previous page)

```
CONAN_USER = "REPO_USER"
CONAN_PASSWORD = "REPO_PASSWORD"
```

Notice the *armv8* profile to indicate your configuration next to the *local.conf*. That way you will be able to match the Conan configuration with the specific architecture or board of your Yocto build.

Listing 28: *poky_build_folder/conf/armv8*

```
[settings]
os_build=Linux
arch_build=x86_64
os=Linux
arch=armv8
compiler=gcc
compiler.version=8
compiler.libcxx=libstdc++11
build_type=Release
```

It is recommended to set up the specific profile to use in your build with `CONAN_PROFILE_PATH` pointing to profile stored in the configuration folder of your build (next to the *conf/local.conf* file), for example: `CONAN_PROFILE_PATH = "${TOPDIR}/conf/armv8"`.

Finally, the Artifactory repository URL where you want to retrieve the packages from and its credentials.

You can also use `CONAN_CONFIG_URL` with a custom Conan configuration to be used with **`conan config install`** and the name of the profile to use in `CONAN_PROFILE_PATH` and just the name of the remote in `CONAN_REMOTE_NAME`. For example:

Listing 29: *poky_build_folder/conf/local.conf*

```
IMAGE_INSTALL_append = " conan-mosquitto"

CONAN_CONFIG_URL = "https://github.com/<your-organization>/conan-config.git"
CONAN_PROFILE_PATH = "armv8"
CONAN_REMOTE_NAME = "my_repo"
CONAN_USER = "REPO_USER"
CONAN_PASSWORD = "REPO_PASSWORD"
```

In this case the *armv8* profile and the `my_repo` remote will be taken from the ones installed with the **`conan config install`** command.

Architecture conversion table

If no specific profile is indicated in `CONAN_PROFILE_PATH`, Conan will map the most common Yocto architectures and machines to the existing ones in Conan. This is the current mapping from Conan architectures to the Yocto ones:

Yocto SDK	Yocto Machine	Conan arch setting
aarch64	qemuarm64	armv8
armv5e	qemuarmv5	armv5el
core2-64	qemux86_64	x86_64
cortexa8hf	quemuarm	armv7hf
i586	qemux86	x86
mips32r2	qemumips	mips
mips64	qemumips64	mips64
ppc7400	qemuppc	ppc32

This mapping may not be complete and some of the binaries generated with the Yocto toolchains will have specific optimization flags for the specific architectures.

Tip: For heavy Yocto users, having a custom setting for this may be very useful. For example, including the specific architecture names in your *settings.yml*

```
arch: [..., "aarch64", "armv5e", "core2-64", ...]
```

Or using a `machine` subsetting under the `Linux` operating system:

```
os:
    Linux:
        machine: [None, "qemuarm64", "qemuarm64", "qemux86_64", ...]
```

Note that the `None` value is important here to be able to build other packages without value for this subsetting to target a non-yocto Linux distro.

See also:

- Yocto Machine configurations: https://git.yoctoproject.org/cgit.cgi/poky/tree/meta/conf/machine
- Conan Architectures in *settings.yml*.

Deploy the application and its dependencies to the final image

You can build the recipe to test that the packages are correctly deployed:

```
$ bitbake -c install conan-mosquitto
```

Packages will be installed with the profile indicated and installed with its dependencies only from the remote specified.

Finally, you can build your image with the Conan packages:

```
$ bitbake core-image-minimal
```

The binaries of **the Conan packages will be deployed** to the */bin* folder of the image once it is created.

15.5.6 Android

There are several ways to cross-compile packages for Android platform via conan.

Using android-ndk package (tool require)

The easiest way so far is to use android-ndk conan package (which is in `conancenter` repository).

Using the `android-ndk` package as a tool requirement will do the following steps:

- Download the appropriate Android NDK archive.
- Set up required environment variables, such as `CC`, `CXX`, `RANLIB` and so on to the appropriate tools from the NDK.
- In case of using CMake, it will inject the appropriate toolchain file and set up the necessary CMake variables.

For instance, in order to cross-compile for `ARMv8`, the following conan profile might be used:

```
include(default)
[settings]
arch=armv8
build_type=Release
compiler=clang
compiler.libcxx=libc++
compiler.version=11
os=Android
os.api_level=21
[tool_requires]
android-ndk/r22b
[options]
[env]
```

Note: In addition to the above, Windows users may need to specify `CONAN_MAKE_PROGRAM`, for instance from the existing MinGW installation (e.g. `C:\MinGW\bin\mingw32-make.exe`), or use make from the `mingw_installer/1.0@conan/stable`.

Similar profile might be used to cross-compile for `ARMv7` (notice the `arch` change):

```
include(default)
[settings]
```

(continues on next page)

(continued from previous page)

```
arch=armv7
build_type=Release
compiler=clang
compiler.libcxx=libc++
compiler.version=11
os=Android
os.api_level=21
[tool_requires]
android-ndk/r22b
[options]
[env]
```

By adjusting `arch` setting, you may cross-compile for `x86` and `x86_64` Android as well (e.g. if you need to run code in a simulator).

Note: `os.api_level` is an important setting which affects compatibility - it defines the **minimum** Android version supported. In other words, it is the same meaning as minSdkVersion.

Use built-in Conan toolchain

Warning: This is an **experimental** feature subject to breaking changes in future releases.

Conan will generate a toolchain for Android if the recipe is using a *CMakeToolchain*. In that case all you need is to provide the path to the Android NDK and *working profiles*. This approach can also use the Android NDK package referenced in the previous section.

Use a regular profile for the *host* context:

Listing 30: **profile_host**

```
[settings]
os=Android
os.api_level=23
arch=x86_64
compiler=clang
compiler.version=9
compiler.libcxx=c++_shared
build_type=Release
```

and add Android NDK to the `PATH` or populate the `CONAN_CMAKE_ANDROID_NDK` environment variable.

Together with the files created by the generators that make it possible to find and link the requirements, **`conan install`** command will generate a toolchain file like the following one:

Listing 31: **conan_toolchain.cmake** (some parts are stripped)

```
set(CMAKE_BUILD_TYPE "Release" CACHE STRING "Choose the type of build." FORCE)

set(CMAKE_SYSTEM_NAME Android)
set(CMAKE_SYSTEM_VERSION 23)
```

(continues on next page)

(continued from previous page)

```
set(CMAKE_ANDROID_ARCH_ABI x86_64)
set(CMAKE_ANDROID_STL_TYPE c++_shared)
set(CMAKE_ANDROID_NDK <path/provided/via/environment/variable>)
```

With this toolchain file you can execute CMake's command to generate the binaries:

```
conan install <conanfile> --profile:host=profile_host --profile:build=default
cmake . -DCMAKE_TOOLCHAIN_FILE=conan_toolchain.cmake -DCMAKE_BUILD_TYPE=Release
cmake --build . --config Release
```

Using Docker images

If you're using Docker for builds, you may consider using docker images from the Conan Docker Tools repository.

Currently, Conan Docker Tools provide the following Android images:

- conanio/android-clang8
- conanio/android-clang8-x86
- conanio/android-clang8-armv7
- conanio/android-clang8-armv8

All above mentioned images have corresponding Android NDK installed, with required environment variables set and with default conan profile configured for android cross-building. Therefore, these images might be especially useful for CI systems.

Using existing NDK

It's also possible to use an existing Android NDK installation with conan. For instance, if you're using Android Studio IDE, you may already have an NDK at `~/Library/Android/sdk/ndk`.

You have to specify different environment variables in the Conan profile for make-based projects. For instance:

```
include(default)
target_host=aarch64-linux-android
android_ndk=/home/conan/Library/Android/sdk/ndk/20.0.5594570
api_level=21
[settings]
arch=armv8
build_type=Release
compiler=clang
compiler.libcxx=libc++
compiler.version=8
os=Android
os.api_level=$api_level
[tool_requires]
[options]
[env]
PATH=[$android_ndk/toolchains/llvm/prebuilt/darwin-x86_64/bin]
CHOST=$target_host
AR=$target_host-ar
AS=$target_host-as
```

(continues on next page)

(continued from previous page)

```
RANLIB=$target_host-ranlib
CC=$target_host$api_level-clang
CXX=$target_host$api_level-clang++
LD=$target_host-ld
STRIP=$target_host-strip
```

However, when building CMake projects, there are several approaches available, and it's not always clear which one to follow.

Using toolchain from Android NDK

This is the official way recommended by Android developers.

For this, you will need a small CMake toolchain file:

```
set(ANDROID_PLATFORM 21)
set(ANDROID_ABI arm64-v8a)
include($ENV{HOME}/Library/Android/sdk/ndk/20.0.5594570/build/cmake/android.toolchain.
↪cmake)
```

This toolchain file only sets up the required CMake variables, and then includes the default toolchain file supplied with Android NDK.

And then, you may use the following profile:

```
include(default)
[settings]
arch=armv8
build_type=Release
compiler=clang
compiler.libcxx=libc++
compiler.version=8
os=Android
os.api_level=21
[tool_requires]
[options]
[env]
CONAN_CMAKE_TOOLCHAIN_FILE=/home/conan/my_android_toolchain.cmake
```

In the profile, `CONAN_CMAKE_TOOLCHAIN_FILE` points to the CMake toolchain file listed above.

Using CMake build-in Android NDK support

> **Warning:** This workflow is not supported by Android and is often broken with new NDK releases or when using older versions of CMake. This workflow is **strongly discouraged** and will not work with Gradle.

For this approach, you don't need to specify CMake toolchain file at all. It's enough to indicate `os` is Android and Conan will automatically set up all required CMake variables for you.

Therefore, the following conan profile could be used for `ARMv8`:

```
include(default)
[settings]
arch=armv8
build_type=Release
compiler=clang
compiler.libcxx=libc++
compiler.version=7.0
os=Android
os.api_level=21
[tool_requires]
[options]
[env]
ANDROID_NDK_ROOT=/home/conan/android-ndk-r18b
```

The only way you have to configure is `ANDROID_NDK_ROOT` which is a path to the Android NDK installation.

Once profile is configured, you should see the following output during the CMake build:

```
-- Android: Targeting API '21' with architecture 'arm64', ABI 'arm64-v8a', and processor
↪'aarch64'
-- Android: Selected Clang toolchain 'aarch64-linux-android-clang' with GCC toolchain
↪'aarch64-linux-android-4.9'
```

It means native CMake integration has successfully found Android NDK and configured the build.

15.5.7 iOS, tvOS, watchOS

Using Darwin toolchain package (tool require)

> **Warning:** This is an **experimental** feature subject to breaking changes in future releases.

One example of a tool requires implementing a toolchain to cross-compile to iOS, tvOS or watchOS, is the Darwin Toolchain package. Although this package is not in Conan Center Index you can check it to see an example of how to use a toolchain for cross-compilation by using a tool requires. You can use a profile like the following to cross-build your packages for `iOS`, `watchOS` and `tvOS`:

Listing 32: ios_profile

```
include(default)

[settings]
os=iOS
os.version=9.0
arch=armv7

[tool_requires]
darwin-toolchain/1.0@theodelrieu/stable
```

```
$ conan install . --profile ios_profile
```

Use built-in Conan toolchain

> **Warning:** This is an **experimental** feature subject to breaking changes in future releases.

Conan will generate a toolchain for iOS if the recipe is using a *CMakeToolchain*. This toolchain provides a minimal implementation supporting only the CMake XCode generator. It will be extended in the future but at the current version (1.31.0) is just for testing purpouses.

For using it, create a regular profile for the *host* context:

Listing 33: **profile_host_ios**

```
[settings]
 os=iOS
 os.version=12.0
 arch=armv8
 compiler=apple-clang
 compiler.version=12.0
 compiler.libcxx=libc++
 build_type=Release
```

Together with the files created by the generators that make it possible to find and link the requirements, **conan install** command will generate a toolchain file like the following one:

Listing 34: **conan_toolchain.cmake** (some parts are stripped)

```
set(CMAKE_BUILD_TYPE "Release" CACHE STRING "Choose the type of build." FORCE)

# set cmake vars
set(CMAKE_SYSTEM_NAME iOS)
set(CMAKE_SYSTEM_VERSION 12.0)
set(DEPLOYMENT_TARGET ${CONAN_SETTINGS_HOST_MIN_OS_VERSION})
# Set the architectures for which to build.
set(CMAKE_OSX_ARCHITECTURES arm64)
# Setting CMAKE_OSX_SYSROOT SDK, when using Xcode generator the name is enough
# but full path is necessary for others
set(CMAKE_OSX_SYSROOT iphoneos)
```

With this toolchain file you can execute CMake's command to generate the binaries:

```
conan install <conanfile> --profile:host=profile_host_ios --profile:build=default
cmake . -GXcode -DCMAKE_TOOLCHAIN_FILE=conan_toolchain.cmake
cmake --build . --config Release
```

15.5.8 VxWorks

It's possible to cross-compile packages for VxWorks operating with Conan.

Conan has support for VxWorks 7. The following architectures are supported:

- armv7

The following C++ standard library implementations are supported for QCC:

- clang++ (LLVM C++)
- g++ (GNU C++)

With a proper build VxWorks Source Build (VSB) set up on the machine, the following conan profile might be used for the cross-compiling (assuming clang in the PATH):

```
include(default)
[settings]
os=VxWorks
os.version=7
arch=armv7
compiler=clang
compiler.version=12
compiler.libcxx=libstdc++11
[options]
[tool_requires]
[env]
```

15.6 Version Control System

Conan uses plain text files for the recipes and configuration files and they can be managed nicely with any version control system. Also, with the *scm* feature, your recipe can capture automatically the `commit/revision` of the source code of your library so the recipe will clone the correct sources automatically.

15.6.1 Git

Conan uses plain text files, `conanfile.txt` or `conanfile.py`, so it's perfectly suitable for the use of any version control system. We use and highly recommend **`git`**.

Check *workflows section* to learn more about project layouts that naturally fit version control systems.

Temporary files

Conan generates some files that should not be committed, as *conanbuildinfo.** and *conaninfo.txt*. These files can change in different computers and are re-generated with the **conan install** command.

However, these files are typically generated in the **build tree** not in the source tree, so they will be naturally disregarded. Just take care in case you have created the **build** folder inside your project (we do this in several examples in the documentation). In this case, you should add it to your *.gitignore* file:

Listing 35: *.gitignore*

```
...
build/
```

Package creators

Check *scm feature* to learn more about managing the libraries source code with Git.

If you are creating a **Conan** package:

- You can use the *url field* to indicate the origin of your package recipe. If you are using an external package recipe, this url should point to the package recipe repository **not** to the external source origin. If a **github** repository is detected, the Conan website will link your github issues page from your Conan's package page.
- You can use **git** to *obtain your source* (requires the git client in the path) when creating external package recipes.

15.6.2 SVN

Conan uses plain text files, `conanfile.txt` or `conanfile.py`, so it's perfectly suitable for the use of any version control system.

Check *workflows section* to learn more about project layouts that naturally fit version control systems.

Check *scm feature* to learn more about managing the libraries source code with SVN.

15.7 Custom integrations

If you intend to use a build system that does not have a built-in generator, you may still be able to do so. There are several options:

- First, search in ConanCenter for generator packages. Generators can be created and contributed by users as regular packages, so you can depend on them as a normal requirement, use versioning and evolve faster without depending on the Conan releases.
- You can use the *txt* or *json* generators. They will generate a text file, simple to read that you can easily parse with your tools to extract the required information.
- Use the **conanfile data model** (*deps_cpp_info*, *deps_env_info*) in your recipe to access its properties and values, so you can directly call your build system with that information, without requiring to generate a file.

- Write and **create your own generator**. So you can upload it, version and reuse it, as well as share it with your team or community. Check *How to create and share a custom generator with generator packages*.

Note: Need help integrating your build system? Tell us what you need: info@conan.io

15.7.1 Use the JSON generator

Specify the `json` generator in your recipe:

Listing 36: *conanfile.txt*

```
[requires]
fmt/6.1.2
poco/1.9.4

[generators]
json
```

A file named *conanbuildinfo.json* will be generated. It will contain the information about every dependency:

Listing 37: *conanbuildinfo.json*

```
{
  "dependencies":
  [
    {
      "name": "fmt",
      "version": "6.1.2",
      "include_paths": [
        "/path/to/.conan/data/fmt/6.1.2/_/_/package/<id>/include"
      ],
      "lib_paths": [
        "/path/to/.conan/data/fmt/6.1.2/_/_/package/<id>/lib"
      ],
      "libs": [
        "fmt"
      ],
      "...": "...",
    },
    {
      "name": "poco",
      "version": "1.9.4",
      "...": "..."
    }
  ]
}
```

15.7.2 Use the text generator

Just specify the `txt` generator in your recipe:

Listing 38: *conanfile.txt*

```
[requires]
poco/1.9.4

[generators]
txt
```

A file is generated with the same information in a generic text format.

Listing 39: *conanbuildinfo.txt*

```
[includedirs]
/home/user/.conan/data/poco/1.9.4/_/_/package/58080bce1cc38259eb7c282aa95c25aecde8efe4/
↪include
/home/user/.conan/data/openssl/1.0.2t/_/_/package/
↪f99afdbf2a1cc98ba2029817b35103455b6a9b77/include
/home/user/.conan/data/zlib/1.2.11/_/_/package/6af9cc7cb931c5ad942174fd7838eb655717c709/
↪include

[libdirs]
/home/user/.conan/data/poco/1.9.4/_/_/package/58080bce1cc38259eb7c282aa95c25aecde8efe4/
↪lib
/home/user/.conan/data/openssl/1.0.2t/_/_/package/
↪f99afdbf2a1cc98ba2029817b35103455b6a9b77/lib
/home/user/.conan/data/zlib/1.2.11/_/_/package/6af9cc7cb931c5ad942174fd7838eb655717c709/
↪lib

[bindirs]
/home/user/.conan/data/openssl/1.0.2t/_/_/package/
↪f99afdbf2a1cc98ba2029817b35103455b6a9b77/bin

[resdirs]
/home/user/.conan/data/openssl/1.0.2t/_/_/package/
↪f99afdbf2a1cc98ba2029817b35103455b6a9b77/res

[builddirs]
/home/user/.conan/data/poco/1.9.4/_/_/package/58080bce1cc38259eb7c282aa95c25aecde8efe4/
/home/user/.conan/data/openssl/1.0.2t/_/_/package/
↪f99afdbf2a1cc98ba2029817b35103455b6a9b77/
/home/user/.conan/data/zlib/1.2.11/_/_/package/6af9cc7cb931c5ad942174fd7838eb655717c709/

[libs]
PocoMongoDB
PocoNetSSL
PocoNet
PocoCrypto
PocoDataSQLite
PocoData
PocoZip
```

(continues on next page)

(continued from previous page)

```
PocoUtil
PocoXML
PocoJSON
PocoRedis
PocoFoundation
rt
ssl
crypto
dl
pthread
z

[system_libs]


[defines]
POCO_STATIC=ON
POCO_NO_AUTOMATIC_LIBS
```

15.7.3 Use the Conan data model (in a *conanfile.py*)

If you are using any other build system you can use Conan too. In the `build()` method you can access your settings and build information from your requirements and pass it to your build system. Note, however, that probably is simpler and much more reusable to create a generator to simplify the task for your build system.

Listing 40: *conanfile.py*

```
from conans import ConanFile


class MyProjectWithConan(ConanFile):
    settings = "os", "compiler", "build_type", "arch"
    requires = "poco/1.9.4"
    ############ IT'S IMPORTANT TO DECLARE THE TXT GENERATOR TO DEAL WITH A GENERIC BUILD␣
↪SYSTEM
    generators = "txt"
    default_options = {"poco:shared": False, "openssl:shared": False}

    def imports(self):
        self.copy("*.dll", dst="bin", src="bin") # From bin to bin
        self.copy("*.dylib*", dst="bin", src="lib") # From lib to bin

    def build(self):
        ############ Without any helper ###########
        # Settings
        print(self.settings.os)
        print(self.settings.arch)
        print(self.settings.compiler)

        # Options
        #print(self.options.my_option)
```

(continues on next page)

(continued from previous page)

```
        print(self.options["openssl"].shared)
        print(self.options["poco"].shared)

        # Paths and libraries, all
        print("-------- ALL ---------------")
        print(self.deps_cpp_info.include_paths)
        print(self.deps_cpp_info.lib_paths)
        print(self.deps_cpp_info.bin_paths)
        print(self.deps_cpp_info.libs)
        print(self.deps_cpp_info.defines)
        print(self.deps_cpp_info.cflags)
        print(self.deps_cpp_info.cxxflags)
        print(self.deps_cpp_info.sharedlinkflags)
        print(self.deps_cpp_info.exelinkflags)

        # Just from OpenSSL
        print("--------- FROM OPENSSL -------------")
        print(self.deps_cpp_info["openssl"].include_paths)
        print(self.deps_cpp_info["openssl"].lib_paths)
        print(self.deps_cpp_info["openssl"].bin_paths)
        print(self.deps_cpp_info["openssl"].libs)
        print(self.deps_cpp_info["openssl"].defines)
        print(self.deps_cpp_info["openssl"].cflags)
        print(self.deps_cpp_info["openssl"].cxxflags)
        print(self.deps_cpp_info["openssl"].sharedlinkflags)
        print(self.deps_cpp_info["openssl"].exelinkflags)

        # Just from POCO
        print("--------- FROM POCO -------------")
        print(self.deps_cpp_info["poco"].include_paths)
        print(self.deps_cpp_info["poco"].lib_paths)
        print(self.deps_cpp_info["poco"].bin_paths)
        print(self.deps_cpp_info["poco"].libs)
        print(self.deps_cpp_info["poco"].defines)
        print(self.deps_cpp_info["poco"].cflags)
        print(self.deps_cpp_info["poco"].cxxflags)
        print(self.deps_cpp_info["poco"].sharedlinkflags)
        print(self.deps_cpp_info["poco"].exelinkflags)

        # self.run("invoke here your configure, make, or others")
        # self.run("basically you can do what you want with your requirements build␣
↪info)

        # Environment variables (from requirements self.env_info objects)
        # are automatically applied in the python ``os.environ`` but can be accesible as␣
↪well:
        print("--------- Globally -------------")
        print(self.env)

        print("--------- FROM MyLib -------------")
        print(self.deps_env_info["mylib"].some_env_var)
```

(continues on next page)

(continued from previous page)

```
        # User declared variables (from requirements self.user_info objects)
        # are available in the self.deps_user_info object
        print("--------- FROM MyLib -------------")
        print(self.deps_user_info["mylib"].some_user_var)
```

15.7.4 Create your own generator

There are two ways in which generators can be contributed:

- Forking and adding the new generator in the Conan codebase. This will be a built-in generator. It might have a much slower release and update cycle, it needs to pass some tests before being accepted, but it has the advantage than no extra things are needed to use that generator (once next Conan version is released).
- Creating a custom *generator package*. You can write a *conanfile.py* and add the custom logic for a generator inside that file, then upload, refer and depend on it as any other package. These generators will be another node in the dependency graph but they have many advantages: much faster release cycles, independent from the Conan codebase and can be versioned. So backwards compatibility and upgrades are much easier.

15.7.5 Extending Conan

There are other powerful mechanisms to integrate other tools with Conan. Check the *Extending Conan* section for further information.

15.8 Linting

You can develop your recipe and binary packages getting feedback of potential issues.

15.8.1 Linting the recipe

IDE

If you have an IDE that supports Python and may do linting automatically, there are false warnings caused by the fact that Conan dynamically populates some fields of the recipe based on context.

Conan provides a plugin which makes pylint aware of these dynamic fields and their types. To use it when running pylint outside Conan, just add the following to your `.pylintrc` file:

```
[MASTER]
load-plugins=conans.pylint_plugin
```

Hook

There is also a `recipe_linter` hook in the official hooks repository that can be activated to run automatic linter checks on the recipes when they are exported to the conan cache (`export`, `create` and `export-pkg` commands). Since Conan 1.47, it has also being added a checker for Conan `2.x` deprecated imports like `from conans import xxxxx` (you should use `from conan import xxxxx` instead) as part of the `recipe_linter` hook.

Read the hook documentation for details. You could also write your own custom linter hook to provide your own recipe quality checks.

15.8.2 Linting binary packages

Using the *Conan hooks* feature you can scan your binaries to ensure that you are generating the correct binary files and even checking the binary contents.

Take a look at the official hooks repository to see several examples of how to implement a binary linter system.

15.9 Deployment

If you have a project with all the dependencies managed by Conan and you want to deploy into a specific format, the process is the following:

- Extract the needed artifacts to a local directory either using the *deploy generator* or the *json generator*.
- Convert the artifacts (typically executables, shared libraries and assets) to a different deploy format. You will find the specific steps for some of the most common deploy technologies below.

15.9.1 System package manager

The Conan packages can be deployed using a system package manager. Usually this process is done by creating a folder structure with the needed files and bundling all of them into the file format specific to the system package manager of choice, like *.rpm* or *.deb*. This method is very convenient for deployment and distribution as it is natively integrated in the system. However, there are some limitations:

- It might require to create a specific package for each of supported distro, or at least use the lowest version (see concerns about `glibc` below), see the section *Customizing settings*, which explains how to customize Conan settings to model different Linux distributions in order to create different packages for them.
- If you want to target different distros, then you need to create one package per supported distro (likely one for Ubuntu, one for Arch Linux, etc.), and formats or guidelines for each distro might differ significantly

Check out the sections *makeself*, *AppImage*, *Flatpak* and *Snap* for information on how to create distribution-agnostic packages.

15.9.2 Makeself

Makeself is a small command-line utility to generate self-extracting archives for Unix. It is pretty popular and it is used by VirtualBox and CMake projects.

Makeself creates archives that are just small startup scripts (*.run*, *.bin* or *.sh*) concatenated with tarballs.

When you run such self-extracting archive:

- A small script (shim) extracts the embedded archive into the temporary directory
- Script passes the execution to the entry point within the unpacked archive
- application is being run
- The temporary directory removed

Therefore, it transparently appears just like a normal application execution.

With help of *deploy generator*, it's only needed to invoke `makeself.sh` in order to generate self-extracting archive for the further deployment:

```
TMPDIR=`dirname $(mktemp -u -t tmp.XXXXXXXXXX)`
curl "https://github.com/megastep/makeself/releases/download/release-2.4.0/makeself-2.4.
↪0.run" --output $TMPDIR/makeself.run -L
chmod +x $TMPDIR/makeself.run
$TMPDIR/makeself.run --target $TMPDIR/makeself
$TMPDIR/makeself/makeself.sh $PREFIX md5.run "conan-generated makeself.sh" "./conan-
↪entrypoint.sh"
```

The `PREFIX` variable in the example points to the directory where binary artifacts are situated. The `md5.run` is an output SFX archive:

```
$ file md5.run
md5.run: POSIX shell script executable (binary data)
```

The `conan-entry-point.sh` is a simple script which sets requires variables (like `PATH` or `LD_LIBRARY_PATH`):

```
#!/usr/bin/env bash
set -ex
export PATH=$PATH:$PWD/bin
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$PWD/lib
pushd $(dirname $PWD/md5)
$(basename $PWD/md5)
popd
```

Check out the complete example on GitHub.

15.9.3 AppImage

AppImage (former `klik`, `PortableLinuxApps`) is a format for Linux portable applications. Its major advantages are:

- It does not require root permissions.
- It does not require to install any application (it uses **`chmod +x`**).
- It does not require the installation of runtime or a daemon into the system.

AppImage might be used to distribute desktop applications, command-line tools and system services (daemons).

AppImage uses filesystem in user-space (FUSE). It allows to easily mount the images and inspect their contents.

The main steps of the packaging process are pretty straightforward and could be easily automated:

- Create a directory like `MyApp.AppDir`
- Download the AppImage runtime (**AppRun** file) and put it into the directory.
- Copy all dependency files, like libraries (*.so*), resources (e.g. images) inside the directory.
- Fill the *myapp.desktop* configuration file with some brief metadata of your application: name, category...
- Run **`appimagetool`**.

The copy step can be automatically done with Conan using the *json generator* and a custom script or just using the *deploy generator*.

The result of the previous steps will give you a *MyApp-x86_64.AppImage* file, which is a regular Linux ELF file:

```
$ file MyApp-x86_64.AppImage
MyApp-x86_64.AppImage: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically␣
↪linked, interpreter /lib64/l, for GNU/Linux 2.6.18, stripped
```

Finally, that file file could be easily distributed just by copying and uploading it to a Web or a FTP server, moving it to the flash drive, etc..

15.9.4 Snap

Snap is the package management system available for the wide range of Linux distributions. Unlike *AppImage*, Snap requires a daemon (`snapd`) installed in the system in order to operate. Under the hood, **Snap** is based on SquashFS. `Snap` is Canonical initiative. Usually, applications are distributed via snapcraft store, but it's not mandatory. `Snap` provides fine-grained control to system resources (e.g. camera, removable media, network, etc.). The major advantage is plug-in system, which allows to easily integrate `Snap` with different languages and build systems (e.g. CMake, autotools, etc.).

The packaging process could be summed up in the following steps:

- Install the snapcraft
- Run `snapcraft init`
- Edit the `snap/snapcraft.yml` manifest
- Run `snapcraft` in order to produce the snap
- Publish and upload snap, so it could be installed on other systems.

In order to integrate with build process managed with help of the conan, the following steps could be used:

- Use *deploy generator* (or *json generator* with custom script) to prepare the assets
- Use the dump plug-in of snapcraft to simply copy the files deployed on previous step into the `snap`

15.9.5 Flatpak

Flatpak (former `xdg-app`) is a package management system to distribute desktop applications for Linux. It is based on OSTree. Flatpak is RedHat initiative.

Unlike *AppImage*, usually applications are distributed via flathub store, and require a special runtime to install applications on target machines.

The major advantage of `Flatpak` is sandboxing: each application runs in its own isolated environment. `Flatpak` provides fine-grained control to system resources (e.g. network, bluetooth, host filesystem, etc.). `Flatpak` also offers a set of runtimes for various Linux desktop applications, e.g. Freedesktop, GNOME and KDE.

The packaging process is:

- Install the flatpak runtime, flatpak-builder and SDK.
- Create a manifest `<app-id>.json`
- Run the `flatpak-builder` in order to produce the application
- Publish the application for further distribution

With help of conan's *json generator*, the manifest creation could be easily automated. For example, the custom script could generate `build-commands` and `sources` entries within the manifest file:

```
app_id = "org.flatpak.%s" % self._name
manifest = {
    "app-id": app_id,
    "runtime": "org.freedesktop.Platform",
    "runtime-version": "18.08",
    "sdk": "org.freedesktop.Sdk",
    "command": "conan-entrypoint.sh",
    "modules": [
        {
            "name": self._name,
            "buildsystem": "simple",
            "build-commands": ["install -D conan-entrypoint.sh /app/bin/conan-entrypoint.
↪sh"],
            "sources": [
                {
                    "type": "file",
                    "path": "conan-entrypoint.sh"
                }
            ]
        }
    ]
}
sources = []
build_commands = []
for root, _, filenames in os.walk(temp_folder):
    for filename in filenames:
        filepath = os.path.join(root, filename)
        unique_name = str(uuid.uuid4())
        source = {
            "type": "file",
            "path": filepath,
            "dest-filename": unique_name
```

(continues on next page)

(continued from previous page)

```
        }
        build_command = "install -D %s /app/%s" % (unique_name, os.path.relpath(filepath,
↪ temp_folder))
        sources.append(source)
        build_commands.append(build_command)

manifest["modules"][0]["sources"].extend(sources)
manifest["modules"][0]["build-commands"].extend(build_commands)
```

Alternatively, `flatpak` allows distributing the single-file package. Such package, however, cannot be run or installed on its own, it's needed to be imported to the local repository on another machine.

CHAPTER

SIXTEEN

CONFIGURATION

The Conan client can be configured to behave differently. Most of the configuration can be found in the *conan.conf reference*, but this section aims to be an introduction to the configuration based on different use cases.

16.1 Download cache

Warning: This is an **experimental** feature subject to breaking changes in future releases.

Conan implements a shared download cache that can be used to reduce the time needed to populate the Conan package cache with commands like **`install`**, **`create`**.

This cache is purely an optimization mechanism. It is completely different to the *Conan package cache*, (typically the `<userhome>/.conan` folder). It is not related to the `short_paths` mechanism for long path in Windows, nor to the `short_paths` cache folder. The cache will contain a copy of the artifacts, it is not a new location of files. Those files will still be copied to the Conan package cache, which will not change anything, its behavior, layout or location of any file.

This cache (whose path can be configured in the *conan.conf* file) will store the following items:

- All files that are downloaded from a Conan server (conan_server, Artifactory), both in the api V1 (without revisions) and V2 (with revisions). This includes files like *conanfile.py*, but also the zipped artifacts like *conan_package.tgz* or *conan_sources.tgz*.
- The downloads done by users with the `tools.download()` or `tools.get()` helpers, as long as they provide a checksum (md5, sha1, etc.). If a checksum is not provided, even if the download cache is enabled, the download will be always executed and the files will not be cached.

Warning: The cache computes a sha256 checksum of the download URL and the file checksum whenever is available. As not always the file checksums are available, the download cache will not be able to correctly cache artifacts with revisions enabled if a proxy suddenly and transparently changes a existing server and moves it to a new location, without the clients changing the URL too.

16.1.1 Activating/deactivating the download cache

The download cache is activated and configured in the *conan.conf* like this:

```
[storage]
download_cache=/path/to/my/cache
```

It can be defined from the command line:

```
$ conan config set storage.download_cache="/path/to/my/cache"
# Display it
$ conan config get storage.download_cache
```

And, as the *conan.conf* is part of the configuration, you can also put a common *conan.conf* file in a git repo or zip file and use the *conan config install* command to automatically install it in Conan clients.

To deactivate the download cache, you can remove the entry `download_cache` from the *conan.conf* with the command:

```
$ conan config rm storage.download_cache
```

16.1.2 Concurrency, multiple caches and CI

The downloads cache implements exclusive locks for concurrency, so it can be shared among different concurrent Conan instances. This is a typical scenario in CI servers, in which each job uses a different Conan package cache (defined by `CONAN_USER_HOME` environment variable). Every different Conan instance could configure its download cache to share the same storage. The download cache implements interprocess exclusive locks, so only 1 process will access at a time to a given cached artifact. If other processes needs the same artifact, they will wait until it is released, avoiding multiple downloads of the same file, even if they were requested almost simultaneously.

For Continuous Integration processes, it is recommended to have a different Conan package cache (`CONAN_USER_HOME`) for each job, in most of the cases, because the Conan package cache is not concurrent, and it might also have old dependencies, stale packages, etc. It is better to run CI jobs in a clean environment.

16.1.3 Removing cached files

The download cache will store a lot of artifacts, for all recipes, packages, versions and configurations that are used. This can grow and consume a lot of storage. If you are using this feature, provide for a sufficiently large and fast download cache folder.

At the moment, it is only a folder. You can clean the cached artifacts just by removing that folder and its contents. You might also be able to run scripts and jobs that remove old artifacts only. If you do such operations, please make sure that there are not other Conan processes using it simultaneously, or they might fail.

Note: Installation of binaries can be accelerated setting up parallel downloads with the `general.parallel_download` **experimental** configuration in *conan.conf*. You might want to try combining both the parallel download and the download cache for extra speed.

CHAPTER

SEVENTEEN

HOWTOS

This section shows common solutions and different approaches to typical problems.

17.1 How to package header-only libraries

17.1.1 Without unit tests

Packaging a header only library, without requiring to build and run unit tests for it within Conan, can be done with a very simple recipe. Assuming you have the recipe in the source repo root folder, and the headers in a subfolder called `include`, you could do:

```
from conans import ConanFile

class HelloConan(ConanFile):
    name = "Hello"
    version = "0.1"
    # No settings/options are necessary, this is header only
    exports_sources = "include/*"
    no_copy_source = True

    def package(self):
        self.copy("*.h")
```

If you want to package an external repository, you can use the `source()` method to do a clone or download instead of the `exports_sources` fields.

- There is no need for `settings`, as changing them will not affect the final package artifacts
- There is no need for `build()` method, as header-only are not built
- There is no need for a custom `package_info()` method. The default one already adds an "include" subfolder to the include path
- `no_copy_source = True` will disable the copy of the source folder to the build directory as there is no need to do so because source code is not modified at all by the `configure()` or `build()` methods.
- Note that this recipe has no other dependencies, settings or options. If it had any of those, it would be very convenient to add the `package_id()` method, to ensure that only one package with always the same ID is created, irrespective of the configurations and dependencies:

```
def package_id(self):
    self.info.header_only()
```

Package is created with:

```
$ conan create . user/channel
```

17.1.2 With unit tests

If you want to run the library unit test while packaging, you would need this recipe:

```
from conans import ConanFile, CMake

class HelloConan(ConanFile):
    name = "Hello"
    version = "0.1"
    settings = "os", "compiler", "arch", "build_type"
    exports_sources = "include/*", "CMakeLists.txt", "example.cpp"
    no_copy_source = True

    def build(self): # this is not building a library, just tests
        cmake = CMake(self)
        cmake.configure()
        cmake.build()
        cmake.test()

    def package(self):
        self.copy("*.h")

    def package_id(self):
        self.info.header_only()
```

Tip: If you are *cross-building* your **library** or **app** you'll probably need to skip the **unit tests** because your target binary cannot be executed in current building host. To do it you can use *CONAN_RUN_TESTS* environment variable, defined as **False** in profile for cross-building in the call to `cmake.test()` this variable will be evaluated and the tests will not run.

Which will use a `CMakeLists.txt` file in the root folder:

```
project(Package CXX)
cmake_minimum_required(VERSION 2.8.12)

include_directories("include")
add_executable(example example.cpp)

enable_testing()
add_test(NAME example
        WORKING_DIRECTORY ${CMAKE_BINARY_DIR}/bin
        COMMAND example)
```

and some `example.cpp` file, which will be our "unit test" of the library:

```
#include <iostream>
#include "hello.h"
```

(continues on next page)

(continued from previous page)

```
int main() {
    hello();
}
```

- This will use different compilers and versions, as configured by Conan settings (in command line or profiles), but will always generate just 1 output package, always with the same ID.
- The necessary files for the unit tests, must be `exports_sources` too (or retrieved from `source()` method)
- If the package had dependencies, via `requires`, it would be necessary to add the `generators = "cmake"` to the package recipe and adding the `conanbuildinfo.cmake` file to the testing CMakeLists.txt:

```
include(${CMAKE_BINARY_DIR}/conanbuildinfo.cmake)
conan_basic_setup()

add_executable(example example.cpp)
target_link_libraries(example ${CONAN_LIBS}) # not necessary if dependencies are also␣
↪header-only
```

Package is created with:

```
$ conan create . user/channel
```

Note: This with/without tests is referring to running full unitary tests over the library, which is different to the **`test`** functionality that checks the integrity of the package. The above examples are describing the approaches for unit-testing the library within the recipe. In either case, it is recommended to have a *test_package* folder, so the **`conan create`** command checks the package once it is created. Check the *packaging getting started guide*

17.2 How to launch conan install from cmake

It is possible to launch **`conan install`** from cmake, which can be convenient for end users, package consumers, that are not creating packages themselves.

This is work under **testing**. Please try it and give feedback or contribute. The CMake code to do this task is here: https://github.com/conan-io/cmake-conan

To be able to use it, you can directly download the code from your CMake script:

Listing 1: *CMakeLists.txt*

```
cmake_minimum_required(VERSION 2.8)
project(myproject CXX)

# Download automatically, you can also just copy the conan.cmake file
if(NOT EXISTS "${CMAKE_BINARY_DIR}/conan.cmake")
   message(STATUS "Downloading conan.cmake from https://github.com/conan-io/cmake-conan")
   file(DOWNLOAD "https://raw.githubusercontent.com/conan-io/cmake-conan/master/conan.
↪cmake"
                 "${CMAKE_BINARY_DIR}/conan.cmake")
endif()
```

(continues on next page)

(continued from previous page)

```
include(${CMAKE_BINARY_DIR}/conan.cmake)

conan_cmake_run(REQUIRES Catch2/2.6.0@catchorg/stable
                BASIC_SETUP)

add_executable(main main.cpp)
target_link_libraries(main ${CONAN_LIBS})
```

If you want to use targets, you could do:

```
include(conan.cmake)
conan_cmake_run(REQUIRES Catch2/2.6.0@catchorg/stable
                BASIC_SETUP CMAKE_TARGETS)

add_executable(main main.cpp)
target_link_libraries(main CONAN_PKG::hello)
```

17.3 How to create and reuse packages based on Visual Studio

Conan has different helpers to manage Visual Studio and MSBuild based projects. This how-to illustrates how to put them together to create and consume packages that are purely based on Visual Studio. This how-to is using VS2015, but other versions can be used too.

17.3.1 Creating packages

Start cloning the existing example repository, containing a simple "Hello World" library, and application:

```
$ git clone https://github.com/memsharded/hello_vs
$ cd hello_vs
```

It contains a `src` folder with the source code and a `build` folder with a Visual Studio 2015 solution, containing 2 projects: the HelloLib static library, and the Greet application. Open it:

```
$ build\HelloLib\HelloLib.sln
```

You should be able to select the `Greet` subproject -> `Set as Startup Project`. Then build and run the app with Ctrl+F5. (Debug -> Start Without Debugging)

```
$ Hello World Debug!
# Switch IDE to Release mode, repeat
$ Hello World Release!
```

Because the `hello.cpp` file contains an `#ifdef _DEBUG` to switch between debug and release message.

In the repository, there is already a `conanfile.py` recipe:

```
from conans import ConanFile, MSBuild

class HelloConan(ConanFile):
```

(continues on next page)

(continued from previous page)

```
    name = "hello"
    version = "0.1"
    license = "MIT"
    url = "https://github.com/memsharded/hello_vs"
    settings = "os", "compiler", "build_type", "arch"
    exports_sources = "src/*", "build/*"

    def build(self):
        msbuild = MSBuild(self)
        msbuild.build("build/HelloLib/HelloLib.sln")

    def package(self):
        self.copy("*.h", dst="include", src="src")
        self.copy("*.lib", dst="lib", keep_path=False)

    def package_info(self):
        self.cpp_info.libs = ["HelloLib"]
```

This recipe is using the *MSBuild() build helper* to build the `sln` project. If our recipe has `requires`, the `MSBUILD` helper will also take care of inject all the needed information from the requirements, as include directories, library names, definitions, flags etc to allow our project to locate the declared dependencies.

The recipe contains also a `test_package` folder with a simple example consuming application. In this example, the consuming application is using CMake to build, but it could also use Visual Studio too. We have left the CMake one because it is the default generated with **conan new**, and also to show that packages created from Visual Studio projects can also be consumed with other build systems like CMake.

Once we want to create a package, it is advised to close VS IDE, clean the temporary build files from VS to avoid problems, then create and test the package. Here it is using system defaults, assuming they are Visual Studio 14, Release, x86_64:

```
# close VS
$ git clean -xdf
$ conan create . memsharded/testing
...
> Hello World Release!
```

Instead of closing the IDE and running the command:*git clean* we could also configure a smarter filter in `exports_sources` field, so temporary build files are not exported into the recipe.

This process can be repeated to create and test packages for different configurations:

```
$ conan create . memsharded/testing -s arch=x86
$ conan create . memsharded/testing -s compiler="Visual Studio" -s compiler.runtime=MDd -
↪s build_type=Debug
$ conan create . memsharded/testing -s compiler="Visual Studio" -s compiler.runtime=MDd -
↪s build_type=Debug -s arch=x86
```

Note: It is not mandatory to specify the `compiler.runtime` setting. If it is not explicitly defined, Conan will automatically use `runtime=MDd` for `build_type==Debug` and `runtime=MD` for `build_type==Release`.

You can list the different created binary packages:

```
$ conan search hello/0.1@memsharded/testing
```

17.3.2 Uploading binaries

Your locally created packages can already be uploaded to a Conan remote. If you created them with the original username "memsharded", as from the git clone, you might want to do a **conan copy** to put them on your own username. Of course, you can also directly use your user name in **conan create**.

Another alternative is to configure the permissions in the remote, to allow uploading packages with different usernames. By default, Artifactory will do it but Conan server won't: Permissions must be given in the `[write_permissions]` section of *server.conf* file.

17.3.3 Reusing packages

To use existing packages directly from Visual Studio, Conan provides the `visual_studio` generator. Let's clone an existing "Chat" project, consisting of a ChatLib static library that makes use of the previous "Hello World" package, and a MyChat application, calling the ChatLib library function.

```
$ git clone https://github.com/memsharded/chat_vs
$ cd chat_vs
```

As above, the repository contains a Visual Studio solution in the `build` folder. But if you try to open it, it will fail to load. This is because it is expecting to find a file with the required information about dependencies, so it is necessary to obtain that file first. Just run:

```
$ conan install .
```

You will see that it created two files, a `conaninfo.txt` file, containing the current configuration of dependencies, and a `conanbuildinfo.props` file, containing the Visual Studio properties (like `<AdditionalIncludeDirectories>`), so it is able to find the installed dependencies.

Now you can open the IDE and build and run the app (by the way, the chat function is just calling the `hello()` function two or three times, depending on the build type):

```
$  build\ChatLib\ChatLib.sln
# Switch to Release
# MyChat -> Set as Startup Project
# Ctrl + F5 (Debug -> Run without debugging)
> Hello World Release!
> Hello World Release!
```

If you wish to link with the debug version of Hello package, just install it and change IDE build type:

```
$ conan install . -s build_type=Debug -s compiler="Visual Studio" -s compiler.runtime=MDd
# Switch to Debug
# Ctrl + F5 (Debug -> Run without debugging)
> Hello World Debug!
> Hello World Debug!
> Hello World Debug!
```

Now you can close the IDE and clean the temporary files:

```
# close VS IDE
$ git clean -xdf
```

Again, there is a `conanfile.py` package recipe in the repository, together with a `test_package`. The recipe is almost identical to the above one, just with two minor differences:

```
requires = "hello/0.1@memsharded/testing"
...
generators = "visual_studio"
```

This will allow us to create and test the package of the ChatLib library:

```
$ conan create . memsharded/testing
> Hello World Release!
> Hello World Release!
```

You can also repeat the process for different build types and architectures.

17.3.4 Other configurations

The above example works as-is for VS2017, because VS supports upgrading from previous versions. The `MSBuild()` already implements such functionality, so building and testing packages with VS2017 can be done.

```
$ conan create . demo/testing -s compiler="Visual Studio" -s compiler.version=15
```

If you have to build for older versions of Visual Studio, it is also possible. In that case, you would probably have different solution projects inside your build folder. Then the recipe only has to select the correct one, something like:

```
def build(self):
    # assuming HelloLibVS12, HelloLibVS14 subfolders
    sln_file = "build/HelloLibVS%s/HelloLib.sln" % self.settings.compiler.version
    msbuild = MSBuild(self)
    msbuild.build(sln_file)
```

Finally, we used just one `conanbuildinfo.props` file, which the solution loaded at a global level. You could also define multiple `conanbuildinfo.props` files, one per configuration (Release/Debug, x86/x86_64), and load them accordingly.

Note: So far, the `visual_studio` generator is single-configuration (packages containing debug or release artifacts, the generally recommended approach). It does not support multi-config packages (packages containing both debug and release artifacts). Please report and provide feedback (submit an issue in github) to request this feature if necessary.

17.4 Creating and reusing packages based on Makefiles

Conan can create packages and reuse them with Makefiles. The `AutoToolsBuildEnvironment` build helper helps with most of the necessary tasks.

This how-to has been tested in Windows with MinGW and Linux with gcc. It uses static libraries but could be extended to shared libraries too. The Makefiles surely can be improved. They are just an example.

17.4.1 Creating packages

Sources for this example can be found in our examples repository in the *features/makefiles* folder:

```
$ git clone https://github.com/conan-io/examples.git
$ cd examples/features/makefiles
$ cd hellolib
```

It contains a *src* folder with the source code and a *conanfile.py* file for creating a package.

Inside the *src* folder, there is *Makefile* to build the static library. This *Makefile* uses standard variables like `$(CPPFLAGS)` or `$(CXX)` to build it:

```
SRC = hello.cpp
OBJ = $(SRC:.cpp=.o)
OUT = libhello.a
INCLUDES = -I.

.SUFFIXES: .cpp

default: $(OUT)

.cpp.o:
    $(CXX) $(INCLUDES) $(CPPFLAGS) $(CXXFLAGS) -c $< -o $@

$(OUT): $(OBJ)
    ar rcs $(OUT) $(OBJ)
```

The *conanfile.py* file uses the `AutoToolsBuildEnvironment` build helper. This helper defines the necessary environment variables with information from dependencies, as well as other variables to match the current Conan settings (like `-m32` or `-m64` based on the Conan `arch` setting)

```
from conans import ConanFile, AutoToolsBuildEnvironment
from conans import tools

class HelloConan(ConanFile):
    name = "hello"
    version = "0.1"
    settings = "os", "compiler", "build_type", "arch"
    generators = "cmake"
    exports_sources = "src/*"

    def build(self):
        with tools.chdir("src"):
            atools = AutoToolsBuildEnvironment(self)
```

(continues on next page)

(continued from previous page)

```
            # atools.configure() # use it to run "./configure" if using autotools
            atools.make()

    def package(self):
        self.copy("*.h", dst="include", src="src")
        self.copy("*.lib", dst="lib", keep_path=False)
        self.copy("*.a", dst="lib", keep_path=False)

    def package_info(self):
        self.cpp_info.libs = ["hello"]
```

With this *conanfile.py* you can create the package:

```
$ conan create . user/testing -s compiler=gcc -s compiler.version=4.9 -s compiler.
↪libcxx=libstdc++
```

17.4.2 Using packages

Now let's move to the application folder:

```
$ cd ../helloapp
```

There you can also see a *src* folder with a *Makefile* creating an executable:

```
SRC = app.cpp
OBJ = $(SRC:.cpp=.o)
OUT = app
INCLUDES = -I.

.SUFFIXES: .cpp

default: $(OUT)

.cpp.o:
    $(CXX) $(CPPFLAGS) $(CXXFLAGS) -c $< -o $@

$(OUT): $(OBJ)
    $(CXX) -o $(OUT)  $(OBJ)  $(LDFLAGS)  $(LIBS)
```

And also a *conanfile.py* very similar to the previous one. In this case adding a `requires` and a `deploy()` method:

```
from conans import ConanFile, AutoToolsBuildEnvironment
from conans import tools

class AppConan(ConanFile):
    name = "app"
    version = "0.1"
    settings = "os", "compiler", "build_type", "arch"
    exports_sources = "src/*"
    requires = "hello/0.1@user/testing"
```

(continues on next page)

(continued from previous page)

```
    def build(self):
        with tools.chdir("src"):
            atools = AutoToolsBuildEnvironment(self)
            atools.make()

    def package(self):
        self.copy("*app", dst="bin", keep_path=False)
        self.copy("*app.exe", dst="bin", keep_path=False)

    def deploy(self):
        self.copy("*", src="bin", dst="bin")
```

Note that in this case, the `AutoToolsBuildEnvironment` will automatically set values to `CPPFLAGS`, `LDFLAGS`, `LIBS`, etc. existing in the *Makefile* with the correct include directories, library names, etc. to properly build and link with the `hello` library contained in the "hello" package.

As above, we can create the package with:

```
$ conan create . user/testing -s compiler=gcc -s compiler.version=4.9 -s compiler.
↪libcxx=libstdc++
```

There are different ways to run executables contained in packages, like using `virtualrunenv` generators. In this case, since the package has a `deploy()` method, we can use it:

```
$ conan install app/0.1@user/testing -s compiler=gcc -s compiler.version=4.9 -s compiler.
↪libcxx=libstdc++
$ ./bin/app
$ Hello World Release!
```

17.5 How to manage the GCC >= 5 ABI

In version 5.1, GCC released libstdc++, which introduced a new library ABI that includes new implementations of `std::string` and `std::list`. These changes were necessary to conform to the 2011 C++ standard which forbids Copy-On-Write strings and requires lists to keep track of their size.

You can choose which ABI to use in your Conan packages by adjusting the `compiler.libcxx`:

- **libstdc++**: Old ABI.
- **libstdc++11**: New ABI.

When Conan creates the default profile the first time it runs, it adjusts the `compiler.libcxx` setting to `libstdc++` for backwards compatibility. However, if you are using GCC >= 5 your compiler is likely to be using the new CXX11 ABI by default (libstdc++11). This can be checked with the following command:

```
$ gcc -v 2>&1 | sed -n 's/.*\(--with-default-libstdcxx-abi=new\).*/\1/p'
--with-default-libstdcxx-abi=new
```

If you want Conan to use the new ABI, edit the default profile at `~/.conan/profiles/default` adjusting `compiler.libcxx=libstdc++11` or override this setting in the profile you are using.

If you are using the *CMake build helper* or the *AutotoolsBuildEnvironment build helper* Conan will automatically adjust the `_GLIBCXX_USE_CXX11_ABI` flag to manage the ABI.

17.6 Using Visual Studio 2017 - CMake integration

Visual Studio 2017 comes with a CMake integration that allows one to just open a folder that contains a *CMakeLists.txt* and Visual will use it to define the project build.

Conan can also be used in this setup to install dependencies. Let`s say that we are going to build an application that depends on an existing Conan package called `hello/0.1@user/testing`. For the purpose of this example, you can quickly create this package by typing in your terminal:

```
$ conan new hello/0.1 -s
$ conan create . user/testing # Default conan profile is Release
$ conan create . user/testing -s build_type=Debug
```

The project we want to develop will be a simple application with these 3 files in the same folder:

Listing 2: **example.cpp**

```
#include <iostream>
#include "hello.h"

int main() {
    hello();
    std::cin.ignore();
}
```

Listing 3: **conanfile.txt**

```
[requires]
hello/0.1@user/testing

[generators]
cmake
```

Listing 4: **CMakeLists.txt**

```
project(Example CXX)
cmake_minimum_required(VERSION 2.8.12)

include(${CMAKE_BINARY_DIR}/conanbuildinfo.cmake)
conan_basic_setup()

add_executable(example example.cpp)
target_link_libraries(example ${CONAN_LIBS})
```

If we open Visual Studio 2017 (with CMake support installed), and select "Open Folder" from the menu, and select the above folder, we will see something like the following error:

```
1> Command line: C:\PROGRAM FILES (X86)\MICROSOFT VISUAL STUDIO\2017\COMMUNITY\COMMON7\
↪IDE\COMMONEXTENSIONS\MICROSOFT\CMAKE\CMake\bin\cmake.exe  -G "Ninja" -DCMAKE_INSTALL_
↪PREFIX:PATH="C:\Users\user\CMakeBuilds\df6639d2-3ef2-bc32-abb3-2cd1bdb3c1ab\install\
↪x64-Debug"  -DCMAKE_CXX_COMPILER="C:/Program Files (x86)/Microsoft Visual Studio/2017/
↪Community/VC/Tools/MSVC/14.12.25827/bin/HostX64/x64/cl.exe"  -DCMAKE_C_COMPILER="C:/
↪Program Files (x86)/Microsoft Visual Studio/2017/Community/VC/Tools/MSVC/14.12.25827/
↪bin/HostX64/x64/cl.exe"  -DCMAKE_BUILD_TYPE="Debug" -DCMAKE_MAKE_PROGRAM="C:\PROGRAM␣
```

(continues on next page)

(continued from previous page)

```
↪FILES (X86)\MICROSOFT VISUAL STUDIO\2017\COMMUNITY\COMMON7\IDE\COMMONEXTENSIONS\
↪MICROSOFT\CMAKE\Ninja\ninja.exe" "C:\Users\user\conanws\visual-cmake"
1> Working directory: C:\Users\user\CMakeBuilds\df6639d2-3ef2-bc32-abb3-2cd1bdb3c1ab\
↪build\x64-Debug
1> -- The CXX compiler identification is MSVC 19.12.25831.0
1> -- Check for working CXX compiler: C:/Program Files (x86)/Microsoft Visual Studio/
↪2017/Community/VC/Tools/MSVC/14.12.25827/bin/HostX64/x64/cl.exe
1> -- Check for working CXX compiler: C:/Program Files (x86)/Microsoft Visual Studio/
↪2017/Community/VC/Tools/MSVC/14.12.25827/bin/HostX64/x64/cl.exe -- works
1> -- Detecting CXX compiler ABI info
1> -- Detecting CXX compiler ABI info - done
1> -- Detecting CXX compile features
1> -- Detecting CXX compile features - done
1> CMake Error at CMakeLists.txt:4 (include):
1>   include could not find load file:
1>
1>     C:/Users/user/CMakeBuilds/df6639d2-3ef2-bc32-abb3-2cd1bdb3c1ab/build/x64-Debug/
↪conanbuildinfo.cmake
```

As expected, our *CMakeLists.txt* is using an `include(${CMAKE_BINARY_DIR}/conanbuildinfo.cmake)`, and that file doesn't exist yet, because Conan has not yet installed the dependencies of this project. Visual Studio 2017 uses different build folders for each configuration. In this case, the default configuration at startup is `x64-Debug`. This means that we need to install the dependencies that match this configuration. Assuming that our default profile is using Visual Studio 2017 for x64 (it should typically be the default one created by Conan if VS2017 is present), then all we need to specify is the `-s build_type=Debug` setting:

```
$ conan install . -s build_type=Debug -if=C:\Users\user\CMakeBuilds\df6639d2-3ef2-bc32-
↪abb3-2cd1bdb3c1ab\build\x64-Debug
```

Now, you should be able to regenerate the CMake project from the IDE, Menu->CMake, build it, select the "example" executable to run, and run it.

Now, let's say that you want to build the Release application. You switch configuration from the IDE, and then the above error happens again. The dependencies for Release mode need to be installed too:

```
$ conan install . -if=C:\Users\user\CMakeBuilds\df6639d2-3ef2-bc32-abb3-2cd1bdb3c1ab\
↪build\x64-Release
```

The process can be extended to x86 (passing `-s arch=x86` in the command line), or to other configurations. For production usage, Conan **profiles** are highly recommended.

17.6.1 Using cmake-conan

The **cmake-conan** project in https://github.com/conan-io/cmake-conan is a CMake script that runs an `execute_process` that automatically launches **`conan install`** to install dependencies. The settings passed in the command line will be derived from the current CMake configuration, that will match the Visual Studio one. This script can be used to further automate the installation task:

```
project(Example CXX)
cmake_minimum_required(VERSION 2.8.12)

# Download automatically, you can also just copy the conan.cmake file
```

(continues on next page)

(continued from previous page)

```
if(NOT EXISTS "${CMAKE_BINARY_DIR}/conan.cmake")
message(STATUS "Downloading conan.cmake from https://github.com/conan-io/cmake-conan")
    file(DOWNLOAD "https://raw.githubusercontent.com/conan-io/cmake-conan/v0.9/conan.
↪cmake"
                  "${CMAKE_BINARY_DIR}/conan.cmake")
endif()

include(${CMAKE_BINARY_DIR}/conan.cmake)

conan_cmake_run(CONANFILE conanfile.txt
                BASIC_SETUP)

add_executable(example example.cpp)
target_link_libraries(example ${CONAN_LIBS})
```

This code will manage to download the **cmake-conan** CMake script, and use it automatically, calling a **conan install** automatically.

There could be an issue, though, for the `Release` configuration. Internally, the Visual Studio 2017 defines the `configurationType` As `RelWithDebInfo` for `Release` builds. But Conan default settings (in the Conan *settings.yml* file), only have `Debug` and `Release` defined. It is possible to modify the *settings.yml* file, and add those extra build types. Then you should create the `hello` package for those settings. And most existing packages, specially in central repositories, are built only for Debug and Release modes.

An easier approach is to change the CMake configuration in Visual: go to the Menu -> CMake -> Change CMake Configuration. That should open the *CMakeSettings.json* file, and there you can change the `configurationType` to `Release`:

```
{
    "name": "x64-Release",
    "generator": "Ninja",
    "configurationType": "Release",
    "inheritEnvironments": [ "msvc_x64_x64" ],
    "buildRoot": "${env.USERPROFILE}\\CMakeBuilds\\${workspaceHash}\\build\\${name}",
    "installRoot": "${env.USERPROFILE}\\CMakeBuilds\\${workspaceHash}\\install\\${name}
↪",
    "cmakeCommandArgs": "",
    "buildCommandArgs": "-v",
    "ctestCommandArgs": ""
}
```

Note that the above CMake code is only valid for consuming existing packages. If you are also creating a package, you would need to make sure the right CMake code is executed, please check https://github.com/conan-io/cmake-conan/blob/master/README.md

17.6.2 Using tasks with tasks.vs.json

Another alternative is using file tasks feature of Visual Studio 2017. This way you can install dependencies by running **conan install** as task directly in the IDE.

All you need is to right click on your *conanfile.py* -> Configure Tasks (see the link above) and add the following to your *tasks.vs.json*.

Warning: The file *tasks.vs.json* is added to your local *.vs* folder so it is not supposed to be added to your version control system.

```
{
    "tasks": [
        {
        "taskName": "conan install debug",
        "appliesTo": "conanfile.py",
        "type": "launch",
        "command": "${env.COMSPEC}",
        "args": [
            "conan install ${file} -s build_type=Debug -if C:/Users/user/CMakeBuilds/
↪4c2d87b9-ec5a-9a30-a47a-32ccb6cca172/build/x64-Debug/"
        ]
        },
        {
        "taskName": "conan install release",
        "appliesTo": "conanfile.py",
        "type": "launch",
        "command": "${env.COMSPEC}",
        "args": [
            "conan install ${file} -s build_type=Release -if C:/Users/user/CMakeBuilds/
↪4c2d87b9-ec5a-9a30-a47a-32ccb6cca172/build/x64-Release/"
        ]
        }
    ],
    "version": "0.2.1"
}
```

Then just right click on your *conanfile.py* and launch your **conan install** and regenerate your *CMakeLists.txt*.

17.7 Working with Intel compilers

17.7.1 intel

Note: This compiler is aimed to manage legacy Intel Parallel Studio XE compiler versions. For new Intel oneAPI, check the information about the `intel-cc` compiler below.

The `Intel` compiler is a particular case, as it uses `Visual Studio` compiler in Windows environments and `gcc` in Linux environments. If you are wondering how to manage the compatibility between the packages generated with

`intel` and the generated with the pure base compiler (`gcc` or `Visual Studio`) check the *Compatible Packages* and *Compatible Compilers* sections.

17.7.2 intel-cc

Warning: The support for this compiler is **experimental** and subject to breaking changes.

Available since: 1.41.0

This new compiler is defined to manage the different Intel oneAPI DPC++/C++ and Classic ones.

Warning: macOS is not supported for the Intel oneAPI DPC++/C++ (icx/icpx or dpcpp) compilers. For macOS or Xcode support, you'll have to use the Intel C++ Classic Compiler.

It can be declared into your local profile like any other compiler as follows:

Listing 5: intelprofile

```
[settings]
...
compiler=intel-cc
compiler.mode=dpcpp
compiler.version=2021.3
compiler.libcxx=libstdc++
build_type=Release
[options]

[tool_requires]
[env]
CC=dpcpp
CXX=dpcpp

[conf]
tools.intel:installation_path=/opt/intel/oneapi
```

Important: Remember to put this `[conf]` entry to find out the root path of your Intel oneAPI folder. Normally, it'll be installed by default in either `/opt/intel/oneapi` (Linux and macOS) or `C:\Program Files (x86)\Intel\oneAPI` (Windows).

We're specifying the `CC` and `CXX` compilers and the `compiler.mode` subsetting. The possible values for `compiler.mode` are:

- `icx` for Intel oneAPI C++ (icx/icpx compilers).
- `dpcpp` for Intel oneAPI DPC++ (dpcpp compiler and dpcpp-cl for Windows only).
- `classic` for Intel C++ Classic (icc for Linux and icl for Windows).

To set up the compiler **without Conan** you need to run an Intel official script to set all the proper variables to use those compilers called `setvars.sh|bat` script.

If you are using either the `CMakeToolChain` or the `MSBuildToolchain`, when using the `intel-cc` compiler, Conan automatically calls the `setvars` script. Otherwise, you can use the *IntelCC generator*.

This is an example of a Conan package called `hello/1.0` using the `CMakeToolchain`. Remember you can use the command **`conan new hello/1.0 -m cmake_lib`** to create a simple project like this one:

Listing 6: conanfile.py

```
from conans import ConanFile
from conan.tools.cmake import CMakeToolchain

class HelloConan(ConanFile):
    name = "hello"
    version = "1.0"

    # more code here...

    def generate(self):
        tc = CMakeToolchain(self)
        tc.generate()
```

Running **`conan create . -pr intelprofile -pr:b intelprofile`**, you'll see something like this output:

Listing 7: output

```
......
hello/1.0: Generating the package
hello/1.0: Package folder /home/franchuti/.conan/data/hello/1.0/_/_/package/
↪7d9c7d5fa3c48c9705c2cb864656c00fa8672524
hello/1.0: Calling package()
hello/1.0: CMake command: cmake --build '/home/franchuti/.conan/data/hello/1.0/_/_/build/
↪7d9c7d5fa3c48c9705c2cb864656c00fa8672524/cmake-build-release' '--target' 'install'
:: initializing oneAPI environment ...
   dash: SH_VERSION = unknown
:: advisor -- latest
:: ccl -- latest
:: clck -- latest
:: compiler -- latest
:: dal -- latest
:: debugger -- latest
:: dev-utilities -- latest
:: dnnl -- latest
:: dpcpp-ct -- latest
:: dpl -- latest
:: inspector -- latest
:: intelpython -- latest
:: ipp -- latest
:: ippcp -- latest
:: ipp -- latest
:: itac -- latest
:: mkl -- latest
:: mpi -- latest
:: tbb -- latest
:: vpl -- latest
```

(continues on next page)

(continued from previous page)

```
:: vtune -- latest
:: oneAPI environment initialized ::
Using Conan toolchain through /home/franchuti/.conan/data/hello/1.0/_/_/build/
↪7d9c7d5fa3c48c9705c2cb864656c00fa8672524/cmake-build-release/conan/conan_toolchain.
↪cmake.
-- Conan toolchain: Setting CMAKE_POSITION_INDEPENDENT_CODE=ON (options.fPIC)
-- Conan toolchain: Setting BUILD_SHARED_LIBS= OFF
-- The CXX compiler identification is Clang 13.0.0
-- Check for working CXX compiler: /opt/intel/oneapi/compiler/2021.3.0/linux/bin/dpcpp
Using Conan toolchain through .
-- Check for working CXX compiler: /opt/intel/oneapi/compiler/2021.3.0/linux/bin/dpcpp --
↪ works
-- Detecting CXX compiler ABI info
Using Conan toolchain through .
-- Detecting CXX compiler ABI info - done
-- Detecting CXX compile features
-- Detecting CXX compile features - done
-- Configuring done
-- Generating done
.......
```

As you can observe, you have used one of these Intel compilers, the DPC++ one and successfully generated the `libhello.a` file.

intel-cc and Microsoft Visual Studio

Note: Ensure you have installed the Intel plugins for Microsoft Visual Studio before reading this section.

If you're working on a Microsoft Visual Studio project, you can add the Intel Toolset as a new *.props* file. Let's suppose you have defined these files into your current project folder:

Listing 8: intelprofile

```
[settings]
os=Windows
os_build=Windows
arch=x86_64
arch_build=x86_64
compiler=intel-cc
compiler.mode=classic
compiler.version=2021.3
compiler.runtime=dynamic
build_type=Release
[options]
[tool_requires]
[env]
[conf]
tools.intel:installation_path="C:\Program Files (x86)\Intel\oneAPI"
```

Listing 9: conanfile.py

```
from conans import ConanFile
from conan.tools.microsoft import MSBuildToolchain

class App(ConanFile):
    settings = "os", "arch", "compiler", "build_type"

    def generate(self):
        tc = MSBuildToolchain(self)
        tc.generate()
```

Running a **conan install . -pr intelprofile**, a file *conantoolchain_release_x64.props* is generated in your current folder:

Listing 10: conantoolchain_release_x64.props

```
<?xml version="1.0" encoding="utf-8"?>
<Project xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
  <ItemDefinitionGroup>
    <ClCompile>
      <PreprocessorDefinitions>
         ;%(PreprocessorDefinitions)
      </PreprocessorDefinitions>
      <RuntimeLibrary>MultiThreadedDLL</RuntimeLibrary>
      <LanguageStandard></LanguageStandard>
    </ClCompile>
  </ItemDefinitionGroup>
  <PropertyGroup Label="Configuration">
    <PlatformToolset>Intel C++ Compiler 19.2</PlatformToolset>
  </PropertyGroup>
</Project>
```

Note that a `PlatformToolset` is set to `Intel C++ Compiler 19.2`. You can import that file to your project or solution of Visual Studio. Read more about the *MSBuildToolchain here*.

Note: See the complete *IntelCC reference* for more information about that tool.

17.8 How to manage C++ standard [EXPERIMENTAL]

Warning: This is an **experimental** feature subject to breaking changes in future releases. Previously, it was implemented as a first level setting `cppstd`, we encourage you to adopt the new subsetting and update your recipes if they were including the deprecated one in its *settings* attribute.

The setting representing the C++ standard is `compiler.cppstd`. The detected default profile doesn't set any value for the `compiler.cppstd` setting,

The consumer can specify it in a *profile* or with the `-s` parameter:

```
conan install . -s compiler.cppstd=gnu14
```

As it is a subsetting, it can have different values for each compiler (also, take into account that depending on the version of the compiler the standard could have only partial support and may change the ABI).

Valid values for `compiler=Visual Studio`:

VALUE	DESCRIPTION
14	C++ 14
17	C++ 17
20	C++ 20 (Still C++20 Working Draft)

Valid values for other compilers:

VALUE	DESCRIPTION
98	C++ 98
gnu98	C++ 98 with GNU extensions
11	C++ 11
gnu11	C++ 11 with GNU extensions
14	C++ 14
gnu14	C++ 14 with GNU extensions
17	C++ 17
gnu17	C++ 17 with GNU extensions
20	C++ 20 (Partial support)
gnu20	C++ 20 with GNU extensions (Partial support)

17.8.1 Build helpers

The value of `compiler.cppstd` provided by the consumer is used by the build helpers:

- The *CMake* build helper will set the `CONAN_CMAKE_CXX_STANDARD` and `CONAN_CMAKE_CXX_EXTENSIONS` definitions that will be converted to the corresponding CMake variables to activate the standard automatically with the `conan_basic_setup()` macro.
- The *AutotoolsBuildEnvironment* build helper will adjust the needed flag to `CXXFLAGS` automatically.
- The *MSBuild/VisualStudioBuildEnvironment* build helper will adjust the needed flag to `CL` env var automatically.

17.8.2 Package compatibility

By default Conan will detect the default standard of your compiler to not generate different binary packages. For example, you already built some `gcc 6.1` packages, where the default C++ standard is `gnu14`. If you introduce the `compiler.cppstd` setting in your profile with the `gnu14` value, Conan won't generate new packages, because it was already the default of your compiler.

Note: Check the *package_id()* reference to know more.

Note: Conan 1.x will also generate the same packages as the ones generated with the deprecated setting `cppstd` for the default value of the standard.

17.8.3 Required version

When the package to be built requires a minimal C++ standard support (e.g. 17), it can be done by comparing the `cppstd`. For such condition, there is the helper *check_min_cppstd*.

17.9 How to use Docker to create C and C++ Conan packages

With Docker, you can run different virtual Linux operating systems in a Linux, Mac OSX or Windows machine. It is useful to reproduce build environments, for example to automate CI processes. You can have different images with different compilers or toolchains and run containers every time is needed.

In this section you will find a *list of pre-built images* with common build tools and compilers as well as Conan installed.

17.9.1 Using Conan inside a container

```
$ docker run -it --rm --name conangcc11 conanio/gcc11-ubuntu16.04 /bin/bash
```

Note: Use `sudo` when needed to run `docker`.

The previous code will run a shell in container. We have specified:

- **`-it`**: Keep STDIN open and allocate a pseudo-tty, in other words, we want to type in the container because we are opening a bash.
- **`--rm`**: Once the container exits, remove the container. Helps to keep clean or hard drive.
- **`--name conangcc11`**`: The Docker container name
- **`conanio/gcc11-ubuntu16.04`**: Image name, check the *available Docker images*.
- **`/bin/bash`**: The command to run

Now we are running on the conangcc11 container we can use Conan normally. In the following example we are creating a package from the recipe by cloning the repository, for OpenSSL. It is always recommended to upgrade Conan from pip first:

```
$ pip install conan --upgrade # We make sure we are running the latest Conan version
$ git clone https://github.com/conan-io/conan-center-index
$ cd conan-center-index/recipes/openssl/1.x.x
$ conan create . 1.1.1n@
```

17.9.2 Sharing a local folder with a Docker container

You can share a local folder with your container, for example a project:

```
$ git clone https://github.com/conan-io/conan-center-index
$ cd conan-center-index/recipes/openssl/1.x.x
$ docker run -it -v$(pwd):/home/conan/project --rm conanio/gcc11-ubuntu16.04 /bin/bash
```

- `v$(pwd):/home/conan/project`: We are mapping the current directory (conan-openssl) to the container `/home/conan/project` directory, so anything we change in this shared folder, will be reflected in our host machine.

```
# Now we are running on the conangcc11 container
$ pip install conan --upgrade # We make sure we are running the latest Conan version
$ cd project
$ conan create . user/channel --build missing
$ conan remote add myremote http://some.remote.url
$ conan upload "*" -r myremote --all
```

17.9.3 Available Docker images

We provide a set of images with the most common compilers installed that can be used to generate Conan packages for different profiles. Their dockerfiles can be found in the Conan Docker Tools repository.

Warning: The images listed below are intended for generating open-source library packages and we cannot guarantee any kind of stability. We strongly recommend using your own generated images for production environments taking these dockerfiles as a reference.

GCC images

Version	Target Arch
conanio/gcc5-ubuntu16.04 (GCC 5)	x86_64
conanio/gcc6-ubuntu16.04 (GCC 6)	x86_64
conanio/gcc7-ubuntu16.04 (GCC 7)	x86_64
conanio/gcc8-ubuntu16.04 (GCC 8)	x86_64
conanio/gcc9-ubuntu16.04 (GCC 9)	x86_64
conanio/gcc10-ubuntu16.04 (GCC 10)	x86_64
conanio/gcc11-ubuntu16.04 (GCC 11)	x86_64

Clang images

Version	Target Arch
conanio/clang10-ubuntu16.04 (Clang 10)	x86_64
conanio/clang11-ubuntu16.04 (Clang 11)	x86_64
conanio/clang12-ubuntu16.04 (Clang 12)	x86_64
conanio/clang13-ubuntu16.04 (Clang 13)	x86_64
conanio/clang14-ubuntu16.04 (Clang 14)	x86_64

17.10 How to reuse Python code in recipes

Warning: To reuse Python code, from Conan 1.7 there is a new `python_requires()` feature. See: *Python requires: reusing Python code in recipes* This "how to" might be deprecated and removed in the future. It is left here for reference only.

First, if you feel that you are repeating a lot of Python code, and that repeated code could be useful for other Conan users, please propose it in a github issue.

There are several ways to handle Python code reuse in package recipes:

- To put common code in files, as explained *below*. This code has to be exported into the recipe itself.
- To create a Conan package with the common Python code, and then `require` it from the recipe.

This howto explains the latter.

17.10.1 A basic Python package

Let's begin with a simple Python package, a "hello world" functionality that we want to package and reuse:

```
def hello():
    print("Hello World from Python!")
```

To create a package, all we need to do is create the following layout:

```
-| hello.py
 | __init__.py
 | conanfile.py
```

The `__init__.py` is blank. It is not necessary to compile code, so the package recipe `conanfile.py` is quite simple:

```
from conans import ConanFile

class HelloPythonConan(ConanFile):
    name = "hello_py"
    version = "0.1"
    exports = '*'
    build_policy = "missing"

    def package(self):
        self.copy('*.py')

    def package_info(self):
        self.env_info.PYTHONPATH.append(self.package_folder)
```

The `exports` will copy both the `hello.py` and the `__init__.py` into the recipe. The `package()` method is also obvious: to construct the package just copy the Python sources.

The `package_info()` adds the current package folder to the `PYTHONPATH` Conan environment variable. It will not affect the real environment variable unless the end user desires it.

It can be seen that this recipe would be practically the same for most Python packages, so it could be factored in a `PythonConanFile` base class to further simplify it. (Open a feature request, or better a pull request. :))

With this recipe, all we have to do is:

```
$ conan export . memsharded/testing
```

Of course if you want to share the package with your team, you can **conan upload** it to a remote server. But to create and test the package, we can do everything locally.

Now the package is ready for consumption. In another folder, we can create a *conanfile.txt* (or a *conanfile.py* if we prefer):

```
[requires]
hello_py/0.1@memsharded/testing
```

And install it with the following command:

```
$ conan install . -g virtualenv
```

Creating the above `conanfile.txt` might be unnecessary for this simple example, as you can directly run **conan install hello_py/0.1@memsharded/testing -g virtualenv**, however, using the file is the canonical way.

The specified `virtualenv` generator will create an `activate` script (in Windows *activate.bat*), that basically contains the environment, in this case, the `PYTHONPATH`. Once we activate it, we are able to find the package in the path and use it:

```
$ activate
$ python
Python 3.6.1 (3.6.1:d33e0cf91556, Jun 27 2016, 15:19:22) [MSC v.1500 32 bit (Intel)] on␣
↪win32
...
>>> import hello
>>> hello.hello()
Hello World from Python!
>>>
```

The above shows an interactive session, but you can import also the functionality in a regular Python script.

17.10.2 Reusing Python code in your recipes

Requiring a Python Conan package

As the Conan recipes are Python code itself, it is easy to reuse Python packages in them. A basic recipe using the created package would be:

```
from conans import ConanFile

class HelloPythonReuseConan(ConanFile):
    requires = "hello_py/0.1@memsharded/testing"

    def build(self):
        from hello import hello
        hello()
```

The `requires` section is just referencing the previously created package. The functionality of that package can be used in several methods of the recipe: `source()`, `build()`, `package()` and `package_info()`, i.e. all of the methods used for creating the package itself. Note that in other places it is not possible, as it would require the dependencies of the

recipe to be already retrieved, and such dependencies cannot be retrieved until the basic evaluation of the recipe has been executed.

```
$ conan install .
...
$ conan build .
Hello World from Python!
```

Sharing a Python module

Another approach is sharing a Python module and exporting within the recipe.

Let's write for example a `msgs.py` file and put it besides the `conanfile.py`:

```
def build_msg(output):
    output.info("Building!")
```

And then the main `conanfile.py` would be:

```
from conans import ConanFile
from msgs import build_msg

class ConanFileToolsTest(ConanFile):
    name = "test"
    version = "1.9"
    exports = "msgs.py"  # Important to remember!

    def build(self):
        build_msg(self.output)
        # ...
```

It is important to note that such `msgs.py` file **must be exported** too when exporting the package, because package recipes must be self-contained.

The code reuse can also be done in the form of a base class, something like a file `base_conan.py`

```
from conans import ConanFile

class ConanBase(ConanFile):
    # common code here
```

And then:

```
from conans import ConanFile
from base_conan import ConanBase

class ConanFileToolsTest(ConanBase):
    name = "test"
    version = "1.9"
    exports = "base_conan.py"
```

17.11 How to create and share a custom generator with generator packages

There are several built-in generators, like `cmake`, `visual_studio`, `xcode`... But what if your build system is not included or the existing built-in ones doesn't satisfy your needs? This **how to** will show you how to create a generator for Premake build system.

Important: Check the reference of the custom_generator section to know the syntax and attributes available.

17.11.1 Creating a Premake generator

Create a folder with a new *conanfile.py* with the following contents:

```
$ mkdir conan-premake && cd conan-premake
```

Listing 11: *conanfile.py*

```
from conans.model import Generator
from conans import ConanFile


class PremakeDeps(object):
    def __init__(self, deps_cpp_info):
        self.include_paths = ",\n".join('"%s"' % p.replace("\\", "/")
                                        for p in deps_cpp_info.include_paths)
        self.lib_paths = ",\n".join('"%s"' % p.replace("\\", "/")
                                    for p in deps_cpp_info.lib_paths)
        self.bin_paths = ",\n".join('"%s"' % p.replace("\\", "/")
                                    for p in deps_cpp_info.bin_paths)
        self.libs = ", ".join('"%s"' % p for p in deps_cpp_info.libs)
        self.defines = ", ".join('"%s"' % p for p in deps_cpp_info.defines)
        self.cppflags = ", ".join('"%s"' % p for p in deps_cpp_info.cppflags)
        self.cflags = ", ".join('"%s"' % p for p in deps_cpp_info.cflags)
        self.sharedlinkflags = ", ".join('"%s"' % p for p in deps_cpp_info.
↪sharedlinkflags)
        self.exelinkflags = ", ".join('"%s"' % p for p in deps_cpp_info.exelinkflags)

        self.rootpath = "%s" % deps_cpp_info.rootpath.replace("\\", "/")


class premake(Generator):

    @property
    def filename(self):
        return "conanpremake.lua"

    @property
    def content(self):
        deps = PremakeDeps(self.deps_build_info)
```

(continues on next page)

(continued from previous page)

```
        template = ('conan_includedirs{dep} = {{{deps.include_paths}}}\n'
                    'conan_libdirs{dep} = {{{deps.lib_paths}}}\n'
                    'conan_bindirs{dep} = {{{deps.bin_paths}}}\n'
                    'conan_libs{dep} = {{{deps.libs}}}\n'
                    'conan_cppdefines{dep} = {{{deps.defines}}}\n'
                    'conan_cppflags{dep} = {{{deps.cppflags}}}\n'
                    'conan_cflags{dep} = {{{deps.cflags}}}\n'
                    'conan_sharedlinkflags{dep} = {{{deps.sharedlinkflags}}}\n'
                    'conan_exelinkflags{dep} = {{{deps.exelinkflags}}}\n')

        sections = ["#!lua"]
        all_flags = template.format(dep="", deps=deps)
        sections.append(all_flags)
        template_deps = template + 'conan_rootpath{dep} = "{deps.rootpath}"\n'

        for dep_name, dep_cpp_info in self.deps_build_info.dependencies:
            deps = PremakeDeps(dep_cpp_info)
            dep_name = dep_name.replace("-", "_")
            dep_flags = template_deps.format(dep="_" + dep_name, deps=deps)
            sections.append(dep_flags)

        return "\n".join(sections)


class MyPremakeGeneratorPackage(ConanFile):
    name = "premakegen"
    version = "0.1"
    url = "https://github.com/memsharded/conan-premake"
    license = "MIT"
```

This is a full working example. Note the `PremakeDeps` class as a helper. The generator is creating Premake information for each individual library separately, then also an aggregated information for all dependencies. This `PremakeDeps` wraps a single item of such information.

Note the **name of the package** will be **premakegen/0.1@<user>/<channel>** as that is the name given to it, while the generator name is **premake** (the name of the class that inherits from `Generator`). You can give the package any name you want, even the same as the generator's name if desired.

You `export` the package recipe to the local cache, so it can be used by other projects as usual:

```
$ conan export . myuser/testing
```

17.11.2 Using the generator

Let's create a test project that uses this generator. We will use a simple application that will use a "Hello World" library package as a requirement.

First, let's create the "Hello World" library package:

```
$ mkdir conan-hello && cd conan-hello
$ conan new hello/0.1
$ conan create . myuser/testing
```

Now, let's create a folder for the application that will use Premake as build system:

```
$ cd ..
$ mkdir premake-project && cd premake-project
```

Put the following files inside. Note the `premakegen@0.1@myuser/testing` package reference in your *conanfile.txt*.

Listing 12: *conanfile.txt*

```
[requires]
hello/0.1@myuser/testing
premakegen@0.1@myuser/testing

[generators]
premake
```

Listing 13: *main.cpp*

```
#include "hello.h"

int main (void) {
    hello();
}
```

Listing 14: *premake4.lua*

```
-- premake4.lua

require 'conanpremake'

-- A solution contains projects, and defines the available configurations solution
↪"MyApplication"

configurations { "Debug", "Release" }
includedirs { conan_includedirs }
libdirs { conan_libdirs }
links { conan_libs }

-- A project defines one build target

project "MyApplication"
    kind "ConsoleApp"
    language "C++"
    files { "**.h", "**.cpp" }

    configuration "Debug"
        defines { "DEBUG" }
        flags { "Symbols" }

    configuration "Release"
        defines { "NDEBUG" }
        flags { "Optimize" }
```

Let's install the requirements:

```
$ conan install . -s compiler=gcc -s compiler.version=4.9 -s compiler.libcxx=libstdc++ --
↪build
```

This generates the *premake4.lua* file with the requirements information for building.

Now we are ready to build the project:

```
$ premake4 gmake
$ make (or mingw32-make if in windows-mingw)
$ ./MyApplication
Hello World Release!
```

Now everything works, so you might want to share your generator:

```
$ conan upload premakegen/0.1@myuser/testing
```

Tip: This is a regular Conan package, so you could create a *test_package* folder with a *conanfile.py* to test the generator as done in the example above (invoke the Premake build in the `build()` method).

17.11.3 Using template files for custom generators

If your generator has a lot of common, non-parameterized text, you might want to use files that contain the template. It is possible to do this as long as the template file is exported in the recipe. The following example uses a simple text file, but you could use other templating formats:

```
import os
from conans import ConanFile, load
from conans.model import Generator


class MyCustomGenerator(Generator):

    @property
    def filename(self):
        return "customfile.gen"

    @property
    def content(self):
        template = load(os.path.join(os.path.dirname(__file__), "mytemplate.txt"))
        return template % "Hello"

class MyCustomGeneratorPackage(ConanFile):
    name = "custom_generator"
    version = "0.1"
    exports = "mytemplate.txt"
```

17.11.4 Storing generators in the Conan local cache

Warning: This is an **experimental** feature subject to breaking changes in future releases.

In addition to distributing them using Conan packages, custom generators can be stored in the generators folder in the Conan local cache (by default `~/.conan/generators`).

Generators stored in the local cache can be used in the same ways as the *built-in generators*, i.e. they can be referenced on the command line with **`conan install`** when using the **`--generator`** option, and do not require installing a package to use. Instead, these generators can be distributed using **`conan config install`**.

Listing 15: *A custom generator which saves all environment variables defined in a package to a json file*

```
import json
from conans.model import Generator


# The generator name will be the literal class name (not the filename)
class custom_generator(Generator):
    @property
    def filename(self):
        return "custom_generator_output.json"

    @property
    def content(self):
        return json.dumps(self.deps_env_info.vars)
```

Listing 16: *Using the custom generator at install time*

```
$ conan install <path_or_reference> --generator custom_generator
```

Note: Generators loaded from the local cache do not need to be accompanied by a recipe class. Additionally, more than one generator can be loaded from the same python module when loaded from the local cache.

17.12 How to manage shared libraries

Shared libraries, *.DLL* in windows, *.dylib* in OSX and *.so* in Linux, are loaded at runtime. That means that the application executable needs to know where are the required shared libraries when it runs.

On Windows, the dynamic linker, will search in the same directory then in the *PATH* directories. On OSX, it will search in the directories declared in *DYLD_LIBRARY_PATH* as on Linux will use the *LD_LIBRARY_PATH*.

Furthermore in OSX and Linux there is another mechanism to locate the shared libraries: The RPATHs.

17.12.1 Manage Shared Libraries with Environment Variables

Shared libraries are loaded at runtime. The application executable needs to know where to find the required shared libraries when it runs.

Depending on the operating system, we can use environment variables to help the dynamic linker to find the shared libraries:

OPERATING SYSTEM	ENVIRONMENT VARIABLE
WINDOWS	PATH
LINUX	LD_LIBRARY_PATH
OSX	DYLD_LIBRARY_PATH

If your package recipe (A) is generating shared libraries you can declare the needed environment variables pointing to the package directory. This way, any other package depending on (A) will automatically have the right environment variable set, so they will be able to locate the (A) shared library.

Similarly if you use the *virtualenv generator* and you activate it, you will get the paths needed to locate the shared libraries in your terminal.

Example

We are packaging a tool called `toolA` with a library and an executable that will, for example, compress data.

The package offers two flavors, shared library or static library (embedded in the executable of the tool and available to link with). You can use the `toolA` package library to develop another executable or library or you can just use the executable provided by the package. In both cases, if you choose to install the *shared* package of `toolA` you will need to have the shared library available.

```
import os
from conans import tools, ConanFile

class ToolA(ConanFile):
    ....
    name = "tool_a"
    version = "1.0"
    options = {"shared": [True, False]}
    default_options = {"shared": False}

    def build(self):
        # build your shared library

    def package(self):
        # Copy the executable
        self.copy(pattern="tool_a*", dst="bin", keep_path=False)

        # Copy the libraries
        if self.options.shared:
            self.copy(pattern="*.dll", dst="bin", keep_path=False)
            self.copy(pattern="*.dylib", dst="lib", keep_path=False)
            self.copy(pattern="*.so*", dst="lib", keep_path=False)
        else:
            ...
```

Using the tool from a different package

If we are now creating a package that uses the `tool_a` executable to compress some data, we can execute directly `tool_a` using RunEnvironment build helper to set the environment variables accordingly:

```
import os
from conans import tools, ConanFile

class PackageB(ConanFile):
    name = "package_b"
    version = "1.0"
    requires = "tool_a/1.0@myuser/stable"

    def build(self):
        exe_name = "tool_a.exe" if self.settings.os == "Windows" else "tool_a"
        self.run([exe_name, "--someparams"], run_environment=True)
        ...
```

Building an application using the shared library from `tool_a`

As we are building a final application, we will probably want to distribute it together with the shared library from the `tool_a`, so we can use the *Imports* to import the required shared libraries to our user space.

Listing 17: *conanfile.txt*

```
[requires]
tool_a/1.0@myuser/stable

[generators]
cmake

[options]
tool_a:shared=True

[imports]
bin, *.dll -> ./bin # Copies all dll files from packages bin folder to my "bin" folder
lib, *.dylib* -> ./bin # Copies all dylib files from packages lib folder to my "bin"␣
↪folder
lib, *.so* -> ./bin # Copies all so files from packages lib folder to my "bin" folder
```

Now you can build the project:

```
$ mkdir build && cd build
$ conan install ..
$ cmake .. -G "Visual Studio 14 Win64"
$ cmake --build . --config Release
$ cd bin && mytool
```

The previous example will work only in Windows and OSX (changing the CMake generator), because the dynamic linker will look in the current directory (the binary directory) where we copied the shared libraries too.

In Linux you still need to set the `LD_LIBRARY_PATH`, or in OSX, the `DYLD_LIBRARY_PATH`:

```
$ cd bin && LD_LIBRARY_PATH=$(pwd) && ./mytool
```

Using shared libraries from dependencies

If you are executing something that depends on shared libraries belonging to your dependencies, those shared libraries have to be found at runtime. In Windows, it is enough if the package added its binary folder to the system PATH. In Linux and OSX, it is necessary that the LD_LIBRARY_PATH and DYLD_LIBRARY_PATH environment variables are used.

Security restrictions might apply in OSX (read this thread), so the DYLD_LIBRARY_PATH and DYLD_FRAMEWORK_PATH environment variables are not directly transferred to the child process. In that case, you have to use it explicitly in your *conanfile.py*:

```
def build(self):
    env_build = RunEnvironment(self)
    with tools.environment_append(env_build.vars):
        # self.run("./myexetool") # won't work, even if 'DYLD_LIBRARY_PATH' and 'DYLD_
↪FRAMEWORK_PATH' are in the env
        self.run("DYLD_LIBRARY_PATH=%s DYLD_FRAMEWORK_PATH=%s ./myexetool" % (os.environ[
↪'DYLD_LIBRARY_PATH'], os.environ['DYLD_FRAMEWORK_PATH']))
```

Or you could use RunEnvironment helper described above.

Using virtualrunenv generator

virtualrunenv generator will set the environment variables PATH, LD_LIBRARY_PATH, DYLD_LIBRARY_PATH pointing to *lib* and *bin* folders automatically.

Listing 18: *conanfile.txt*

```
[requires]
tool_a/1.0@myuser/stable

[options]
tool_a:shared=True

[generators]
virtualrunenv
```

In the terminal window:

```
$ conan install .
$ source activate_run
$ tool_a --someparams
# Only For Mac OS users to avoid restrictions:
$ DYLD_LIBRARY_PATH=$DYLD_LIBRARY_PATH toolA --someparams
```

17.12.2 Manage RPATHs

The **rpath** is encoded inside dynamic libraries and executables and helps the linker to find its required shared libraries.

If we have an executable, **my_exe**, that requires a shared library, **shared_lib_1**, and **shared_lib_1**, in turn, requires another **shared_lib_2**.

So the **rpaths** values are:

File	rpath
my_exe	/path/to/shared_lib_1
shared_lib_1	/path/to/shared_lib_2
shared_lib_2	

In **Linux** if the linker doesn't find the library in **rpath**, it will continue the search in **system defaults paths** (LD_LIBRARY_PATH... etc) In OSX, if the linker detects an invalid **rpath** (the file does not exist there), it will fail.

Default Conan approach

The consumer project of dependencies with shared libraries needs to import them to the executable directory to be able to run it:

conanfile.txt

```
[requires]
poco/1.9.4

[imports]
bin, *.dll -> ./bin # Copies all dll files from packages bin folder to my "bin" folder
lib, *.dylib* -> ./bin # Copies all dylib files from packages lib folder to my "bin"␣
↪folder
```

On **Windows** this approach works well, importing the shared library to the directory containing your executable is a very common procedure.

On **Linux** there is an additional problem, the dynamic linker doesn't look by default in the executable directory, and you will need to adjust the *LD_LIBRARY_PATH* environment variable like this:

```
LD_LIBRARY_PATH=$(pwd) && ./mybin
```

On **OSX** if absolute rpaths are hardcoded in an executable or shared library and they don't exist the executable will fail to run. This is the most common problem when we reuse packages in a different environment from where the artifacts have been generated.

So for **OSX**, Conan, by default, when you build your library with **CMake**, the rpaths will be generated without any path:

File	rpath
my_exe	shared_lib_1.dylib
shared_lib_1.dylib	shared_lib_2.dylib
shared_lib_2.dylib	

The `conan_basic_setup()` macro will set the `set(CMAKE_SKIP_RPATH 1)` in OSX.

You can skip this default behavior by passing the KEEP_RPATHS parameter to the conan_basic_setup macro:

```
include(${CMAKE_BINARY_DIR}/conanbuildinfo.cmake)
conan_basic_setup(KEEP_RPATHS)

add_executable(timer timer.cpp)
target_link_libraries(timer ${CONAN_LIBS})
```

If you are using autotools Conan won't auto-adjust the rpaths behavior. if you want to follow this default behavior you will probably need to replace the install_name in the **configure** or **MakeFile** generated files in your recipe to not use $rpath:

```
replace_in_file("./configure", r"-install_name \$rpath/", "-install_name ")
```

Different approaches

You can adjust the **rpaths** in the way that adapts better to your needs.

If you are using CMake take a look to the CMake RPATH handling guide.

Remember to pass the KEEP_RPATHS variable to the conan_basic_setup:

```
include(${CMAKE_BINARY_DIR}/conanbuildinfo.cmake)
conan_basic_setup(KEEP_RPATHS)
```

Then, you could, for example, use the @executable_path in OSX and $ORIGIN in Linux to adjust a relative path from the executable. Also, enabling CMAKE_BUILD_WITH_INSTALL_RPATH will build the application with the RPATH value of CMAKE_INSTALL_RPATH and avoid the need to be relinked when installed.

```
if (APPLE)
    set(CMAKE_INSTALL_RPATH "@executable_path/../lib")
else()
    set(CMAKE_INSTALL_RPATH "$ORIGIN/../lib")
endif()

set(CMAKE_BUILD_WITH_INSTALL_RPATH ON)
```

You can use this imports statements in the consumer project:

```
[requires]
poco/1.9.4

[imports]
bin, *.dll -> ./bin # Copies all dll files from packages bin folder to my "bin" folder
lib, *.dylib* -> ./lib # Copies all dylib files from packages lib folder to my "lib"␣
↪folder
lib, *.so* -> ./lib # Copies all so files from packages lib folder to my "lib" folder
```

And your final application can follow this layout:

```
bin
 |_____ my_executable
 |_____ mylib.dll
 |
```

(continues on next page)

(continued from previous page)

```
lib
 |_____ libmylib.so
 |_____ libmylib.dylib
```

You could move the entire application folder to any location and the shared libraries will be located correctly.

17.13 How to reuse cmake install for package() method

It is possible that your project's *CMakeLists.txt* has already defined some functionality that extracts the artifacts (headers, libraries, binaries) from the build and source folder to a predetermined place and does the post-processing (*e.g.*, strips rpaths). For example, one common practice is to use CMake install directive to that end.

When using Conan, the install phase of CMake is wrapped in the `package()` method. That way the flags like **`conan create --keep-build`** or the commands for the *Package development flow* are consistent with every step of the packaging process.

The following excerpt shows how to build and package with CMake within Conan. Mind that you need to configure CMake both in `build()` and in `package()`, since these methods are called independently.

```
def _configure_cmake(self):
    cmake = CMake(self)
    cmake.definitions["SOME_DEFINITION"] = "VALUE"
    cmake.configure()
    return cmake

def build(self):
    cmake = self._configure_cmake()
    cmake.build()

def package(self):
    cmake = self._configure_cmake()
    cmake.install()

def package_info(self):
    self.cpp_info.libs = ["libname"]
```

The `package_info()` method specifies the list of the necessary libraries, defines and flags for different build configurations for the consumers of the package. This is necessary as there is no possible way to extract this information from the CMake install automatically.

17.14 How to collaborate with other users' packages

If a certain existing package does not work for you, or you need to store pre-compiled binaries for a platform not provided by the original package creator, you might still be able to do so:

17.14.1 Collaborate from source repository

If the original package creator has the package recipe in a repository, this would be the simplest approach. Just clone the package recipe on your machine, change something if you want, and then export the package recipe under your own user name. Point your project's [requires] to the new package name, and use it as usual:

```
$ git clone <repository>
$ cd <repository>
//make changes if desired
$ conan export . <youruser/yourchannel>
```

You can just directly run:

```
$ conan create . demo/testing
```

Once you have generated the desired binaries, you can store your pre-compiled binaries in your own free Artifactory CE repository:

```
$ conan upload package/0.1@myuser/stable -r=myremote --all
```

Finally, if you made useful changes, you might want to create a pull request to the original repository of the package creator.

17.14.2 Copy a package

If you don't need to modify the original package creator recipe, it is fine to just copy the package to your local storage. You can copy the recipes and existing binary packages. This could be enough for caching existing binary packages from the original remote into your own remote, under your own username:

```
$ conan copy poco/1.9.4@ myuser/testing
$ conan upload poco/1.9.4@myuser/testing -r=myremote --all
```

17.15 How to link with Apple Frameworks

It is common in MacOS that your Conan package needs to link with a complete Apple framework, and, of course, you want to propagate this information to all projects/libraries that use your package.

With regular libraries, use `self.cpp_info.libs` object to append to it all the libraries:

```
def package_info(self):

    self.cpp_info.libs = ["SDL2"]
    self.cpp_info.libs.append("OpenGL32")
```

With frameworks we need to use `self.cpp_info.frameworks` in a similar manner:

```
def package_info(self):

    self.cpp_info.libs = ["SDL2"]

    self.cpp_info.frameworks.extend(["Carbon", "CoreAudio", "Security", "IOKit"])
```

17.16 How to package Apple Frameworks

To package a **MyFramework** Apple framework, copy/create a folder `MyFramework.framework` to your package folder, where you should put all the subdirectories (`Headers`, `Modules`, etc).

```
def package(self):
    # If you have the framework folder built in your build_folder:
    self.copy("MyFramework.framework/*", symlinks=True)
    # Or build the destination folder:
    tools.mkdir("MyFramework.framework/Headers")
    self.copy("*.h", dst="MyFramework.framework/Headers")
    # ...
```

Declare the framework in the `cpp_info` object, the directory of the framework folder (self.package_folder) into the `cpp_info.frameworkdirs` and the framework name into the `cpp_info.frameworks`.

```
def package_info(self):
    ...
    self.cpp_info.frameworkdirs.append(self.package_folder)
    self.cpp_info.frameworks.append("MyFramework")
```

17.17 How to collect licenses of dependencies

With the `imports` feature it is possible to collect the License files from all packages in the dependency graph. Please note that the licenses are artifacts that must exist in the binary packages to be collected, as different binary packages might have different licenses. E.g., A package creator might provide a different license for static or shared linkage with different "License" files if they want to.

Also, we will assume the convention that the package authors will provide a "License" (case not important) file at the root of their packages.

In *conanfile.txt* we would use the following syntax:

```
[imports]
., license* -> ./licenses @ folder=True, ignore_case=True
```

And in *conanfile.py* we will use the `imports()` method:

```
def imports(self):
     self.copy("license*", dst="licenses", folder=True, ignore_case=True)
```

In both cases, after **`conan install`**, it will store all the found License files inside the local **licenses** folder, which will contain one subfolder per dependency with the license file inside.

17.18 How to extract licenses from headers

Sometimes there is no `license` file, and you will need to extract the license from a header file, as in the following example:

```
def package():
    # Extract the License/s from the header to a file
    tmp = tools.load("header.h")
    license_contents = tmp[2:tmp.find("*/", 1)] # The license begins with a C␣
↪comment /* and ends with */
    tools.save("LICENSE", license_contents)

    # Package it
    self.copy("license*", dst="licenses",  ignore_case=True, keep_path=False)
```

17.19 How to dynamically define the name and version of a package

The `name` and `version` fields are used to define constant values. The `set_name()` and `set_version()` methods can be used to dynamically define those values, for example if we want to extract the version from a text file or from the git repository.

The version of a recipe is stored in the package metadata when it is exported (or created) and always taken from the metadata later on. This means that the `set_name()` and `set_version()` methods will not be executed once the recipe is in the cache, or when it is installed from a server. Both methods will use the current folder as the current working directory to resolve relative paths. To define paths relative to the location of the *conanfile.py* use the `self.recipe_folder` attribute.

17.20 How to capture package version from SCM: git

The `Git()` helper from tools can be used to capture data from the Git repo in which the *conanfile.py* recipe resides, and use it to define the version of the Conan package.

```
from conans import ConanFile, tools

class HelloConan(ConanFile):
    name = "hello"

    def set_version(self):
        git = tools.Git(folder=self.recipe_folder)
        self.version = "%s_%s" % (git.get_branch(), git.get_revision())

    def build(self):
        ...
```

In this example, the package created with **`conan create`** will be called `hello/branch_commit@user/channel`.

17.21 How to capture package version from SCM: svn

The `SVN()` helper from tools can be used to capture data from the subversion repo in which the *conanfile.py* recipe resides, and use it to define the version of the Conan package.

```
from conans import ConanFile, tools

class HelloLibrary(ConanFile):
    name = "hello"
    def set_version(self):
        scm = tools.SVN(folder=self.recipe_folder)
        revision = scm.get_revision()
        branch = scm.get_branch() # Delivers e.g trunk, tags/v1.0.0, branches/my_branch
        branch = branch.replace("/","_")
        if scm.is_pristine():
            dirty = ""
        else:
            dirty = ".dirty"
        self.version = "%s-%s+%s%s" % (version, revision, branch, dirty) # e.g. 1.2.0-
↪1234+trunk.dirty

    def build(self):
        ...
```

In this example, the package created with **conan create** will be called `hello/generated_version@user/channel`. Note: this function should never raise, see the section about when the version is computed and saved above.

17.22 How to capture package version from text or build files

It is common that a library version number would be already encoded in a text file, build scripts, etc. As an example, let's assume we have the following library layout, and that we want to create a package from it:

```
conanfile.py
CMakeLists.txt
src
   hello.cpp
   ...
```

The *CMakeLists.txt* will have some variables to define the library version number. For simplicity, let's also assume that it includes a line such as the following:

```
cmake_minimum_required(VERSION 2.8)
set(MY_LIBRARY_VERSION 1.2.3) # This is the version we want
add_library(hello src/hello.cpp)
```

You can extract the version dynamically using:

```
from conans import ConanFile
from conans.tools import load
import re, os

class HelloConan(ConanFile):
```

(continues on next page)

(continued from previous page)

```
    name = "hello"
    def set_version(self):
        content = load(os.path.join(self.recipe_folder, "CMakeLists.txt"))
        version = re.search(r"set\(MY_LIBRARY_VERSION (.*)\)", content).group(1)
        self.version = version.strip()
```

17.23 How to use Conan as other language package manager

Conan is a generic package manager. In the *getting started* section we saw how to use Conan and manage a C/C++ library, like POCO.

But Conan just provided some tools, related to C/C++ (like some generators and the cpp_info), to offer a better user experience. The general basis of Conan can be used with other programming languages.

Obviously, this does not try to compete with other package managers. Conan is a C and C++ package manager, focused on C and C++ developers. But when we realized that this was possible, we thought it was a good way to showcase its power, simplicity and versatility.

And of course, if you are doing C/C++ and occasionally you need some package from other language in your workflow, as in the Conan package recipes themselves, or for some other tooling, you might find this functionality useful.

17.23.1 Conan: A Go package manager

The source code

You can just clone the following example repository:

```
$ git clone https://github.com/conan-io/examples/tree/master/features/goserver
```

Or, alternatively, manually create the folder and copy the following files inside:

```
$ mkdir conan-goserver-example
$ cd conan-goserver-example
$ mkdir src
$ mkdir src/server
```

The files are:

src/server/main.go is a small http server that will answer "Hello world!" if we connect to it.

```
package main

import "github.com/go-martini/martini"

func main() {
  m := martini.Classic()
  m.Get("/", func() string {
    return "Hello world!"
  })
  m.Run()
}
```

Declaring and installing dependencies

Create a *conanfile.txt*, with the following content:

Listing 19: *conanfile.txt*

```
[requires]
go-martini/1.0@

[imports]
src, * -> ./deps/src
```

Our project requires a package, **go-martini/1.0@**, and we indicate that all **src contents** from all our requirements have to be copied to *./deps/src*.

The package go-martini depends on go-inject, so Conan will handle automatically the go-inject dependency.

```
$ conan install .
```

This command will download our packages and will copy the contents in the *./deps/src* folder.

Running our server

Just add the **deps** folder to `GOPATH`:

```
# Linux / Macos
$ export GOPATH=${GOPATH}:${PWD}/deps

# Windows
$ SET GOPATH=%GOPATH%;%CD%/deps
```

And run the server:

```
$ cd src/server
$ go run main.go
```

Open your browser and go to *localhost:9300*

```
Hello World!
```

Generating Go packages

Creating a Conan package for a Go library is very simple. In a Go project, you compile all the code from sources in the project itself, including all of its dependencies.

So we don't need to take care of settings at all. Architecture, compiler, operating system, etc. are only relevant for pre-compiled binaries. Source code packages are settings agnostic.

Let's take a look at the *conanfile.py* of the **go inject** library:

Listing 20: *conanfile.py*

```
import os
from conans import ConanFile, tools
```

(continues on next page)

(continued from previous page)

```
class GoInjectConan(ConanFile):
    name = "go-inject"
    version = "1.0"
    license = "MIT"
    homepage = "https://github.com/codegangsta/inject"
    no_copy_source = True

    def source(self):
        tools.get("https://github.com/codegangsta/inject/archive/v1.0-rc1.tar.gz",
                  sha256="22b265ea391a19de6961aaa8811ecfcc5bbe7979594e30663c610821cdad6c7b
↪")

    def package(self):
        self.copy(pattern='*',
                  dst=os.path.join("src", "github.com", "codegangsta", "inject"),
                  src="inject-1.0-rc1", keep_path=True)
```

If you have read the *Building a hello world package*, the previous code may look quite simple to you.

We want to pack **version 1.0** of the **go inject** library, so the **version** variable is **"1.0"**. In the `source()` method, we declare how to obtain the source code of the library, in this case just by downloading **v1.0-rc1** tag. In the `package()` method, we are just copying all the sources to a folder named "src/github.com/codegangsta/inject".

This way, we can keep importing the library in the same way:

```
import "github.com/codegangsta/inject"
```

We can export and upload the package to a remote and we are done:

```
$ conan create . # Or any other user/channel
$ conan upload go-inject/1.0@ --all
```

Now look at the **go martini** conanfile:

Listing 21: *conanfile.py*

```
import os
from conans import ConanFile, tools


class GoMartiniConan(ConanFile):
    name = "go-martini"
    version = "1.0"
    requires = "go-inject/1.0@"
    license = "MIT"
    homepage = "https://github.com/go-martini/martini"
    no_copy_source = True

    def source(self):
        tools.get("https://github.com/go-martini/martini/archive/v1.0.tar.gz",
                  sha256="3db135845d076d611f4420e0500e91625543a6b00dc9431cbe45d3571741281b
↪")
```

(continues on next page)

(continued from previous page)

```
    def package(self):
        self.copy(pattern="*", dst=os.path.join("src", "github.com", "go-martini",
↪"martini"),
                  src="martini-1.0", keep_path=True)
```

It is very similar. The only difference is the `requires` variable. It defines the **go-inject/1.0@** library, as a requirement.

```
$ conan create . # Or any other user/channel
$ conan upload go-martini/1.0@ --all
```

Now we are able to use them easily and without the problems of versioning with github checkouts.

17.23.2 Conan: A Python package manager

Conan is a C and C++ package manager, and to deal with the vast variability of C and C++ build systems, compilers, configurations, etc., it was designed to be extremely flexible, to allow users the freedom to configure builds in virtually any manner required. This is one of the reasons to use Python as the scripting language for Conan package recipes.

With this flexibility, Conan is able to do very different tasks: package Visual Studio modules, *package Go code*, build packages from sources or from binaries retrieved from elsewhere, etc.

Python code can be reused and packaged with Conan to share functionalities or tools among *conanfile.py* files. Here we can see a full example of Conan as a Python package manager.

A full Python and C/C++ package manager

The real utility of this is that Conan is a C and C++ package manager. So, for example, you are able to create a Python package that wraps the functionality of the Poco C++ library. Poco itself has transitive (C/C++) dependencies, but they are already handled by Conan. Furthermore, a very interesting thing is that nothing has to be done in advance for that library, thanks to useful tools such as **pybind11**, that lets you easily create Python bindings.

So let's build a package with the following files:

- *conanfile.py*: The package recipe.
- *__init__.py*: A required file which should remain blank.
- *pypoco.cpp*: The C++ code with the `pybind11` wrapper for Poco that generates a Python extension (a shared library that can be imported from Python).
- *CMakeLists.txt*: The CMake build file that is able to compile *pypoco.cpp* into a Python extension (*pypoco.pyd* in Windows, *pypoco.so* in Linux)
- *poco.py*: A Python file that makes use of the pypoco Python binary extension built with *pypoco.cpp*.
- *test_package/conanfile.py*: A test consumer "convenience" recipe to create and test the package.

The *pypoco.cpp* file can be coded easily thanks to the elegant `pybind11` library:

Listing 22: pypoco.cpp

```
#include <pybind11/pybind11.h>
#include "Poco/Random.h"

using Poco::Random;
```

(continues on next page)

(continued from previous page)

```
namespace py = pybind11;

PYBIND11_PLUGIN(pypoco) {
    py::module m("pypoco", "pybind11 example plugin");
    py::class_<Random>(m, "Random")
        .def(py::init<>())
        .def("nextFloat", &Random::nextFloat);
    return m.ptr();
}
```

And the *poco.py* file is straightforward:

Listing 23: *poco.py*

```
import sys
import pypoco

def random_float():
    r = pypoco.Random()
    return r.nextFloat()
```

The *conanfile.py* is a bit longer, but is still quite easy to understand:

Listing 24: *conanfile.py*

```
from conans import ConanFile, tools, CMake

class PocoPyReuseConan(ConanFile):
    name = "PocoPy"
    version = "0.1"
    requires = "poco/1.9.4", "pybind11/2.3.0@conan/stable"
    settings = "os", "compiler", "arch", "build_type"
    exports = "*"
    generators = "cmake"
    build_policy = "missing"

    def build(self):
        cmake = CMake(self)
        pythonpaths = "-DPYTHON_INCLUDE_DIR=C:/Python27/include -DPYTHON_LIBRARY=C:/
↪Python27/libs/python27.lib"
        self.run('cmake %s %s -DEXAMPLE_PYTHON_VERSION=2.7' % (cmake.command_line,␣
↪pythonpaths))
        self.run("cmake --build . %s" % cmake.build_config)

    def package(self):
        self.copy('*.py*')
        self.copy("*.so")

    def package_info(self):
        self.env_info.PYTHONPATH.append(self.package_folder)
```

The recipe now declares 2 `requires` that will be used to create the binary extension: the **Poco library** and the **pybind11 library**.

As we are actually building C++ code, there are a few important things that we need:

- Input `settings` that define the OS, compiler, version and architecture we are using to build our extension. This is necessary because the binary we are building must match the architecture of the Python interpreter that we will be using.
- The `build()` method is actually used to invoke CMake. You may see that we had to hardcode the Python path in the example, as the *CMakeLists.txt* call to `find_package(PythonLibs)` didn't find my Python installation in *C:/Python27*, even though that is a standard path. I have also added the `cmake` generator to be able to easily use the declared `requires` build information inside my *CMakeLists.txt*.
- The *CMakeLists.txt* is not posted here, but is basically the one used in the pybind11 example with just 2 lines to include the cmake file generated by Conan for dependencies. It can be inspected in the GitHub repo.
- Note that we are using Python 2.7 as an input option. If necessary, more options for other interpreters/architectures could be easily provided, as well as avoiding the hardcoded paths. Even the Python interpreter itself could be packaged in a Conan package.

The above recipe will generate a different binary for different compilers or versions. As the binary is being wrapped by Python, we could avoid this and use the same binary for different setups, modifying this behavior with the `conan_info()` method.

```
$ conan export . memsharded/testing
$ conan install pocopy/0.1@memsharded/testing -s arch=x86 -g virtualenv
$ activate
$ python
>>> import poco
>>> poco.random_float()
0.697845458984375
```

Now, the first invocation of **`conan install`** will retrieve the dependencies and build the package. The next invocation will use the cached binaries and be much faster. Note how we have to specify `-s arch=x86` to match the architecture of the Python interpreter to be used, in our case, 32 bits.

The output of the **`conan install`** command also shows us the dependencies that are being pulled:

```
Requirements
    openssl/1.0.2t from conan.io
    poco/1.9.4 from conan.io
    pocopy/0.1@memsharded/testing from local
    pybind11/2.3.0@conan/stable from conan.io
    zlib/1.2.11 from conan.io
```

This is one of the great advantages of using Conan for this task, because by depending on Poco, other C and C++ transitive dependencies are retrieved and used in the application.

For a deeper look into the code of these examples, please refer to this github repo. The above examples and code have only been tested on Win10, VS14u2, but may work on other configurations with little or no extra work.

17.24 How to manage SSL (TLS) certificates

17.24.1 Server certificate validation

By default, when a remote is added, if the URL schema is `https`, the Conan client will verify the certificate using a list of authorities declared in the `cacert.pem` file located in the Conan home (~/.conan).

If you have a self signed certificate (not signed by any authority) you have two options:

- Use the *conan remote* command to disable the SSL verification.
- Append your server `crt` file to the `cacert.pem` file.

17.24.2 Client certificates

If your server is requiring client certificates to validate a connection from a Conan client, you need to create two files in the Conan home directory (default ~/.conan):

- A file `client.crt` with the client certificate.
- A file `client.key` with the private key.

Note: You can create only the `client.crt` file containing both the certificate and the private key concatenated and not create the `client.key`

If you are a familiar with the curl tool, this mechanism is similar to specify the `--cert` / `--key` parameters.

17.25 How to check the version of the Conan client inside a conanfile

Sometimes it might be useful to check the Conan version that is running in that moment your recipe. Although we consider ConanCenter recipes only forward compatible, this kind of check makes sense to update them so they can maintain compatibility with old versions of Conan.

Let's have a look at a basic example of this:

Listing 25: conanfile.py

```
from conans import ConanFile, CMake, __version__ as conan_version
from conans.tools import Version


class MyLibraryConan(ConanFile):
    name = "mylibrary"
    version = "1.0"

    def build(self):
        if conan_version < Version("0.29"):
            cmake = CMake(self.settings)
        else:
            cmake = CMake(self)
    ...
```

Here it checks the Conan version to maintain compatibility of the CMake build helper for versions lower than Conan 0.29. It also uses the internal `Version()` class to perform the semver comparison in the if-clause.

You can also use it to take advantage of new features when the client is new enough, for example:

```
from conans import ConanFile, tools, __version__ as conan_version
from conans.tools import Version

class MyPackage(ConanFile):
    name = "package"
    ...

    def package_id(self):
        if conan_version >= Version("1.20"):
            if self.settings.compiler == "gcc" and self.settings.compiler.version == "4.9
↪":
                compatible_pkg = self.info.clone()
                compatible_pkg.settings.compiler.version = "4.8"
                self.compatible_packages.append(compatible_pkg)
```

It can be useful to introduce new features in your recipes while all the consumers update their client version. Together with our *stability commitment for Conan 1.x* it should be easy to adopt new Conan versions while evolving your recipes.

17.26 Use a generic CI with Conan and Artifactory

Warning: Some problems regarding the use of BuildInfo with Conan packages have been reported. If the BuildInfo contains artifacts that have the same checksum as other artifacts, this may result in losing the path of the artifact in the BuildInfo in Artifactory and also fail in the promotion process.

We are currently working along with the Artifactory team to solve those problems. Until this issue gets fixed, we do not recommend using BuildInfo's for Conan.

17.26.1 Uploading the BuildInfo

If you are using *Jenkins with Conan and Artifactory*, along with the Jenkins Artifactory Plugin, any Conan package downloaded or uploaded during your build will be automatically recorded in the BuildInfo json file, that will be automatically uploaded to the specified Artifactory instance.

However, using the *conan_build_info* command, you can gather and upload that information using other CI infrastructure. There are two possible ways of using this commmand:

Extracting build-info from the Conan trace log

1. Before calling Conan the first time in your build, set the environment variable *CONAN_TRACE_FILE* to a file path. The generated file will contain the BuildInfo json.
2. You also need to create the *artifacts.properties* file in your Conan home containing the build information. All this properties will be automatically associated to all the published artifacts.

```
artifact_property_build.name=MyBuild
artifact_property_build.number=23
artifact_property_build.timestamp=1487676992
```

3. Call Conan as many times as you need. For example, if you are testing a Conan package and uploading it at the end, you will run something similar to:

```
$ conan create . user/stable # Will retrieve the dependencies and create the package
$ conan upload mypackage/1.0@user/stable -r artifactory
```

4. Call the command *conan_build_info* passing the path to the generated Conan traces file and a parameter `--output` to indicate the output file. You can also, delete the traces.log` file` otherwise while the *CONAN_TRACE_FILE* is present, any Conan command will keep appending actions.

```
$ conan_build_info /tmp/traces.log --output /tmp/build_info.json
$ rm /tmp/traces.log
```

5. Edit the *build_info.json* file to append `name` (build name), `number` (build number) and the `started` (started date) and any other field that you need according to the Build Info json format.

   The `started` field has to be in the format: `yyyy-MM-dd'T'HH:mm:ss.SSSZ`

   To edit the file you can import the json file using the programming language you are using in your framework, groovy, java, python...

6. Push the json file to Artifactory, using the REST-API:

```
curl -X PUT -u<username>:<password> -H "Content-type: application/json" -T /tmp/build_
↪info.json "http://host:8081/artifactory/api/build"
```

Generating build info from lockfiles information

> **Warning:** This is an **experimental** feature subject to breaking changes in future releases.

To maintain compatibility with the current implementation of the `conan_build_info` command, this version must be invoked using the argument `--v2` before any subcommand.

1. To begin associating the build information to the uploaded packages the first thing is calling to the `start` subcommand of `conan_build_info`. This will set the *artifact_property_build.name* and *artifact_property_build.name* properties in the *artifacts.properties*.

```
$ conan_build_info --v2 start MyBuildName 42
```

2. Call Conan using *lockfiles* to create information for the Build Info json format.

```
$ cd mypackage
$ conan create . mypackage/1.0@user/stable # We create one package
$ cd .. && cd consumer
$ conan install . # Consumes mypackage, generates a lockfile
$ conan create . consumer/1.0@user/stable --lockfile conan.lock
$ conan upload "*" -c -r local # Upload all packages to local remotes
```

3. Create build information based on the contents of the generated *conan.lock* lockfile and the information retrieved from the remote (the authentication is for the remote where you uploaded the packages).

```
$ conan_build_info --v2 create buildinfo.json --lockfile conan.lock --user admin --
↪password password
```

4. Publish the build information to Artifactory with the `publish` subcommand:

Using user and password

```
$ conan_build_info --v2 publish buildinfo.json --url http://localhost:8081/artifactory --
↪user admin --password password
```

or an API key:

```
$ conan_build_info --v2 publish buildinfo.json --url http://localhost:8081/artifactory --
↪apikey apikey
```

5. If the whole process has finished and you don't want to continue associating the build number and build name to the files uploaded to Artifactory then you can use the `stop` subcommand:

```
$ conan_build_info --v2 stop
```

It is also possible to merge different build info files using the `update` subcommand. This is useful in CI when many slaves are generating different build info files.

```
$ conan_build_info --v2 update buildinfo1.json buildinfo2.json --output-file␣
↪mergedbuildinfo.json
```

You can check the complete *conan_build_info reference*.

17.27 Compiler sanitizers

Sanitizers are tools that can detect bugs such as buffer overflows or accesses, dangling pointer or different types of undefined behavior.

The two compilers that mainly support sanitizing options are gcc and clang. These options are passed to the compiler as flags and, depending on if you are using clang or gcc, different sanitizers are supported.

Here we explain different options on how to model and use sanitizers with your Conan packages.

17.27.1 Adding custom settings

If you want to model the sanitizer options so that the package id is affected by them, you have to introduce new settings in the *settings.yml* file (see *Customizing settings* section for more information).

Sanitizer options should be modeled as sub-settings of the compiler. Depending on how you want to combine the sanitizers you have two choices.

Adding a list of commonly used values

If you have a fixed set of sanitizers or combinations of them that are the ones you usually set for your builds you can add the sanitizers as a list of values. An example for *apple-clang* would be like this:

Listing 26: *settings.yml*

```
apple-clang:
    version: ["5.0", "5.1", "6.0", "6.1", "7.0", "7.3", "8.0", "8.1",
              "9.0", "9.1", "10.0", "11.0"]
```

(continues on next page)

(continued from previous page)

```
    libcxx: [libstdc++, libc++]
    cppstd: [None, 98, gnu98, 11, gnu11, 14, gnu14, 17, gnu17, 20, gnu20]
    sanitizer: [None, Address, Thread, Memory, UndefinedBehavior,␣
↪AddressUndefinedBehavior]
```

Here you have modeled the use of `-fsanitize=address`, `-fsanitize=thread`, `-fsanitize=memory`, `-fsanitize=undefined` and the combination of `-fsanitize=address` and `-fsanitize=undefined`. Note that for example, for clang it is not possible to combine more than one of the `-fsanitize=address`, `-fsanitize=thread`, and `-fsanitize=memory` checkers in the same program.

Adding thread sanitizer for a **conan install**, in this case, could be done by calling **conan install .. -s compiler.sanitizer=Thread**

Adding different values to combine

Another option would be to add the sanitizer values as multiple `True` or `None` fields so that they can be freely combined later. An example of that for the previous sanitizer options would be as follows:

Listing 27: *settings.yml*

```
apple-clang:
    version: ["5.0", "5.1", "6.0", "6.1", "7.0", "7.3", "8.0",
              "8.1", "9.0", "9.1", "10.0", "11.0"]
    libcxx: [libstdc++, libc++]
    cppstd: [None, 98, gnu98, 11, gnu11, 14, gnu14, 17, gnu17, 20, gnu20]
    address_sanitizer: [None, True]
    thread_sanitizer: [None, True]
    undefined_sanitizer: [None, True]
```

Then, you can add different sanitizers calling, for example, to **conan install .. -s compiler.address_sanitizer=True -s compiler.undefined_sanitizer=True**

A drawback of this approach is that not all the combinations will be valid or will make sense, but it is up to the consumer to use it correctly.

17.27.2 Passing the information to the compiler or build system

Here again, we have multiple choices to pass sanitizers information to the compiler or build system.

Using from custom profiles

It is possible to have different custom profiles defining the compiler sanitizer setting and environment variables to inject that information to the compiler, and then passing those profiles to Conan commands. An example of this would be a profile like:

Listing 28: *address_sanitizer_profile*

```
[settings]
os=Macos
os_build=Macos
arch=x86_64
```

(continues on next page)

(continued from previous page)

```
arch_build=x86_64
compiler=apple-clang
compiler.version=10.0
compiler.libcxx=libc++
build_type=Release
compiler.sanitizer=Address
[env]
CFLAGS=-fsanitize=address
CXXFLAGS=-fsanitize=address
LDFLAGS=-fsanitize=address
```

Then calling **conan create . -pr address_sanitizer_profile** would inject `-fsanitize=address` to the build through the `CFLAGS`, `CXXFLAGS`, and `LDFLAGS` environment variables.

Managing sanitizer settings with the build system

Another option is to make use of the information that is propagated to the *conan generator*. For example, if we are using CMake we could use the information from the *CMakeLists.txt* to append the flags to the compiler settings like this:

Listing 29: *CMakeLists.txt*

```
cmake_minimum_required(VERSION 3.2)
project(SanitizerExample)
set (CMAKE_CXX_STANDARD 11)
include(${CMAKE_BINARY_DIR}/conanbuildinfo.cmake)
conan_basic_setup()
set(SANITIZER ${CONAN_SETTINGS_COMPILER_SANITIZER})
if(SANITIZER)
    if(SANITIZER MATCHES "(Address)")
    set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -fsanitize=address" )
    endif()
endif()
add_executable(sanit_example src/main.cpp)
```

The sanitizer setting is propagated to CMake as the `CONAN_SETTINGS_COMPILER_SANITIZER` variable with a value equals to `"Address"` and we can set the behavior in CMake depending on the value of the variable.

Using conan Hooks to set compiler environment variables

Warning: This way of adding sanitizers is recommended just for testing purposes. In general, it's not a good practice to inject this in the environment using a Conan hook. It's much better explicitly defining this in the profiles.

Important: Take into account that the package ID doesn't encode information about the environment, so different binaries due to different *CXX_FLAGS* would be considered by Conan as the same package.

If you are not interested in modelling the settings in the Conan package you can use a *Hook* to modify the environment variable and apply the sanitizer flags to the build. It could be something like:

Listing 30: *sanitizer_hook.py*

```
import os


class SanitizerHook(object):
    def __init__(self):
        self._old_cxx_flags = None

    def set_sanitize_address_flag(self):
        self._old_cxx_flags = os.environ.get("CXXFLAGS")
        flags_str = self._old_cxx_flags or ""
        os.environ["CXXFLAGS"] = flags_str + " -fsanitize=address"

    def reset_sanitize_address_flag(self):
        if self._old_cxx_flags is None:
            del os.environ["CXXFLAGS"]
        else:
            os.environ["CXXFLAGS"] = self._old_cxx_flags


sanitizer = SanitizerHook()


def pre_build(output, conanfile, **kwargs):
    sanitizer.set_sanitize_address_flag()


def post_build(output, conanfile, **kwargs):
    sanitizer.reset_sanitize_address_flag()
```

CHAPTER

EIGHTEEN

REFERENCE

General information about the commands, configuration files, etc.

Contents:

18.1 Commands

18.1.1 Consumer commands

Commands related with the installation and usage of Conan packages:

conan install

```
$ conan install [-h] [-g GENERATOR] [-if INSTALL_FOLDER] [-of OUTPUT_FOLDER] [-m␣
↪[MANIFESTS]] [-mi [MANIFESTS_INTERACTIVE]]
                [-v [VERIFY]] [--no-imports] [--build-require] [-j JSON] [-b [BUILD]] [-
↪r REMOTE] [-u] [-l LOCKFILE] [--lockfile-out LOCKFILE_OUT]
                [-e ENV_HOST] [-e:b ENV_BUILD] [-e:h ENV_HOST] [-o OPTIONS_HOST] [-o:b␣
↪OPTIONS_BUILD] [-o:h OPTIONS_HOST] [-pr PROFILE_HOST]
                [-pr:b PROFILE_BUILD] [-pr:h PROFILE_HOST] [-s SETTINGS_HOST] [-s:b␣
↪SETTINGS_BUILD] [-s:h SETTINGS_HOST] [-c CONF_HOST]
                [-c:b CONF_BUILD] [-c:h CONF_HOST] [--lockfile-node-id LOCKFILE_NODE_ID]␣
↪[--require-override REQUIRE_OVERRIDE]
                path_or_reference [reference]
```

Installs the requirements specified in a recipe (conanfile.py or conanfile.txt).

It can also be used to install a concrete package specifying a reference. If any requirement is not found in the local cache, it will retrieve the recipe from a remote, looking for it sequentially in the configured remotes. When the recipes have been downloaded it will try to download a binary package matching the specified settings, only from the remote from which the recipe was retrieved. If no binary package is found, it can be built from sources using the '–build' option. When the package is installed, Conan will write the files for the specified generators.

```
positional arguments:
  path_or_reference     Path to a folder containing a recipe (conanfile.py or
                        conanfile.txt) or to a recipe file. e.g.,
                        ./my_project/conanfile.txt. It could also be a
                        reference
  reference             Reference for the conanfile path of the first
```

(continues on next page)

(continued from previous page)

```
                        argument: user/channel, version@user/channel or
                        pkg/version@user/channel(if name or version declared
                        in conanfile.py, they should match)

optional arguments:
  -h, --help            show this help message and exit
  -g GENERATOR, --generator GENERATOR
                        Generators to use
  -if INSTALL_FOLDER, --install-folder INSTALL_FOLDER
                        Use this directory as the directory where to put the
                        generatorfiles. e.g., conaninfo/conanbuildinfo.txt
  -of OUTPUT_FOLDER, --output-folder OUTPUT_FOLDER
                        The root output folder for generated and build files
  -m [MANIFESTS], --manifests [MANIFESTS]
                        Install dependencies manifests in folder for later
                        verify. Default folder is .conan_manifests, but can be
                        changed
  -mi [MANIFESTS_INTERACTIVE], --manifests-interactive [MANIFESTS_INTERACTIVE]
                        Install dependencies manifests in folder for later
                        verify, asking user for confirmation. Default folder
                        is .conan_manifests, but can be changed
  -v [VERIFY], --verify [VERIFY]
                        Verify dependencies manifests against stored ones
  --no-imports          Install specified packages but avoid running imports
  --build-require       The provided reference is a build-require
  -j JSON, --json JSON  Path to a json file where the install information will
                        be written
  -b [BUILD], --build [BUILD]
                        Optional, specify which packages to build from source.
                        Combining multiple '--build' options on one command
                        line is allowed. For dependencies, the optional
                        'build_policy' attribute in their conanfile.py takes
                        precedence over the command line parameter. Possible
                        parameters: --build Force build for all packages, do
                        not use binary packages. --build=never Disallow build
                        for all packages, use binary packages or fail if a
                        binary package is not found. Cannot be combined with
                        other '--build' options. --build=missing Build
                        packages from source whose binary package is not
                        found. --build=outdated Build packages from source
                        whose binary package was not generated from the latest
                        recipe or is not found. --build=cascade Build packages
                        from source that have at least one dependency being
                        built from source. --build=[pattern] Build packages
                        from source whose package reference matches the
                        pattern. The pattern uses 'fnmatch' style wildcards.
                        --build=![pattern] Excluded packages, which will not
                        be built from the source, whose package reference
                        matches the pattern. The pattern uses 'fnmatch' style
                        wildcards. Default behavior: If you omit the '--build'
                        option, the 'build_policy' attribute in conanfile.py
                        will be used if it exists, otherwise the behavior is
```

(continues on next page)

(continued from previous page)

```
                      like '--build=never'.
-r REMOTE, --remote REMOTE
                      Look in the specified remote server
-u, --update          Will check the remote and in case a newer version
                      and/or revision of the dependencies exists there, it
                      will install those in the local cache. When using
                      version ranges, it will install the latest version
                      that satisfies the range. Also, if using revisions, it
                      will update to the latest revision for the resolved
                      version range.
-l LOCKFILE, --lockfile LOCKFILE
                      Path to a lockfile
--lockfile-out LOCKFILE_OUT
                      Filename of the updated lockfile
-e ENV_HOST, --env ENV_HOST
                      Environment variables that will be set during the
                      package build (host machine). e.g.: -e
                      CXX=/usr/bin/clang++
-e:b ENV_BUILD, --env:build ENV_BUILD
                      Environment variables that will be set during the
                      package build (build machine). e.g.: -e:b
                      CXX=/usr/bin/clang++
-e:h ENV_HOST, --env:host ENV_HOST
                      Environment variables that will be set during the
                      package build (host machine). e.g.: -e:h
                      CXX=/usr/bin/clang++
-o OPTIONS_HOST, --options OPTIONS_HOST
                      Define options values (host machine), e.g.: -o
                      Pkg:with_qt=true
-o:b OPTIONS_BUILD, --options:build OPTIONS_BUILD
                      Define options values (build machine), e.g.: -o:b
                      Pkg:with_qt=true
-o:h OPTIONS_HOST, --options:host OPTIONS_HOST
                      Define options values (host machine), e.g.: -o:h
                      Pkg:with_qt=true
-pr PROFILE_HOST, --profile PROFILE_HOST
                      Apply the specified profile to the host machine
-pr:b PROFILE_BUILD, --profile:build PROFILE_BUILD
                      Apply the specified profile to the build machine
-pr:h PROFILE_HOST, --profile:host PROFILE_HOST
                      Apply the specified profile to the host machine
-s SETTINGS_HOST, --settings SETTINGS_HOST
                      Settings to build the package, overwriting the
                      defaults (host machine). e.g.: -s compiler=gcc
-s:b SETTINGS_BUILD, --settings:build SETTINGS_BUILD
                      Settings to build the package, overwriting the
                      defaults (build machine). e.g.: -s:b compiler=gcc
-s:h SETTINGS_HOST, --settings:host SETTINGS_HOST
                      Settings to build the package, overwriting the
                      defaults (host machine). e.g.: -s:h compiler=gcc
-c CONF_HOST, --conf CONF_HOST
                      Configuration to build the package, overwriting the defaults␣
```

(continues on next page)

(continued from previous page)

```
↪(host machine). e.g.: -c
                      tools.cmake.cmaketoolchain:generator=Xcode
  -c:b CONF_BUILD, --conf:build CONF_BUILD
                      Configuration to build the package, overwriting the defaults␣
↪(build machine). e.g.: -c:b
                      tools.cmake.cmaketoolchain:generator=Xcode
  -c:h CONF_HOST, --conf:host CONF_HOST
                      Configuration to build the package, overwriting the defaults␣
↪(host machine). e.g.: -c:h
                      tools.cmake.cmaketoolchain:generator=Xcode
  --lockfile-node-id LOCKFILE_NODE_ID
                      NodeID of the referenced package in the lockfile
  --require-override REQUIRE_OVERRIDE
                      Define a requirement override
```

conan install executes methods of a *conanfile.py* in the following order:

1. `config_options()`
2. `configure()`
3. `requirements()`
4. `package_id()`
5. `package_info()`
6. `deploy()`

Note this describes the process of installing a pre-built binary package. If the package has to be built, **conan install --build** executes the following:

1. `config_options()`
2. `configure()`
3. `requirements()`
4. `package_id()`
5. `build_requirements()`
6. `build_id()`
7. `system_requirements()`
8. `source()`
9. `imports()`
10. `build()`
11. `package()`
12. `package_info()`
13. `deploy()`

Examples

- Install a package requirement from a `conanfile.txt`, saved in your current directory with one option and setting (other settings will be defaulted as defined in `<userhome>/.conan/profiles/default`):

```
$ conan install . -o pkg_name:use_debug_mode=on -s compiler=clang
```

- Install the requirements defined in a `conanfile.py` file in your current directory, with the default settings in default profile `<userhome>/.conan/profiles/default`, and specifying the version, user and channel (as they might be used in the recipe):

```
class Pkg(ConanFile):
    name = "mypkg"
    # see, no version defined!
    def requirements(self):
        # this trick allow to depend on packages on your same user/channel
        self.requires("dep/0.3@%s/%s" % (self.user, self.channel))

    def build(self):
        if self.version == "myversion":
            # something specific for this version of the package.
```

```
$ conan install . myversion@someuser/somechannel
```

  Those values are cached in a file, so later calls to local commands like `conan build` can find and use this version, user and channel data.

- Install the **opencv/4.1.1@conan/stable** reference with its default options and default settings from `<userhome>/.conan/profiles/default`:

```
$ conan install opencv/4.1.1@conan/stable
```

- Install the **opencv/4.1.1@conan/stable** reference updating the recipe and the binary package if new upstream versions are available:

```
$ conan install opencv/4.1.1@conan/stable --update
```

build options

Both the conan **install** and **create** commands accept **--build** options to specify which packages to build from source. Combining multiple **--build** options on one command line is allowed, where a package is built from source if at least one of the given build options selects it for the build. For dependencies, the optional `build_policy` attribute in their *conanfile.py* can override the behavior of the given command line parameters. Possible values are:

- **--build**: Always build everything from source. Produces a clean re-build of all packages. and transitively dependent packages
- **--build=never**: Conan will not try to build packages when the requested configuration does not match, in which case it will throw an error. This option can not be combined with other **--build** options.
- **--build=missing**: Conan will try to build packages from source whose binary package was not found in the requested configuration on any of the active remotes or the cache.
- **--build=outdated**: Conan will try to build packages from source whose binary package was not built with the current recipe or when missing the binary package.
- **--build=cascade**: Conan selects packages for the build where at least one of its dependencies is selected for the build. This is useful to rebuild packages that, directly or indirectly, depend on changed packages.

- **--build=[pattern]**: A fnmatch case-sensitive pattern of a package reference or only the package name. Conan will force the build of the packages whose reference matches the given **pattern**. Several patterns can be specified, chaining multiple options:
  - e.g., **--build=pattern1 --build=pattern2** can be used to specify more than one pattern.
  - e.g., **--build=zlib** will match any package named `zlib` (same as `zlib/*`).
  - e.g., **--build=z*@conan/stable** will match any package starting with `z` with `conan/stable` as user/channel.
- **--build=![pattern]**: A fnmatch case-sensitive pattern of a package reference or only the package name. Conan will exclude the build of the packages whose reference matches the given **pattern**. Several patterns can be specified, chaining multiple options:
  - e.g., **--build=!zlib --build** Build all packages from source, except for zlib.
  - e.g., **--build=!z* --build** Build all packages from source, except for those starting with `z`

If you omit the **--build** option, the `build_policy` attribute in *conanfile.py* will be looked up. If it is set to `missing` or `always`, this build option will be used, otherwise the command will behave like **--build=never** was set.

env variables

With the **-e** parameters you can define:

- Global environment variables (**-e SOME_VAR="SOME_VALUE"**). These variables will be defined before the *build* step in all the packages and will be cleaned after the *build* execution.
- Specific package environment variables (**-e zlib:SOME_VAR="SOME_VALUE"**). These variables will be defined only in the specified packages (e.g., zlib).

You can specify this variables not only for your direct `requires` but for any package in the dependency graph.

If you want to define an environment variable but you want to append the variables declared in your requirements you can use the [] syntax:

```
$ conan install . -e PATH=[/other/path]
```

This way the first entry in the `PATH` variable will be */other/path* but the `PATH` values declared in the requirements of the project will be appended at the end using the system path separator.

settings

With the **-s** parameters you can define:

- Global settings (**-s compiler="Visual Studio"**). Will apply to all the requires.
- Specific package settings (**-s zlib:compiler="MinGW"**). Those settings will be applied only to the specified packages. They accept patterns too, like `-s *@myuser/*:compiler=MinGW`, which means that packages that have the username "myuser" will use MinGW as compiler.
- **Experimental:** Settings only for the consumer package. (**-s &:compiler="MinGW"**). If *&* is specified as the package name it will apply only to the consumer conanfile (.py or .txt). This is a special case because the consumer conanfile might not declare a *name* so it would be impossible to reference it.

You can specify custom settings not only for your direct `requires` but for any package in the dependency graph.

options

With the **-o** parameters you can only define specific package options.

```
$ conan install . -o zlib:shared=True
$ conan install . -o zlib:shared=True -o bzip2:option=132
# you can also apply the same options to many packages with wildcards:
$ conan install . -o *:shared=True
```

Experimental: To define an option just for the consumer conanfile.py use **-o &:shared=True** syntax. If *&* is specified as the package name it will apply only to the consumer conanfile.py. This is a special case because the consumer conanfile might not declare a *name* so it would be impossible to reference it.

Note: You can use *profiles* files to create predefined sets of **settings**, **options** and **environment variables**.

---

folders

Warning: This is an **experimental** feature subject to breaking changes in future releases.

The `--output-folder` define together with the `layout()` recipe method the location of the output files. For example, the files created by build system integrations such as `CMakeToolchain` or `PkgConfigDeps` will be created in the folder defined by the `layout()` `generators` folder, inside the defined `--output-folder`. By default, the `--output-folder` is the folder containing the `conanfile.py`.

conf

Warning: This is an **experimental** feature subject to breaking changes in future releases.

With the **-c** parameters you can define specific tool configurations.

```
$ conan install . -c tools.microsoft.msbuild:verbosity=Diagnostic
$ conan install . -c tools.microsoft.msbuild:verbosity=Detailed -c tools.
↪build:processes=10
```

Note: To list all possible configurations available, run **conan config list**.

See also:

You can see more information about configurations in *global.conf section*.

reference

An optional positional argument, if used the first argument should be a path. If the reference specifies name and/or version, and they are also declared in the `conanfile.py`, they should match, otherwise, an error will be raised.

```
$ conan install . # OK, user and channel will be None
$ conan install . user/testing # OK
$ conan install . version@user/testing # OK
$ conan install . pkg/version@user/testing # OK
$ conan install pkg/version@user/testing user/channel # Error, first arg is not a path
```

lockfiles

The `install` command accepts several arguments related to *lockfiles*:

- `--lockfile=<path-to-lockfile>`: The `conan install ... --lockfile=path/to/file.lock` command will provide an input lockfile to the command. Versions, revisions, and other data contained in that lockfile will be respected. If something has changed locally that diverges with respect the locked information in the lockfile, the command will fail.
- `--lockfile-out=<path-to-lockfile>`: This argument will define the filename of the resulting `install` operation. If the input lockfile has not completely locked something, and the install command can, for example, build some dependency from source with the `--build=<dep-name>` argument, this will provide new data, like a new package revision. This new data can be captured and locked in the output lockfile.
- `--lockfile-node-id=<node-id>`: **Experimental, subject to breaking changes**. In some cases, it is impossible to reference a package in the dependency graph by name or reference, because there might be several instances of it with the same one. This could happen with some special type of requirements, like build-requires or private requires. Providing the `node-id`, as defined in the lockfile file, can define without any ambiguity the package in the graph that the command is referencing.

Note: Installation of binaries can be accelerated setting up parallel downloads with the `general.parallel_download` **experimental** configuration in *conan.conf*.

–build-require

The `--build-require`, new in Conan 1.37, is experimental. It allows to install the package using the configuration and settings of the "build" context, as it was a `build_require`. Lets see it with an example:

We have a `mycmake/1.0` package, which bundles cmake executable, and we are cross-compiling from Windows to Linux, so all the usual install commands will use something like `-pr:b=Windows -pr:h=Linux`. At some point we might want to install the `build-require` to test it, executing it directly in the terminal, with `-build-require` it is possible:

```
$ conan install mycmake/1.0@ --build-require -g virtualenv -pr:b=Windows -pr:h=Linux
# Installs Windows package binary, not the Linux one.
$ source ./activate.sh && mycmake
# This will execute the "mycmake" from the Windows package.
```

This also works when building a dependency graph, including build-requires, in CI. As the `conan lock build-order` command will return a list including the build/host context, it is possible to use that to add the `--build-require` to the command, and build `build-requires` as necessary without needing to change the profiles at all.

–require-override

Warning: This is an **experimental** feature subject to breaking changes in future releases.

New from **Conan 1.39**

The `--require-override` argument allows to inject an override requirement to the consumer conanfile being called by this command, that would be equivalent to:

```
class Pkg(ConanFile):

    def requirements(self):
        self.requires("zlib/1.3", override=True)
```

This allows to dynamically test specific versions upstream without requiring editions to conanfiles. Note however this would not be a generally recommended practice for production, it would be better to actually update the conanfiles to explicitly reflect in code which specific versions upstream are being used.

If the consumer conanfile already contains a direct requirement to that dependency, then such version will be directly overwritten, but no `override=True` will be added (note that `override=True` means that the current package does not depend on that other package).

This feature affects only to regular `requires`, not to `tool_requires` or `python_requires`, as those don't have such an overriding mechanism, and they are private to their consumer, not propagating downstream nor upstream.

conan config

```
$ conan config [-h] {get,home,install,rm,set,init,list} ...
```

Manages Conan configuration.

Used to edit conan.conf, or install config files.

```
positional arguments:
  {get,home,install,rm,set,init,list}
                        sub-command help
    get                 Get the value of configuration item
    home                Retrieve the Conan home directory
    install             Install a full configuration from a local or remote
                        zip file
    rm                  Remove an existing config element
    set                 Set a value for a configuration item
    init                Initializes Conan configuration files
    list                List Conan configuration properties

optional arguments:
  -h, --help            show this help message and exit
```

Examples

- Change the logging level to 10:

  ```
  $ conan config set log.level=10
  ```

- Get the logging level:

```
$ conan config get log.level
$> 10
```

- Get the Conan home directory:

```
$ conan config home
$> /home/user/.conan
```

- Create all missing configuration files:

```
$ conan config init
```

- Delete the existing configuration files and create all configuration files:

```
$ conan config init --force
```

- List all possible properties allowed for *global.conf*

```
$ conan config list
```

- Set config install scheduler for every 1 week:

```
$ conan config set general.config_install_interval=1w
```

conan config install

```
usage: conan config install [-h] [--verify-ssl [VERIFY_SSL]] [--type {git}]
                            [--args ARGS] [-sf SOURCE_FOLDER] [-tf TARGET_FOLDER]
                            [-l] [-r REMOVE]
                            [item]

positional arguments:
  item                  git repository, local file or folder or zip file (local or
                        http) where the configuration is stored

optional arguments:
  -h, --help            show this help message and exit
  --verify-ssl [VERIFY_SSL]
                        Verify SSL connection when downloading file
  --type {git,dir,file,url}, -t {git,dir,file,url}
                        Type of remote config
  --args ARGS, -a ARGS  String with extra arguments for "git clone"
  -sf SOURCE_FOLDER, --source-folder SOURCE_FOLDER
                        Install files only from a source subfolder from the
                        specified origin
  -tf TARGET_FOLDER, --target-folder TARGET_FOLDER
                        Install to that path in the conan cache
  -l, --list            List stored configuration origins
  -r REMOVE, --remove REMOVE
                        Remove configuration origin by index in list (index provided by -
↪-list argument)
```

The `config install` is intended to share the Conan client configuration. For example, in a company or organization, is important to have common `settings.yml`, `profiles`, etc.

It can install one specific file or get its configuration files from a local or remote zip file, from a local directory or from a git repository. It then installs the files in the local Conan configuration.

The configuration may contain all or a subset of the allowed configuration files. Only the files that are present will be replaced. The only exception is the *conan.conf* file for which only the variables declared will be installed, leaving the other variables unchanged.

This means for example that **profiles** and **hooks** files will be overwritten if already present, but no profile or hook file that the user has in the local machine will be deleted.

All the configuration files will be copied to the Conan home directory. These are the special files and the rules applied to merge them:

File	How it is applied
profiles/MyProfile	Overrides the local ~/.conan/profiles/MyProfile if already exists
settings.yml	Overrides the local ~/.conan/settings.yml
remotes.txt	Overrides remotes. Will remove remotes that are not present in file
config/conan.conf	Merges the variables, overriding only the declared variables
hooks/my_hook.py	Overrides the local ~/.conan/hooks/my_hook.py if already exists

The file *remotes.txt* is the only file listed above which does not have a direct counterpart in the *~/.conan* folder. Its format is a list of entries, one on each line, with the form of

```
[remote name] [remote url] [bool]
```

where `[bool]` (either `True` or `False`) indicates whether SSL should be used to verify that remote. The remote definitions can be found in the *remotes.json* file and it provides a helpful starting point when writing the *remotes.txt* to be packaged in a Conan client configuration.

Note: During the installation, Conan skips any file with the name *README.md* or *LICENSE.txt*.

The **conan config install <item>** calls are stored in a *config_install.json* file in the Conan local cache. That allows to issue a **conan config install** command, without arguments, to iterate over the cached configurations, executing them again (updating).

The **conan config install** can be periodically executed, before any command, when *config_install_interval* is configured in *conan.conf*. Conan runs it based on *config_install.json*, including the timestamp of the last change.

Examples:

- Install the configuration from a URL:

```
$ conan config install http://url/to/some/config.zip
```

- Install the configuration from a URL, but only getting the files inside a *origin* folder inside the zip file, and putting them inside a *target* folder in the local cache:

```
$ conan config install http://url/to/some/config.zip -sf=origin -tf=target
```

- Install configuration from 2 different zip files from 2 different urls, using different source and target folders for each one, then update all:

```
$ conan config install http://url/to/some/config.zip -sf=origin -tf=target
$ conan config install http://url/to/some/config.zip -sf=origin2 -tf=target2
$ conan config install http://other/url/to/other.zip -sf=hooks -tf=hooks
# Later on, execute again the previous configurations cached:
$ conan config install
```

It's not needed to specify any argument, it will iterate previously stored configurations in *config_install.json*, executing them again.

- Install the configuration from a Git repository with submodules:

```
$ conan config install http://github.com/user/conan_config/.git --args "--recursive"
```

You can also force the git download by using **--type git** (in case it is not deduced from the URL automatically):

```
$ conan config install http://github.com/user/conan_config/.git --type git
```

- Install from a URL skipping SSL verification:

```
$ conan config install http://url/to/some/config.zip --verify-ssl=False
```

This will disable the SSL check of the certificate.

- Install a specific file from a local path:

```
$ conan config install my_settings\settings.yml
```

- Install the configuration from a local path:

```
$ conan config install /path/to/some/config.zip
```

- List all previously installed origins (the ones that will be used if **conan config install** is called without args):

```
$ conan config install --list
```

This will display the list of stored origins, with their index inside the list.

- Remove one of the previously installed origins:

```
$ conan config install --remove=1
```

This will remove the element with index=1 (second element in the list) of the existing origins. This means that the next **conan config install** manual or scheduled calls to this command will not use this origin anymore.

conan get

```
$ conan get [-h] [-p PACKAGE] [-r REMOTE] [-raw] reference [path]
```

Gets a file or list a directory of a given reference or package.

```
positional arguments:
  reference             Recipe reference or package reference e.g.,
                        'MyPackage/1.2@user/channel', 'MyPackage/1.2@user/chan
                        nel:af7901d8bdfde621d086181aa1c495c25a17b137'
```

(continues on next page)

(continued from previous page)

```
  path                  Path to the file or directory. If not specified will
                        get the conanfile if only a reference is specified and
                        a conaninfo.txt file contents if the package is also
                        specified

optional arguments:
  -h, --help            show this help message and exit
  -p PACKAGE, --package PACKAGE
                        Package ID [DEPRECATED: use full reference instead]
  -r REMOTE, --remote REMOTE
                        Get from this specific remote
  -raw, --raw           Do not decorate the text
```

Examples:

- Print the conanfile.py from a remote package:

```
$ conan get zlib/1.2.8@ -r conancenter
```

- List the files for a local package recipe:

```
$ conan get zlib/1.2.11@ .

Listing directory '.':
 CMakeLists.txt
 conanfile.py
 conanmanifest.txt
```

- Print a file from a recipe folder:

```
$ conan get zlib/1.2.11@ conanmanifest.txt
```

- Print the conaninfo.txt file for a binary package:

```
$ conan get zlib/1.2.11@:2144f833c251030c3cfd61c4354ae0e38607a909
```

```
[settings]
    arch=x86_64
    build_type=Release
    compiler=apple-clang
    compiler.version=8.1
    os=Macos

[requires]


[options]
    # ...
```

- List the files from a binary package in a remote:

```
$ conan get zlib/1.2.11@:ff82a1e70ba8430648a79986385b20a3648f8c19 . -r conancenter
```

(continues on next page)

(continued from previous page)

```
Listing directory '.':
 conan_package.tgz
 conaninfo.txt
 conanmanifest.txt
```

conan info

```
$ conan info [-h] [--paths] [-bo BUILD_ORDER] [-g GRAPH]
             [-if INSTALL_FOLDER] [-j [JSON]] [-n ONLY]
             [--package-filter [PACKAGE_FILTER]] [-db [DRY_BUILD]]
             [-b [BUILD]] [-r REMOTE] [-u] [-l LOCKFILE]
             [--lockfile-out LOCKFILE_OUT] [-e ENV_HOST] [-e:b ENV_BUILD]
             [-e:h ENV_HOST] [-o OPTIONS_HOST] [-o:b OPTIONS_BUILD]
             [-o:h OPTIONS_HOST] [-pr PROFILE_HOST] [-pr:b PROFILE_BUILD]
             [-pr:h PROFILE_HOST] [-s SETTINGS_HOST]
             [-s:b SETTINGS_BUILD] [-s:h SETTINGS_HOST]
             [-c CONF_HOST] [-c:b CONF_BUILD] [-c:h CONF_HOST]
             path_or_reference
```

Gets information about the dependency graph of a recipe.

It can be used with a recipe or a reference for any existing package in your local cache.

```
positional arguments:
  path_or_reference     Path to a folder containing a recipe (conanfile.py or
                        conanfile.txt) or to a recipe file. e.g.,
                        ./my_project/conanfile.txt. It could also be a
                        reference

optional arguments:
  -h, --help            show this help message and exit
  --paths               Show package paths in local cache
  -bo BUILD_ORDER, --build-order BUILD_ORDER
                        given a modified reference, return an ordered list to
                        build (CI). [DEPRECATED: use 'conan lock build-order
                        ...' instead]
  -g GRAPH, --graph GRAPH
                        Creates file with project dependencies graph. It will
                        generate a DOT or HTML file depending on the filename
                        extension
  -if INSTALL_FOLDER, --install-folder INSTALL_FOLDER
                        local folder containing the conaninfo.txt and
                        conanbuildinfo.txt files (from a previous conan
                        install execution). Defaulted to current folder,
                        unless --profile, -s or -o is specified. If you
                        specify both install-folder and any setting/option it
                        will raise an error.
  -j [JSON], --json [JSON]
                        Path to a json file where the information will be
                        written
  -n ONLY, --only ONLY  Show only the specified fields: "id", "build_id",
```

(continues on next page)

(continued from previous page)

```
                      "remote", "url", "license", "requires", "update",
                      "required", "date", "author", "description",
                      "provides", "deprecated", "None". '--paths'
                      information can also be filtered with options
                      "export_folder", "build_folder", "package_folder",
                      "source_folder". Use '--only None' to show only
                      references.
--package-filter [PACKAGE_FILTER]
                      Print information only for packages that match the
                      filter pattern e.g., MyPackage/1.2@user/channel or
                      MyPackage*
-db [DRY_BUILD], --dry-build [DRY_BUILD]
                      Apply the --build argument to output the information,
                      as it would be done by the install command
-b [BUILD], --build [BUILD]
                      Given a build policy, return an ordered list of
                      packages that would be built from sources during the
                      install command
-r REMOTE, --remote REMOTE
                      Look in the specified remote server
-u, --update          Will check if updates of the dependencies exist in the
                      remotes (a new version that satisfies a version range,
                      a new revision or a newer recipe if not using
                      revisions).
-l LOCKFILE, --lockfile LOCKFILE
                      Path to a lockfile
--lockfile-out LOCKFILE_OUT
                      Filename of the updated lockfile
-e ENV_HOST, --env ENV_HOST
                      Environment variables that will be set during the
                      package build (host machine). e.g.: -e
                      CXX=/usr/bin/clang++
-e:b ENV_BUILD, --env:build ENV_BUILD
                      Environment variables that will be set during the
                      package build (build machine). e.g.: -e:b
                      CXX=/usr/bin/clang++
-e:h ENV_HOST, --env:host ENV_HOST
                      Environment variables that will be set during the
                      package build (host machine). e.g.: -e:h
                      CXX=/usr/bin/clang++
-o OPTIONS_HOST, --options OPTIONS_HOST
                      Define options values (host machine), e.g.: -o
                      Pkg:with_qt=true
-o:b OPTIONS_BUILD, --options:build OPTIONS_BUILD
                      Define options values (build machine), e.g.: -o:b
                      Pkg:with_qt=true
-o:h OPTIONS_HOST, --options:host OPTIONS_HOST
                      Define options values (host machine), e.g.: -o:h
                      Pkg:with_qt=true
-pr PROFILE_HOST, --profile PROFILE_HOST
                      Apply the specified profile to the host machine
-pr:b PROFILE_BUILD, --profile:build PROFILE_BUILD
```

(continues on next page)

(continued from previous page)

```
                        Apply the specified profile to the build machine
  -pr:h PROFILE_HOST, --profile:host PROFILE_HOST
                        Apply the specified profile to the host machine
  -s SETTINGS_HOST, --settings SETTINGS_HOST
                        Settings to build the package, overwriting the
                        defaults (host machine). e.g.: -s compiler=gcc
  -s:b SETTINGS_BUILD, --settings:build SETTINGS_BUILD
                        Settings to build the package, overwriting the
                        defaults (build machine). e.g.: -s:b compiler=gcc
  -s:h SETTINGS_HOST, --settings:host SETTINGS_HOST
                        Settings to build the package, overwriting the
                        defaults (host machine). e.g.: -s:h compiler=gcc
  -c CONF_HOST, --conf CONF_HOST
                        Configuration to build the package, overwriting the defaults␣
↪(host machine). e.g.: -c
                        tools.cmake.cmaketoolchain:generator=Xcode
  -c:b CONF_BUILD, --conf:build CONF_BUILD
                        Configuration to build the package, overwriting the defaults␣
↪(build machine). e.g.: -c:b
                        tools.cmake.cmaketoolchain:generator=Xcode
  -c:h CONF_HOST, --conf:host CONF_HOST
                        Configuration to build the package, overwriting the defaults␣
↪(host machine). e.g.: -c:h
                        tools.cmake.cmaketoolchain:generator=Xcode
```

Examples:

```
$ conan info .
$ conan info myproject_folder
$ conan info myproject_folder/conanfile.py
$ conan info hello/1.0@user/channel
```

The output will look like:

```
Dependency/0.1@user/channel
 ID: 5ab84d6acfe1f23c4fae0ab88f26e3a396351ac9
 BuildID: None
 Context: host
 Remote: None
 URL: http://...
 License: MIT
 Description: A common dependency
 Updates: Version not checked
 Creation date: 2017-10-31 14:45:34
 Required by:
    hello/1.0@user/channel

hello/1.0@user/channel
 ID: 5ab84d6acfe1f23c4fa5ab84d6acfe1f23c4fa8
 BuildID: None
 Context: host
 Remote: None
```

(continues on next page)

(continued from previous page)

```
URL: http://...
License: MIT
Description: Hello World!
Updates: Version not checked
Required by:
   Project
Requires:
   hello0/0.1@user/channel
```

conan info builds the complete dependency graph, like **conan install** does. The main difference is that it doesn't try to install or build the binaries, but the package recipes will be retrieved from remotes if necessary.

It is very important to note, that the **info** command outputs the dependency graph for a given configuration (settings, options), as the dependency graph can be different for different configurations. Then, the input to the **conan info** command is the same as **conan install**, the configuration can be specified directly with settings and options, or using profiles.

Also, if you did a previous **conan install** with a specific configuration, or maybe different installs with different configurations, you can reuse that information with the **--install-folder** argument:

```
$ # dir with a conanfile.txt
$ mkdir build_release && cd build_release
$ conan install .. --profile=gcc54release
$ cd .. && mkdir build_debug && cd build_debug
$ conan install .. --profile=gcc54debug
$ cd ..
$ conan info . --install-folder=build_release
> info for the release dependency graph install
$ conan info . --install-folder=build_debug
> info for the debug dependency graph install
```

It is possible to use the **conan info** command to extract useful information for Continuous Integration systems. More precisely, it has the **--build-order, -bo** option (deprecated in favor of *conan lock build-order*), that will produce a machine-readable output with an ordered list of package references, in the order they should be built. E.g., let's assume that we have a project that depends on Boost and Poco, which in turn depends on OpenSSL and zlib transitively. So we can query our project with a reference that has changed (most likely due to a git push on that package):

```
$ conan info . -bo zlib/1.2.11@
[zlib/1.2.11], [openssl/1.0.2u], [boost/1.71.0, poco/1.9.4]
```

Note the result is a list of lists. When there is more than one element in one of the lists, it means that they are decoupled projects and they can be built in parallel by the CI system.

You can also specify the **--build-order=ALL** argument, if you want just to compute the whole dependency graph build order

```
$ conan info . --build-order=ALL
> [zlib/1.2.11], [openssl/1.0.2u], [boost/1.71.0, poco/1.9.4]
```

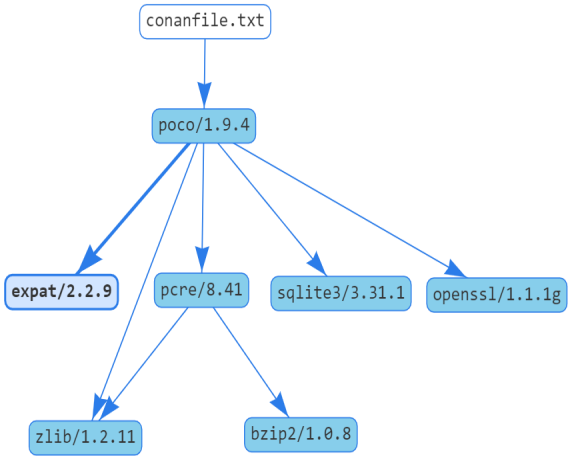
Also you can get a list of nodes that would be built (simulation) in an install command specifying a build policy with the `--build` parameter.

E.g., if I try to install `boost/1.71.0` recipe with `--build missing` build policy and `arch=x86`, which libraries will be built?

```
$ conan info boost/1.71.0@ --build missing -s arch=x86
bzip2/1.0.8, zlib/1.2.11, boost/1.71.0
```

You can generate a graph of your dependencies, in dot or html formats:

```
$ conan info .. --graph=file.html
$ file.html # or open the file, double-click
```



The generated html output contains links to third party resources, the *vis.js* library (2 files: *vis.min.js*, *vis.min.css*). By default they are retrieved from cloudfare. However, for environments without internet connection, these files could be also used from the local cache and installed with **conan config install** by putting those files in the root of the configuration folder:

- *vis.min.js*: Default link to "https://cdnjs.cloudflare.com/ajax/libs/vis/4.18.1/vis.min.js"
- *vis.min.css*: Default link to "https://cdnjs.cloudflare.com/ajax/libs/vis/4.18.1/vis.min.css"

It is not necessary to modify the generated html file. Conan will automatically use the local paths to the cache files if present, or the internet ones if not.

You can find where the package is installed in your cache by using the argument **--paths**:

```
$ conan info foobar/1.0.0@user/channel --paths
```

The output will look like:

```
foobar/1.0.0@user/channel
    ID: 6af9cc7cb931c5ad942174fd7838eb655717c709
    BuildID: None
    Context: host
    export_folder: /home/conan/.conan/data/foobar/1.0.0/user/channel/export
    source_folder: /home/conan/.conan/data/foobar/1.0.0/user/channel/source
    build_folder: /home/conan/.conan/data/foobar/1.0.0/user/channel/build/
↪6af9cc7cb931c5ad942174fd7838eb655717c709
    package_folder: /home/conan/.conan/data/foobar/1.0.0/user/channel/package/
↪6af9cc7cb931c5ad942174fd7838eb655717c709
    Remote: None
    License: MIT
    Description: Foobar project
```

(continues on next page)

(continued from previous page)

```
    Author: Dummy
    Topics: None
    Recipe: Cache
    Binary: Cache
    Binary remote: None
    Creation date: 2019-09-03 11:22:17
```

conan search

```
$ conan search [-h] [-o] [-q QUERY] [-r REMOTE] [--case-sensitive]
               [--raw] [--table TABLE] [-j JSON] [-rev]
               [pattern_or_reference]
```

Searches package recipes and binaries in the local cache or a remote. Unless a remote is specified only the local cache is searched.

If you provide a pattern, then it will search for existing package recipes matching it. If a full reference is provided (pkg/0.1@user/channel) then the existing binary packages for that reference will be displayed. The default remote is ignored, if no remote is specified, the search will be done in the local cache. Search is case sensitive, the exact case has to be used. For case insensitive file systems, like Windows, case sensitive search can be forced with '–case-sensitive'.

```
positional arguments:
  pattern_or_reference  Pattern or package recipe reference, e.g., 'boost/*',
                        'MyPackage/1.2@user/channel'

optional arguments:
  -h, --help            show this help message and exit
  -o, --outdated        Show only outdated from recipe packages. This flag can
                        only be used with a reference
  -q QUERY, --query QUERY
                        Packages query: 'os=Windows AND (arch=x86 OR
                        compiler=gcc)'. The 'pattern_or_reference' parameter
                        has to be a reference: MyPackage/1.2@user/channel
  -r REMOTE, --remote REMOTE
                        Remote to search in. '-r all' searches all remotes
  --case-sensitive      Make a case-sensitive search. Use it to guarantee
                        case-sensitive search in Windows or other case-
                        insensitive file systems
  --raw                 Print just the list of recipes
  --table TABLE         Outputs html file with a table of binaries. Only valid
                        for a reference search
  -j JSON, --json JSON  json file path where the search information will be
                        written to
  -rev, --revisions     Get a list of revisions for a reference or a package
                        reference.
```

Examples

```
$ conan search "zlib/*"
$ conan search "zlib/*" -r=conancenter
```

To search for recipes in all defined remotes use **-r all** (this is only valid for searching recipes, not binaries):

```
$ conan search "zlib/*" -r=all
```

If you use instead the full package recipe reference, you can explore the binaries existing for that recipe, also in a remote or in the local conan cache:

```
$ conan search boost/1.71.0@
```

Query syntax

A query syntax is allowed to look for specific binaries, you can use `AND` and `OR` operators and parenthesis, with settings and also options.

```
$ conan search boost/1.71.0@ -q arch=x86_64
$ conan search boost/1.71.0@ -q "(arch=x86_64 OR arch=ARM) AND (build_type=Release OR␣
↪os=Windows)"
```

Query syntax allows sub-settings, even for custom ones. e.g:

```
$ conan search boost/1.71.0@ -q "compiler=gcc AND compiler.version=9"
$ conan search boost/1.71.0@ -q "os=Linux AND os.distro=Ubuntu AND os.distro.version=19.
↪04"
```

If you specify a query filter for a setting and the package recipe is not restricted by this setting, Conan won't find the packages. e.g:

```
class MyRecipe(ConanFile):
    name = "my_recipe"
    settings = "arch",
```

```
$ conan search my_recipe/1.0@lasote/stable -q os=Windows
```

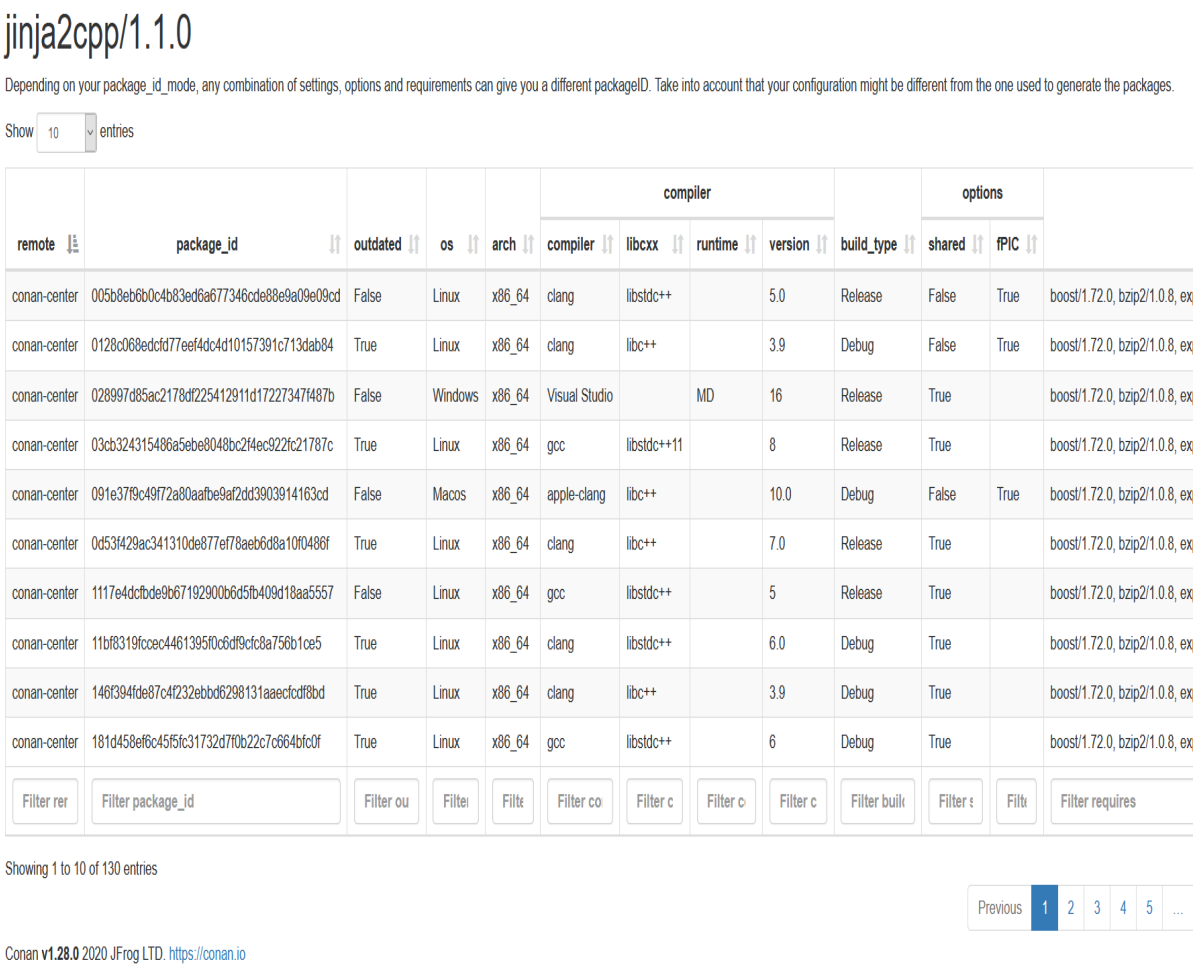
The query above won't find the `my_recipe` binary packages (because the recipe doesn't declare "os" as a setting) unless you specify the `None` value:

```
$ conan search my_recipe/1.0@lasote/stable -q os=None
```

Tabular output

You can generate a table for all binaries from a given recipe with the `--table` argument:

```
$ conan search jinja2cpp/1.1.0@ --table=file.html -r=conancenter
$ file.html # or open the file, double-click
```



jinja2cpp/1.1.0

Depending on your package_id_mode, any combination of settings, options and requirements can give you a different packageID. Take into account that your configuration might be different from the one used to generate the packages.

Show 10 entries

					compiler					options		
remote	package_id	outdated	os	arch	compiler	libcxx	runtime	version	build_type	shared	fPIC	
conan-center	005b8eb6b0c4b83ed6a677346cde88e9a09e09cd	False	Linux	x86_64	clang	libstdc++		5.0	Release	False	True	boost/1.72.0, bzip2/1.0.8, ex
conan-center	0128c068edcfd77eef4dc4d10157391c713dab84	True	Linux	x86_64	clang	libc++		3.9	Debug	False	True	boost/1.72.0, bzip2/1.0.8, ex
conan-center	028997d85ac2178df225412911d17227347f487b	False	Windows	x86_64	Visual Studio		MD	16	Release	True		boost/1.72.0, bzip2/1.0.8, ex
conan-center	03cb324315486a5ebe8048bc2f4ec922fc21787c	True	Linux	x86_64	gcc	libstdc++11		8	Release	True		boost/1.72.0, bzip2/1.0.8, ex
conan-center	091e37f9c49f72a80aafbe9af2dd3903914163cd	False	Macos	x86_64	apple-clang	libc++		10.0	Debug	False	True	boost/1.72.0, bzip2/1.0.8, ex
conan-center	0d53f429ac341310de877ef78aeb6d8a10f0486f	True	Linux	x86_64	clang	libc++		7.0	Release	True		boost/1.72.0, bzip2/1.0.8, ex
conan-center	1117e4dcfbde9b67192900b6d5fb409d18aa5557	False	Linux	x86_64	gcc	libstdc++		5	Release	True		boost/1.72.0, bzip2/1.0.8, ex
conan-center	11bf8319fccec4461395f0c6df9cfc8a756b1ce5	True	Linux	x86_64	clang	libstdc++		6.0	Debug	True		boost/1.72.0, bzip2/1.0.8, ex
conan-center	146f394fde87c4f232ebbd6298131aaecfcdf8bd	True	Linux	x86_64	clang	libc++		3.9	Debug	True		boost/1.72.0, bzip2/1.0.8, ex
conan-center	181d458ef6c45f5fc31732d7f0b22c7c664bfc0f	True	Linux	x86_64	gcc	libstdc++		6	Debug	True		boost/1.72.0, bzip2/1.0.8, ex
Filter rer	Filter package_id	Filter ou	Filte	Filte	Filter co	Filter c	Filter c	Filter c	Filter build	Filter s	Filte	Filter requires

Showing 1 to 10 of 130 entries

Previous 1 2 3 4 5 ...

Conan **v1.28.0** 2020 JFrog LTD. https://conan.io

Recipe and package revisions

Search all the local Conan packages matching a pattern and showing the revision:

```
$ conan search "lib*" --revisions
$ Existing package recipes:

  lib/1.0@user/channel#404e86c18e4a47a166fabe70b3b15e33
```

Search the local revision for a local cache recipe:

```
$ conan search lib/1.0@conan/testing --revisions
$ Revisions for 'lib/1.0@conan/testing':
```

(continues on next page)

(continued from previous page)

```
    a55e3b054fdbf4e2c6f10e955da69502 (2019-03-05 16:37:27 UTC)
```

Search the remote revisions in a server:

```
$ conan search lib/1.0@conan/testing --revisions -r=myremote
  Revisions for 'lib/1.0@conan/testing' at remote 'myremote':
    78fcef25a1eaeecd5facbbf08624c561 (2019-03-05 16:37:27 UTC)
    f3367e0e7d170aa12abccb175fee5f97 (2019-03-05 16:37:27 UTC)
```

18.1.2 Creator commands

Commands related to the creation of Conan recipes and packages:

conan create

```
$ conan create [-h] [-j JSON] [-k] [-kb] [-ne] [-tbf TEST_BUILD_FOLDER]
               [-tf TEST_FOLDER] [--ignore-dirty] [--build-require]
               [--require-override REQUIRE_OVERRIDE] [-m [MANIFESTS]]
               [-mi [MANIFESTS_INTERACTIVE]] [-v [VERIFY]] [-b [BUILD]]
               [-r REMOTE] [-u] [-l LOCKFILE]
               [--lockfile-out LOCKFILE_OUT] [-e ENV_HOST]
               [-e:b ENV_BUILD] [-e:h ENV_HOST] [-o OPTIONS_HOST]
               [-o:b OPTIONS_BUILD] [-o:h OPTIONS_HOST]
               [-pr PROFILE_HOST] [-pr:b PROFILE_BUILD]
               [-pr:h PROFILE_HOST] [-s SETTINGS_HOST]
               [-s:b SETTINGS_BUILD] [-s:h SETTINGS_HOST]
               [-c CONF_HOST] [-c:b CONF_BUILD] [-c:h CONF_HOST]
               path [reference]
```

Builds a binary package for a recipe (conanfile.py).

Uses the specified configuration in a profile or in -s settings, -o options, etc. If a 'test_package' folder (the name can be configured with -tf) is found, the command will run the consumer project to ensure that the package has been created correctly. Check 'conan test' command to know more about 'test_folder' project.

```
positional arguments:
  path                  Path to a folder containing a conanfile.py or to a
                        recipe file e.g., my_folder/conanfile.py
  reference             user/channel, version@user/channel or
                        pkg/version@user/channel (if name or version declared
                        in conanfile.py, they should match)

optional arguments:
  -h, --help            show this help message and exit
  -j JSON, --json JSON  json file path where the install information will be
                        written to
  -k, -ks, --keep-source
                        Do not remove the source folder in the local cache,
                        even if the recipe changed. Use this for testing
                        purposes only
  -kb, --keep-build     Do not remove the build folder in local cache. Implies
```

(continues on next page)

(continued from previous page)

```
                      --keep-source. Use this for testing purposes only
-ne, --not-export     Do not export the conanfile.py
-tbf TEST_BUILD_FOLDER, --test-build-folder TEST_BUILD_FOLDER
                      Working directory for the build of the test project.
-tf TEST_FOLDER, --test-folder TEST_FOLDER
                      Alternative test folder name. By default it is
                      "test_package". Use "None" to skip the test stage
--ignore-dirty        When using the "scm" feature with "auto" values,
                      capture the revision and url even if there are
                      uncommitted changes
--build-require       The provided reference is a build-require
--require-override REQUIRE_OVERRIDE
                      Define a requirement override
-m [MANIFESTS], --manifests [MANIFESTS]
                      Install dependencies manifests in folder for later
                      verify. Default folder is .conan_manifests, but can be
                      changed
-mi [MANIFESTS_INTERACTIVE], --manifests-interactive [MANIFESTS_INTERACTIVE]
                      Install dependencies manifests in folder for later
                      verify, asking user for confirmation. Default folder
                      is .conan_manifests, but can be changed
-v [VERIFY], --verify [VERIFY]
                      Verify dependencies manifests against stored ones
-b [BUILD], --build [BUILD]
                      Optional, specify which packages to build from source.
                      Combining multiple '--build' options on one command
                      line is allowed. For dependencies, the optional
                      'build_policy' attribute in their conanfile.py takes
                      precedence over the command line parameter. Possible
                      parameters: --build Force build for all packages, do
                      not use binary packages. --build=never Disallow build
                      for all packages, use binary packages or fail if a
                      binary package is not found. Cannot be combined with
                      other '--build' options. --build=missing Build
                      packages from source whose binary package is not
                      found. --build=outdated Build packages from source
                      whose binary package was not generated from the latest
                      recipe or is not found. --build=cascade Build packages
                      from source that have at least one dependency being
                      built from source. --build=[pattern] Build packages
                      from source whose package reference matches the
                      pattern. The pattern uses 'fnmatch' style wildcards.
                      --build=![pattern] Excluded packages, which will not
                      be built from the source, whose package reference
                      matches the pattern. The pattern uses 'fnmatch' style
                      wildcards. Default behavior: If you omit the '--build'
                      option, the 'build_policy' attribute in conanfile.py
                      will be used if it exists, otherwise the behavior is
                      like '--build=package name'.
-r REMOTE, --remote REMOTE
                      Look in the specified remote server
-u, --update          Will check the remote and in case a newer version
```

(continues on next page)

(continued from previous page)

```
                        and/or revision of the dependencies exists there, it
                        will install those in the local cache. When using
                        version ranges, it will install the latest version
                        that satisfies the range. Also, if using revisions, it
                        will update to the latest revision for the resolved
                        version range.
  -l LOCKFILE, --lockfile LOCKFILE
                        Path to a lockfile
  --lockfile-out LOCKFILE_OUT
                        Filename of the updated lockfile
  -e ENV_HOST, --env ENV_HOST
                        Environment variables that will be set during the
                        package build (host machine). e.g.: -e
                        CXX=/usr/bin/clang++
  -e:b ENV_BUILD, --env:build ENV_BUILD
                        Environment variables that will be set during the
                        package build (build machine). e.g.: -e:b
                        CXX=/usr/bin/clang++
  -e:h ENV_HOST, --env:host ENV_HOST
                        Environment variables that will be set during the
                        package build (host machine). e.g.: -e:h
                        CXX=/usr/bin/clang++
  -o OPTIONS_HOST, --options OPTIONS_HOST
                        Define options values (host machine), e.g.: -o
                        Pkg:with_qt=true
  -o:b OPTIONS_BUILD, --options:build OPTIONS_BUILD
                        Define options values (build machine), e.g.: -o:b
                        Pkg:with_qt=true
  -o:h OPTIONS_HOST, --options:host OPTIONS_HOST
                        Define options values (host machine), e.g.: -o:h
                        Pkg:with_qt=true
  -pr PROFILE_HOST, --profile PROFILE_HOST
                        Apply the specified profile to the host machine
  -pr:b PROFILE_BUILD, --profile:build PROFILE_BUILD
                        Apply the specified profile to the build machine
  -pr:h PROFILE_HOST, --profile:host PROFILE_HOST
                        Apply the specified profile to the host machine
  -s SETTINGS_HOST, --settings SETTINGS_HOST
                        Settings to build the package, overwriting the
                        defaults (host machine). e.g.: -s compiler=gcc
  -s:b SETTINGS_BUILD, --settings:build SETTINGS_BUILD
                        Settings to build the package, overwriting the
                        defaults (build machine). e.g.: -s:b compiler=gcc
  -s:h SETTINGS_HOST, --settings:host SETTINGS_HOST
                        Settings to build the package, overwriting the
                        defaults (host machine). e.g.: -s:h compiler=gcc
  -c CONF_HOST, --conf CONF_HOST
                        Configuration to build the package, overwriting the defaults␣
↪(host machine). e.g.: -c
                        tools.cmake.cmaketoolchain:generator=Xcode
  -c:b CONF_BUILD, --conf:build CONF_BUILD
                        Configuration to build the package, overwriting the defaults␣
```

(continues on next page)

(continued from previous page)

```
↪(build machine). e.g.: -c:b
                        tools.cmake.cmaketoolchain:generator=Xcode
  -c:h CONF_HOST, --conf:host CONF_HOST
                        Configuration to build the package, overwriting the defaults␣
↪(host machine). e.g.: -c:h
                        tools.cmake.cmaketoolchain:generator=Xcode
```

conan create . demo/testing is equivalent to:

```
$ conan export . demo/testing
$ conan install hello/0.1@demo/testing --build=hello
# package is created now, use test to test it
$ cd test_package
$ conan test . hello/0.1@demo/testing
```

Tip: Sometimes you need to **skip/disable test stage** to avoid a failure while creating the package, i.e: when you are cross compiling libraries and target code cannot be executed in current host platform. In that case you can skip/disable the test package stage:

```
$ conan create . demo/testing --test-folder=None
```

conan create executes methods of a *conanfile.py* in the following order:

1. `config_options()`
2. `configure()`
3. `requirements()`
4. `package_id()`
5. `build_requirements()`
6. `build_id()`
7. `system_requirements()`
8. `source()`
9. `imports()`
10. `build()`
11. `package()`
12. `package_info()`

In case of installing a pre-built binary, steps from 5 to 11 will be skipped. Note that `deploy()` method is only used in **conan install**.

Note: Installation of binaries can be accelerated setting up parallel downloads with the `general.parallel_download` **experimental** configuration in *conan.conf*.

The `--build-require`, new in Conan 1.37, is experimental. It allows to create the package using the configuration and settings of the "build" context, as it was a `build_require`. This feature allows to create packages in a way that is consistent to the way they will be used later. When there is a `test_package`, it is possible to use there the

`test_type="explicit"` and `self.test_requires(self.tested_reference_str)`. There is no need to provide it in the command line, *check "testing tool requires"* to know more.

–require-override

Warning: This is an **experimental** feature subject to breaking changes in future releases.

New from **Conan 1.39**.

This argument is the same, and has the same behavior as the *conan install command*.

conan export

```
$ conan export [-h] [-k] [-l LOCKFILE] [--lockfile-out LOCKFILE_OUT]
               [--ignore-dirty]
               path [reference]
```

Copies the recipe (conanfile.py & associated files) to your local cache.

Use the 'reference' param to specify a user and channel where to export it. Once the recipe is in the local cache it can be shared and reused with any remote with the 'conan upload' command.

```
positional arguments:
  path                  Path to a folder containing a conanfile.py or to a
                        recipe file e.g., my_folder/conanfile.py
  reference             user/channel, Pkg/version@user/channel (if name and
                        version are not declared in the conanfile.py)
                        Pkg/version@ if user/channel is not relevant.

optional arguments:
  -h, --help            show this help message and exit
  -k, -ks, --keep-source
                        Do not remove the source folder in the local cache,
                        even if the recipe changed. Use this for testing
                        purposes only
  -l LOCKFILE, --lockfile LOCKFILE
                        Path to a lockfile file.
  --lockfile-out LOCKFILE_OUT
                        Filename of the updated lockfile
  --ignore-dirty        When using the "scm" feature with "auto" values,
                        capture the revision and url even if there are
                        uncommitted changes
```

The `reference` field can be:

- A complete package reference: `pkg/version@user/channel`. In this case, the recipe doesn't need to declare the name or the version. If the recipe declares them, they should match the provided values in the command line.
- The user and channel: `user/channel`. The command will assume that the name and version are provided by the recipe.
- The version, user and channel: `version@user/channel`. The recipe must provide the name, and if it does provide the version, it should match the command line one.

There is also a "recipe_linter" hook in the official hooks repository that can be activated to run automatic linter checks on the recipes when they are exported.

Examples

- Export a recipe using a full reference. Only valid if `name` and `version` are not declared in the recipe:

```
$ conan export . mylib/1.0@myuser/channel
```

- Same as above, but without any `user/channel`. The ending `@` is here to disambiguate from the `user/channel` part:

```
$ conan export . mylib/1.0@
```

- Export a recipe from any folder directory, under the `myuser/stable` user and channel:

```
$ conan export ./folder_name myuser/stable
```

- Export a recipe without removing the source folder in the local cache:

```
$ conan export . fenix/stable -k
```

conan export-pkg

```
$ conan export-pkg [-h] [-bf BUILD_FOLDER] [-f] [-if INSTALL_FOLDER]
                   [-pf PACKAGE_FOLDER] [-sf SOURCE_FOLDER] [-j JSON]
                   [-l LOCKFILE] [--lockfile-out LOCKFILE_OUT]
                   [--ignore-dirty] [-e ENV_HOST] [-e:b ENV_BUILD]
                   [-e:h ENV_HOST] [-o OPTIONS_HOST] [-o:b OPTIONS_BUILD]
                   [-o:h OPTIONS_HOST] [-pr PROFILE_HOST]
                   [-pr:b PROFILE_BUILD] [-pr:h PROFILE_HOST]
                   [-s SETTINGS_HOST] [-s:b SETTINGS_BUILD] [-s:h SETTINGS_HOST]
                   [-c CONF_HOST] [-c:b CONF_BUILD] [-c:h CONF_HOST]
                   path [reference]
```

Exports a recipe, then creates a package from local source and build folders.

If '–package-folder' is provided it will copy the files from there, otherwise, it will execute package() method over '–source-folder' and '–build-folder' to create the binary package.

```
positional arguments:
  path                  Path to a folder containing a conanfile.py or to a
                        recipe file e.g., my_folder/conanfile.py
  reference             user/channel or pkg/version@user/channel (if name and
                        version are not declared in the conanfile.py)

optional arguments:
  -h, --help            show this help message and exit
  -bf BUILD_FOLDER, --build-folder BUILD_FOLDER
                        Directory for the build process. Defaulted to the
                        current directory. A relative path to the current
                        directory can also be specified
  -f, --force           Overwrite existing package if existing
  -if INSTALL_FOLDER, --install-folder INSTALL_FOLDER
```

(continues on next page)

(continued from previous page)

```
                      Directory containing the conaninfo.txt and
                      conanbuildinfo.txt files (from previous 'conan
                      install'). Defaulted to --build-folder If these files
                      are found in the specified folder and any of '-e',
                      '-o', '-pr' or '-s' arguments are used, it will raise
                      an error.
-pf PACKAGE_FOLDER, --package-folder PACKAGE_FOLDER
                      folder containing a locally created package. If a
                      value is given, it won't call the recipe 'package()'
                      method, and will run a copy of the provided folder.
-sf SOURCE_FOLDER, --source-folder SOURCE_FOLDER
                      Directory containing the sources. Defaulted to the
                      conanfile's directory. A relative path to the current
                      directory can also be specified
-j JSON, --json JSON  Path to a json file where the install information will
                      be written
-l LOCKFILE, --lockfile LOCKFILE
                      Path to a lockfile.
--lockfile-out LOCKFILE_OUT
                      Filename of the updated lockfile
--ignore-dirty        When using the "scm" feature with "auto" values,
                      capture the revision and url even if there are
                      uncommitted changes
-e ENV_HOST, --env ENV_HOST
                      Environment variables that will be set during the
                      package build (host machine). e.g.: -e
                      CXX=/usr/bin/clang++
-e:b ENV_BUILD, --env:build ENV_BUILD
                      Environment variables that will be set during the
                      package build (build machine). e.g.: -e:b
                      CXX=/usr/bin/clang++
-e:h ENV_HOST, --env:host ENV_HOST
                      Environment variables that will be set during the
                      package build (host machine). e.g.: -e:h
                      CXX=/usr/bin/clang++
-o OPTIONS_HOST, --options OPTIONS_HOST
                      Define options values (host machine), e.g.: -o
                      Pkg:with_qt=true
-o:b OPTIONS_BUILD, --options:build OPTIONS_BUILD
                      Define options values (build machine), e.g.: -o:b
                      Pkg:with_qt=true
-o:h OPTIONS_HOST, --options:host OPTIONS_HOST
                      Define options values (host machine), e.g.: -o:h
                      Pkg:with_qt=true
-pr PROFILE_HOST, --profile PROFILE_HOST
                      Apply the specified profile to the host machine
-pr:b PROFILE_BUILD, --profile:build PROFILE_BUILD
                      Apply the specified profile to the build machine
-pr:h PROFILE_HOST, --profile:host PROFILE_HOST
                      Apply the specified profile to the host machine
-s SETTINGS_HOST, --settings SETTINGS_HOST
                      Settings to build the package, overwriting the
```

(continues on next page)

(continued from previous page)

```
                        defaults (host machine). e.g.: -s compiler=gcc
  -s:b SETTINGS_BUILD, --settings:build SETTINGS_BUILD
                        Settings to build the package, overwriting the
                        defaults (build machine). e.g.: -s:b compiler=gcc
  -s:h SETTINGS_HOST, --settings:host SETTINGS_HOST
                        Settings to build the package, overwriting the
                        defaults (host machine). e.g.: -s:h compiler=gcc
  -c CONF_HOST, --conf CONF_HOST
                        Configuration to build the package, overwriting the defaults␣
↪(host machine). e.g.: -c
                        tools.cmake.cmaketoolchain:generator=Xcode
  -c:b CONF_BUILD, --conf:build CONF_BUILD
                        Configuration to build the package, overwriting the defaults␣
↪(build machine). e.g.: -c:b
                        tools.cmake.cmaketoolchain:generator=Xcode
  -c:h CONF_HOST, --conf:host CONF_HOST
                        Configuration to build the package, overwriting the defaults␣
↪(host machine). e.g.: -c:h
                        tools.cmake.cmaketoolchain:generator=Xcode
```

The **export-pkg** command let you create a package from already existing files in your working folder, it can be useful if you are using a build process external to Conan and do not want to provide it with the recipe. Nevertheless, you should take into account that it will generate a package and Conan won't be able to guarantee its reproducibility or regenerate it again. This is **not** the normal or recommended flow for creating Conan packages.

Execution of this command will result in several files copied to the package folder in the cache identified by its `package_id` (Conan will perform all the required actions to compute this _id_: build the graph, evaluate the requirements and options, and call any required method), but there could be two different sources for the files:

- If the argument `--package-folder` is provided, Conan will just copy all the contents of that folder to the package one in the cache.
- If no `--package-folder` is given, Conan will execute the method `package()` once and the `self.copy(...)` functions will copy matching files from the `source_folder` **and** `build_folder` to the corresponding path in the Conan cache (working directory corresponds to the `build_folder`).
- If the arguments `--package-folder`, `--build-folder` or `--source-folder` are declared, but the path is incorrect, **export-pkg** will raise an exception.

There are different scenarios where this command could look like useful:

- You are *working locally on a package* and you want to upload it to the cache to be able to consume it from other recipes. In this situation you can use the **export-pkg** command to copy the package to the cache, but you could also put the *package in editable mode* and avoid this extra step.
- You only have precompiled binaries available, then you can use the **export-pkg** to create the Conan package, or you can build a working recipe to download and package them. These scenarios are described in the documentation section *How to package existing binaries*.

Packages created with `conan export-pkg` cannot be rebuilt from sources in the cache with the `--build` command line argument. It is possible to specify the class attribute `build_policy="never"` in this recipes (this is an experimental feature, available from Conan 1.37) to avoid the `--build=*` or `--build` argument to try to rebuild them from sources as part of a dependency graph.

Note: Note that if **--profile**, settings or options are not provided to **export-pkg**, the configuration will be extracted from the information stored after a previous **conan install**. That information might be incomplete in some edge

cases, so we strongly recommend the usage of **--profile** or **--settings, --options**, etc.

Examples

- Create a package from a directory containing the binaries for Windows/x86/Release:

  We need to collect all the files from the local filesystem and tell Conan to compute the proper `package_id` so its get associated with the correct settings and it works when consuming it.

  If the files in the working folder are:

```
Release_x86/lib/libmycoollib.a
Release_x86/lib/other.a
Release_x86/include/mylib.h
Release_x86/include/other.h
```

  then, just run:

```
$ conan new hello/0.1 --bare  # It creates a minimum recipe example
$ conan export-pkg . hello/0.1@user/stable -s os=Windows -s arch=x86 -s build_
↪type=Release --package-folder=Release_x86
```

  This last command will copy all the contents from the `package-folder` and create the package associated with the settings provided through the command line.

- Create a package from a source and build folder:

  The objective is to collect the files that will be part of the package from the source folder (*include files*) and from the build folder (libraries), so, if these are the files in the working folder:

```
sources/include/mylib.h
sources/src/file.cpp
build/lib/mylib.lib
build/lib/mylib.tmp
build/file.obj
```

  we would need a slightly more complicated *conanfile.py* than in the previous example to select which files to copy, we need to change the patterns in the `package()` method:

```
def package(self):
    self.copy("*.h", dst="include", src="include")
    self.copy("*.lib", dst="lib", keep_path=False)
```

  Now, we can run Conan to create the package:

```
$ conan export-pkg . hello/0.1@user/stable -pr:host=myprofile --source-
↪folder=sources --build-folder=build
```

conan new

```
$ conan new [-h] [-t] [-i] [-c] [-s] [-b] [-m TEMPLATE] [-cis] [-cilg]
            [-cilc] [-cio] [-ciw] [-ciglg] [-ciglc] [-ciccg] [-ciccc]
            [-cicco] [-gi] [-ciu CI_UPLOAD_URL]
            name
```

Creates a new package recipe template with a 'conanfile.py' and optionally, 'test_package' testing files.

```
positional arguments:
  name                  Package name, e.g.: "poco/1.9.4" or complete reference
                        for CI scripts: "poco/1.9.4@user/channel"

optional arguments:
  -h, --help            show this help message and exit
  -t, --test            Create test_package skeleton to test package
  -i, --header          Create a headers only package template
  -c, --pure-c          Create a C language package only package, deleting
                        "self.settings.compiler.libcxx" setting in the
                        configure method
  -s, --sources         Create a package with embedded sources in "src"
                        folder, using "exports_sources" instead of retrieving
                        external code with the "source()" method
  -b, --bare            Create the minimum package recipe, without build()
                        method. Useful in combination with "export-pkg"
                        command
  -m TEMPLATE, --template TEMPLATE
                        Use the given template to generate a conan project
  -cis, --ci-shared     Package will have a "shared" option to be used in CI
  -cilg, --ci-travis-gcc
                        Generate travis-ci files for linux gcc
  -cilc, --ci-travis-clang
                        Generate travis-ci files for linux clang
  -cio, --ci-travis-osx
                        Generate travis-ci files for OSX apple-clang
  -ciw, --ci-appveyor-win
                        Generate appveyor files for Appveyor Visual Studio
  -ciglg, --ci-gitlab-gcc
                        Generate GitLab files for linux gcc
  -ciglc, --ci-gitlab-clang
                        Generate GitLab files for linux clang
  -ciccg, --ci-circleci-gcc
                        Generate CircleCI files for linux gcc
  -ciccc, --ci-circleci-clang
                        Generate CircleCI files for linux clang
  -cicco, --ci-circleci-osx
                        Generate CircleCI files for OSX apple-clang
  -gi, --gitignore      Generate a .gitignore with the known patterns to
                        excluded
  -ciu CI_UPLOAD_URL, --ci-upload-url CI_UPLOAD_URL
                        Define URL of the repository to upload
  -d DEFINE, --define DEFINE
                        Define additional variables to be processed within
```

(continues on next page)

(continued from previous page)

```
                        template
```

Examples:

- Create a new `conanfile.py` for a new package **mypackage/1.0@myuser/stable**

```
$ conan new mypackage/1.0
```

- Create also a `test_package` folder skeleton:

```
$ conan new mypackage/1.0 -t
```

- Create files for travis (both Linux and OSX) and appveyor Continuous Integration:

```
$ conan new mypackage/1.0@myuser/stable -t -cilg -cio -ciw
```

- Create files for gitlab (linux) Continuous integration and set upload conan server:

```
$ conan new mypackage/1.0@myuser/stable -t -ciglg -ciglc -ciu https://myrepo.url
```

- Create files from a custom, predefined, user template recipe or template directory:

```
$ conan new mypackage/1.0 --template=myconanfile.py  # Single template file
$ conan new mypackage/1.0 --template=header_only  # Template directory
```

  Refer to the section *Package scaffolding for conan new command* for more information about these templates.

conan upload

```
$ conan upload [-h] [-p PACKAGE] [-q QUERY] [-r REMOTE] [--all]
               [--skip-upload] [--force] [--check] [-c] [--retry RETRY]
               [--retry-wait RETRY_WAIT] [-no [{all,recipe}]] [-j JSON]
               [--parallel]
               pattern_or_reference
```

Uploads a recipe and binary packages to a remote.

If no remote is specified, the first configured remote (by default conancenter, use 'conan remote list' to list the remotes) will be used.

```
positional arguments:
  pattern_or_reference  Pattern, recipe reference or package reference e.g.,
                        'boost/*', 'MyPackage/1.2@user/channel', 'MyPackage/1.
                        2@user/channel:af7901d8bdfde621d086181aa1c495c25a17b13
                        7'

optional arguments:
  -h, --help            show this help message and exit
  -p PACKAGE, --package PACKAGE
                        Package ID [DEPRECATED: use full reference instead]
  -q QUERY, --query QUERY
                        Only upload packages matching a specific query.
                        Packages query: 'os=Windows AND (arch=x86 OR
```

(continues on next page)

(continued from previous page)

```
                      compiler=gcc)'. The 'pattern_or_reference' parameter
                      has to be a reference: MyPackage/1.2@user/channel
  -r REMOTE, --remote REMOTE
                      upload to this specific remote
  --all               Upload both package recipe and packages
  --skip-upload       Do not upload anything, just run the checks and the
                      compression
  --force             Ignore checks before uploading the recipe: it will
                      bypass missing fields in the scm attribute and it will
                      override remote recipe with local regardless of recipe
                      date
  --check             Perform an integrity check, using the manifests,
                      before upload
  -c, --confirm       Upload all matching recipes without confirmation
  --retry RETRY       In case of fail retries to upload again the specified
                      times.
  --retry-wait RETRY_WAIT
                      Waits specified seconds before retry again
  -no [{all,recipe}], --no-overwrite [{all,recipe}]
                      Uploads package only if recipe is the same as the
                      remote one
  -j JSON, --json JSON  json file path where the upload information will be
                      written to
  --parallel          Upload files in parallel using multiple threads. The
                      default number of launched threads is set to the value
                      of cpu_count and can be configured using the
                      CONAN_CPU_COUNT environment variable or defining
                      cpu_count in conan.conf
```

Examples:

Uploads a package recipe (*conanfile.py* and the exported files):

```
$ conan upload OpenCV/1.4.0@lasote/stable
```

Uploads a package recipe and a single binary package:

```
$ conan upload OpenCV/1.4.0@lasote/stable:d50a0d523d98c15bb147b18fa7d203887c38be8b
```

Uploads a package recipe and all the generated binary packages to a specified remote:

```
$ conan upload OpenCV/1.4.0@lasote/stable --all -r my_remote
```

Uploads all recipes and binary packages from our local cache to `my_remote` without confirmation:

```
$ conan upload "*" --all -r my_remote -c
```

Uploads the recipe for OpenCV alongside any of its binary packages which are built with settings `arch=x86_64` and `os=Linux` from our local cache to `my_remote`:

```
$ conan upload OpenCV/1.4.0@lasote/stable -q 'arch=x86_64 and os=Linux' -r my_remote
```

Upload all local packages and recipes beginning with "Op" retrying 3 times and waiting 10 seconds between upload attempts:

```
$ conan upload "Op*" --all -r my_remote -c --retry 3 --retry-wait 10
```

Upload packages without overwriting the recipe and packages if the recipe has changed:

```
$ conan upload OpenCV/1.4.0@lasote/stable --all --no-overwrite  # defaults to --no-
↪overwrite all
```

Upload packages without overwriting the recipe if the packages have changed:

```
$ conan upload OpenCV/1.4.0@lasote/stable --all --no-overwrite recipe
```

Upload packages using multiple threads without requiring confirmation to my_remote. By default, the number of threads used is the number of cores available in the machine running Conan. It can also be configured setting the environment variable *CONAN_CPU_COUNT* or defining `cpu_count` in the *conan.conf*.

```
$ conan upload "*" --confirm --parallel -r my_remote
```

Warning: Note that *non_interactive mode* will be forced to *true* when using parallel upload

conan test

```
$ conan test [-h] [-tbf TEST_BUILD_FOLDER] [-b [BUILD]] [-r REMOTE] [-u]
             [-l LOCKFILE] [--lockfile-out LOCKFILE_OUT] [-e ENV_HOST]
             [-e:b ENV_BUILD] [-e:h ENV_HOST] [-o OPTIONS_HOST]
             [-o:b OPTIONS_BUILD] [-o:h OPTIONS_HOST] [-pr PROFILE_HOST]
             [-pr:b PROFILE_BUILD] [-pr:h PROFILE_HOST]
             [-s SETTINGS_HOST] [-s:b SETTINGS_BUILD] [-s:h SETTINGS_HOST]
             [-c CONF_HOST] [-c:b CONF_BUILD] [-c:h CONF_HOST]
             path reference
```

Tests a package consuming it from a conanfile.py with a test() method.

This command installs the conanfile dependencies (including the tested package), calls a 'conan build' to build test apps and finally executes the test() method. The testing recipe does not require name or version, neither definition of package() or package_info() methods. The package to be tested must exist in the local cache or any configured remote.

```
positional arguments:
  path                  Path to the "testing" folder containing a conanfile.py
                        or to a recipe file with test() method e.g. conan
                        test_package/conanfile.py pkg/version@user/channel
  reference             pkg/version@user/channel of the package to be tested

optional arguments:
  -h, --help            show this help message and exit
  -tbf TEST_BUILD_FOLDER, --test-build-folder TEST_BUILD_FOLDER
                        Working directory of the build process.
  -b [BUILD], --build [BUILD]
                        Optional, specify which packages to build from source.
                        Combining multiple '--build' options on one command
                        line is allowed. For dependencies, the optional
                        'build_policy' attribute in their conanfile.py takes
```

(continues on next page)

(continued from previous page)

```
                      precedence over the command line parameter. Possible
                      parameters: --build Force build for all packages, do
                      not use binary packages. --build=never Disallow build
                      for all packages, use binary packages or fail if a
                      binary package is not found. Cannot be combined with
                      other '--build' options. --build=missing Build
                      packages from source whose binary package is not
                      found. --build=outdated Build packages from source
                      whose binary package was not generated from the latest
                      recipe or is not found. --build=cascade Build packages
                      from source that have at least one dependency being
                      built from source. --build=[pattern] Build packages
                      from source whose package reference matches the
                      pattern. The pattern uses 'fnmatch' style wildcards.
                      --build=![pattern] Excluded packages, which will not
                      be built from the source, whose package reference
                      matches the pattern. The pattern uses 'fnmatch' style
                      wildcards. Default behavior: If you omit the '--build'
                      option, the 'build_policy' attribute in conanfile.py
                      will be used if it exists, otherwise the behavior is
                      like '--build=never'.
-r REMOTE, --remote REMOTE
                      Look in the specified remote server
-u, --update          Will check the remote and in case a newer version
                      and/or revision of the dependencies exists there, it
                      will install those in the local cache. When using
                      version ranges, it will install the latest version
                      that satisfies the range. Also, if using revisions, it
                      will update to the latest revision for the resolved
                      version range.
-l LOCKFILE, --lockfile LOCKFILE
                      Path to a lockfile
--lockfile-out LOCKFILE_OUT
                      Filename of the updated lockfile
-e ENV_HOST, --env ENV_HOST
                      Environment variables that will be set during the
                      package build (host machine). e.g.: -e
                      CXX=/usr/bin/clang++
-e:b ENV_BUILD, --env:build ENV_BUILD
                      Environment variables that will be set during the
                      package build (build machine). e.g.: -e:b
                      CXX=/usr/bin/clang++
-e:h ENV_HOST, --env:host ENV_HOST
                      Environment variables that will be set during the
                      package build (host machine). e.g.: -e:h
                      CXX=/usr/bin/clang++
-o OPTIONS_HOST, --options OPTIONS_HOST
                      Define options values (host machine), e.g.: -o
                      Pkg:with_qt=true
-o:b OPTIONS_BUILD, --options:build OPTIONS_BUILD
                      Define options values (build machine), e.g.: -o:b
                      Pkg:with_qt=true
```

(continues on next page)

(continued from previous page)

```
  -o:h OPTIONS_HOST, --options:host OPTIONS_HOST
                        Define options values (host machine), e.g.: -o:h
                        Pkg:with_qt=true
  -pr PROFILE_HOST, --profile PROFILE_HOST
                        Apply the specified profile to the host machine
  -pr:b PROFILE_BUILD, --profile:build PROFILE_BUILD
                        Apply the specified profile to the build machine
  -pr:h PROFILE_HOST, --profile:host PROFILE_HOST
                        Apply the specified profile to the host machine
  -s SETTINGS_HOST, --settings SETTINGS_HOST
                        Settings to build the package, overwriting the
                        defaults (host machine). e.g.: -s compiler=gcc
  -s:b SETTINGS_BUILD, --settings:build SETTINGS_BUILD
                        Settings to build the package, overwriting the
                        defaults (build machine). e.g.: -s:b compiler=gcc
  -s:h SETTINGS_HOST, --settings:host SETTINGS_HOST
                        Settings to build the package, overwriting the
                        defaults (host machine). e.g.: -s:h compiler=gcc
  -c CONF_HOST, --conf CONF_HOST
                        Configuration to build the package, overwriting the defaults␣
↪(host machine). e.g.: -c
                        tools.cmake.cmaketoolchain:generator=Xcode
  -c:b CONF_BUILD, --conf:build CONF_BUILD
                        Configuration to build the package, overwriting the defaults␣
↪(build machine). e.g.: -c:b
                        tools.cmake.cmaketoolchain:generator=Xcode
  -c:h CONF_HOST, --conf:host CONF_HOST
                        Configuration to build the package, overwriting the defaults␣
↪(host machine). e.g.: -c:h
                        tools.cmake.cmaketoolchain:generator=Xcode
```

This command is useful for testing existing packages, that have been previously built (with **conan create**, for example). **conan create** will automatically run this test if a *test_package* folder is found besides the *conanfile.py*, or if the **--test-folder** argument is provided to **conan create**.

Example:

```
$ conan new hello/0.1 -s -t
$ mv test_package test_package2
$ conan create . user/testing
# doesn't automatically run test, it has been renamed
# now run test
$ conan test test_package2 hello/0.1@user/testing
```

The test package folder, could be elsewhere, or could be even applied to different versions of the package.

18.1.3 Package development commands

Commands related to the local (user space) development of a Conan package:

conan source

```
$ conan source [-h] [-sf SOURCE_FOLDER] [-if INSTALL_FOLDER] path
```

Calls your local conanfile.py 'source()' method.

Usually downloads and uncompresses the package sources.

```
positional arguments:
  path                  Path to a folder containing a conanfile.py or to a
                        recipe file e.g., my_folder/conanfile.py

optional arguments:
  -h, --help            show this help message and exit
  -sf SOURCE_FOLDER, --source-folder SOURCE_FOLDER
                        Destination directory. Defaulted to current directory
  -if INSTALL_FOLDER, --install-folder INSTALL_FOLDER
                        Directory containing the conaninfo.txt and
                        conanbuildinfo.txt files (from previous 'conan
                        install'). Defaulted to --build-folder Optional,
                        source method will run without the information
                        retrieved from the conaninfo.txt and
                        conanbuildinfo.txt, only required when using
                        conditional source() based on settings, options,
                        env_info and user_info
```

The `source()` method might use (optional) *settings*, *options* and *environment variables* from the specified profile and dependencies information from the declared `deps_XXX_info` objects in the conanfile requirements.

All that information is saved automatically in the *conaninfo.txt* and *conanbuildinfo.txt* files respectively, when you run the **`conan install`** command. Those files have to be located in the specified **`--install-folder`**.

Examples:

- Call a local recipe's source method: In user space, the command will execute a local *conanfile.py* `source()` method, in the *mysrc* folder in the current directory.

```
$ conan new lib/1.0@conan/stable
$ conan source . --source-folder mysrc
```

- In case you need the settings/options or any info from the requirements, perform first an install:

```
$ conan install . --install-folder mybuild
$ conan source . --source-folder mysrc --install-folder mybuild
```

conan build

```
$ conan build [-h] [-b] [-bf BUILD_FOLDER] [-c] [-i] [-t]
              [-if INSTALL_FOLDER] [-pf PACKAGE_FOLDER]
              [-sf SOURCE_FOLDER]
              path
```

Calls your local conanfile.py 'build()' method.

The recipe will be built in the local directory specified by –build-folder, reading the sources from –source-folder. If you are using a build helper, like CMake(), the –package-folder will be configured as the destination folder for the install step.

```
positional arguments:
  path                  Path to a folder containing a conanfile.py or to a
                        recipe file e.g., my_folder/conanfile.py

optional arguments:
  -h, --help            show this help message and exit
  -b, --build           Execute the build step (variable should_build=True).
                        When specified, configure/install/test won't run
                        unless --configure/--install/--test specified
  -bf BUILD_FOLDER, --build-folder BUILD_FOLDER
                        Directory for the build process. Defaulted to the
                        current directory. A relative path to the current
                        directory can also be specified
  -c, --configure       Execute the configuration step (variable
                        should_configure=True). When specified,
                        build/install/test won't run unless
                        --build/--install/--test specified
  -i, --install         Execute the install step (variable
                        should_install=True). When specified,
                        configure/build/test won't run unless
                        --configure/--build/--test specified
  -t, --test            Execute the test step (variable should_test=True).
                        When specified, configure/build/install won't run
                        unless --configure/--build/--install specified
  -if INSTALL_FOLDER, --install-folder INSTALL_FOLDER
                        Directory containing the conaninfo.txt and
                        conanbuildinfo.txt files (from previous 'conan
                        install'). Defaulted to --build-folder
  -pf PACKAGE_FOLDER, --package-folder PACKAGE_FOLDER
                        Directory to install the package (when the build
                        system or build() method does it). Defaulted to the
                        '{build_folder}/package' folder. A relative path can
                        be specified, relative to the current folder. Also an
                        absolute path is allowed.
  -sf SOURCE_FOLDER, --source-folder SOURCE_FOLDER
                        Directory containing the sources. Defaulted to the
                        conanfile's directory. A relative path to the current
                        directory can also be specified
```

The `build()` method might use *settings*, *options* and *environment variables* from the specified profile and dependencies information from the declared `deps_XXX_info` objects in the conanfile requirements. All that information is

saved automatically in the *conaninfo.txt* and *conanbuildinfo.txt* files respectively, when you run the **conan install** command. Those files have to be located in the specified **--build-folder** or in the **--install-folder** if specified.

The **--configure, --build, --install** arguments control which parts of the `build()` are actually executed. They have related conanfile boolean variables `should_configure, should_build, should_install`, which are `True` by default, but that will change if some of these arguments are used in the command line. The `CMake` and `Meson` and `AutotoolsBuildEnvironment` helpers already use these variables.

Example: Building a conan package (for architecture x86) in a local directory.

Listing 1: conanfile.py

```
from conans import ConanFile, CMake, tools

class LibConan(ConanFile):
    ...

    def source(self):
        self.run("git clone https://github.com/conan-io/hello.git")

    def build(self):
        cmake = CMake(self)
        cmake.configure(source_folder="hello")
        cmake.build()
```

First we will call **conan source** to get our source code in the *src* directory, then **conan install** to install the requirements and generate the info files, and finally **conan build** to build the package:

```
$ conan source . --source-folder src
$ conan install . --install-folder build_x86 -s arch=x86
$ conan build . --build-folder build_x86 --source-folder src
```

Or if we want to create the `conaninfo.txt` and `conanbuildinfo.txt` files in a different folder:

```
$ conan source . --source-folder src
$ conan install . --install-folder install_x86 -s arch=x86
$ conan build . --build-folder build_x86 --install-folder install_x86 --source-folder␣
↪src
```

However, we recommend the `conaninfo.txt` and `conanbuildinfo.txt` to be generated in the same –build-folder, otherwise, you will need to specify a different folder in your build system to include the files generators file. E.g., `conanbuildinfo.cmake`

Example: Control the build stages

You can control the build stages using **--configure/--build/--install/--test** arguments. Here is an example using the CMake build helper:

```
$ conan build . --configure # only run cmake.configure(). Other methods will do nothing
$ conan build . --build     # only run cmake.build(). Other methods will do nothing
$ conan build . --install   # only run cmake.install(). Other methods will do nothing
$ conan build . --test      # only run cmake.test(). Other methods will do nothing
# They can be combined
$ conan build . -c -b # run cmake.configure() + cmake.build(), but not cmake.install()␣
↪nor cmake.test
```

If nothing is specified, all the methods will be called.

See also:

Read more about *should_configure, should_build, should_install, should_test*.

conan package

```
$ conan package [-h] [-bf BUILD_FOLDER] [-if INSTALL_FOLDER]
                [-pf PACKAGE_FOLDER] [-sf SOURCE_FOLDER]
                path
```

Calls your local conanfile.py 'package()' method.

This command works in the user space and it will copy artifacts from the –build-folder and –source-folder folder to the –package-folder one. It won't create a new package in the local cache, if you want to do it, use 'conan create' or 'conan export-pkg' after a 'conan build' command.

```
positional arguments:
  path                  Path to a folder containing a conanfile.py or to a
                        recipe file e.g., my_folder/conanfile.py

optional arguments:
  -h, --help            show this help message and exit
  -bf BUILD_FOLDER, --build-folder BUILD_FOLDER
                        Directory for the build process. Defaulted to the
                        current directory. A relative path to the current
                        directory can also be specified
  -if INSTALL_FOLDER, --install-folder INSTALL_FOLDER
                        Directory containing the conaninfo.txt and
                        conanbuildinfo.txt files (from previous 'conan
                        install'). Defaulted to --build-folder
  -pf PACKAGE_FOLDER, --package-folder PACKAGE_FOLDER
                        folder to install the package. Defaulted to the
                        '{build_folder}/package' folder. A relative path can
                        be specified (relative to the current directory). Also
                        an absolute path is allowed.
  -sf SOURCE_FOLDER, --source-folder SOURCE_FOLDER
                        Directory containing the sources. Defaulted to the
                        conanfile's directory. A relative path to the current
                        directory can also be specified
```

The `package()` method might use *settings*, *options* and *environment variables* from the specified profile and dependencies information from the declared `deps_XXX_info` objects in the conanfile requirements.

All that information is saved automatically in the *conaninfo.txt* and *conanbuildinfo.txt* files respectively, when you run **`conan install`**. Those files have to be located in the specified **`--build-folder`**.

```
$ conan install . --build-folder=build
```

Examples

This example shows how `package()` works in a package which can be edited and built in user folders instead of the local cache.

```
$ conan new hello/0.1 -s
$ conan install . --install-folder=build_x86 -s arch=x86
```

(continues on next page)

(continued from previous page)

```
$ conan build . --build-folder=build_x86
$ conan package . --build-folder=build_x86 --package-folder=package_x86
$ ls package/x86
> conaninfo.txt  conanmanifest.txt  include/  lib/
```

Note: The packages created locally are just for the user, but cannot be directly consumed by other packages, nor they can be uploaded to a remote repository. In order to make these packages available to the system, they have to be put in the conan local cache, which can be done with the **conan export-pkg** command instead of using **conan package** command:

```
$ conan new hello/0.1 -s
$ conan install . --install-folder=build_x86 -s arch=x86
$ conan build . --build-folder=build_x86
$ conan export-pkg . hello/0.1@user/stable --build-folder=build_x86 -s arch=x86
```

conan editable

```
$ conan editable [-h] {add,remove,list} ...
```

Manages editable packages (packages that reside in the user workspace, but are consumed as if they were in the cache).

Use the subcommands 'add', 'remove' and 'list' to create, remove or list packages currently installed in this mode.

```
positional arguments:
  {add,remove,list}  sub-command help
    add              Put a package in editable mode
    remove           Disable editable mode for a package
    list             List packages in editable mode

optional arguments:
  -h, --help         show this help message and exit
```

conan editable add

```
$ conan editable add [-h] [-l LAYOUT] path reference
```

Opens the package <reference> in editable mode in the user folder <path>

```
positional arguments:
path                  Path to the package folder in the user workspace
reference             Package reference e.g.: mylib/1.X@user/channel

optional arguments:
-h, --help            show this help message and exit
-l LAYOUT, --layout LAYOUT
                        Relative or absolute path to a file containing the
                        layout. Relative paths will be resolved first relative
                        to current dir, then to local cache "layouts" folder
```

(continues on next page)

(continued from previous page)

```
-of OUTPUT_FOLDER, --output-folder OUTPUT_FOLDER
                    The root output folder for generated and build files
```

This command puts a package in *"Editable mode"*, and consumers of this package will use it from the given user folder instead of using it from the cache. The path pointed by `path` should exist and contain a `conanfile.py`.

Example: Put the package `cool/version@user/dev` in editable mode, using the layout specified by the file `win_layout`.

```
$ conan editable add . cool/version@user/dev --layout=win_layout
```

The `--output-folder` will specify the location of the output (build files, generators, etc), with the same meaning as the `conan install` command. By default, if not specified, they will point to the folder containing the `conanfile.py`. These arguments work together with the `layout()` method to determine the location of the different Conan and build files.

conan editable remove

```
$ conan editable remove [-h] reference
```

Removes the editable mode of package `reference`.

```
positional arguments:
reference   Package reference e.g.: mylib/1.X@user/channel

optional arguments:
-h, --help  show this help message and exit
```

Example: remove the "Editable mode", use again package from the cache:

```
$ conan editable remove cool/version@user/dev
```

conan editable list

```
$ conan editable list [-h]
```

Shows the list of the packages that are opened in "editable" mode.

conan workspace

```
$ conan workspace [-h] {install} ...
```

Manages a workspace (a set of packages consumed from the user workspace that belongs to the same project).

Use this command to manage a Conan workspace, use the subcommand 'install' to create the workspace from a file.

```
positional arguments:
  {install}   sub-command help
    install   same as a "conan install" command but using the workspace data
              from the file. If no file is provided, it will look for a file
```

(continues on next page)

(continued from previous page)

```
              named "conanws.yml"

optional arguments:
  -h, --help  show this help message and exit
```

conan workspace install

```
$ conan workspace install [-h] [-b [BUILD]] [-r REMOTE] [-u] [-l [LOCKFILE]]
                          [-e ENV_HOST] [-e:b ENV_BUILD] [-e:h ENV_HOST]
                          [-o OPTIONS_HOST] [-o:b OPTIONS_BUILD] [-o:h OPTIONS_HOST]
                          [-pr PROFILE_HOST] [-pr:b PROFILE_BUILD]
                          [-pr:h PROFILE_HOST] [-s SETTINGS_HOST]
                          [-s:b SETTINGS_BUILD] [-s:h SETTINGS_HOST]
                          [-c CONF_HOST] [-c:b CONF_BUILD] [-c:h CONF_HOST]
                          [-if INSTALL_FOLDER]
                          path
```

```
positional arguments:
  path                  path to workspace definition file (it will look for a "conanws.
↪yml"
                        inside if a directory is given)

optional arguments:
  -h, --help            show this help message and exit
  -b [BUILD], --build [BUILD]
                        Optional, use it to choose if you want to build from sources:
                        --build Build all from sources, do not use binary packages.
                        --build=never Never build, use binary packages or fail if a␣
↪binary
                        package is not found. --build=missing Build from code if a binary
                        package is not found. --build=cascade Will build from code all␣
↪the
                        nodes with some dependency being built (for any reason). Can be
                        used together with any other build policy. Useful to make sure␣
↪that
                        any new change introduced in a dependency is incorporated by
                        building again the package. --build=outdated Build from code if␣
↪the
                        binary is not built with the current recipe or when missing a
                        binary package. --build=[pattern] Build always these packages␣
↪from
                        source, but never build the others. Allows multiple --build
                        parameters. 'pattern' is a fnmatch file pattern of a package
                        reference. Default behavior: If you don't specify anything, it␣
↪will
                        be similar to '--build=never', but package recipes can override␣
↪it
                        with their 'build_policy' attribute in the conanfile.py.
  -r REMOTE, --remote REMOTE
                        Look in the specified remote server
```

(continues on next page)

(continued from previous page)

```
  -u, --update          Will check the remote and in case a newer version
                        and/or revision of the dependencies exists there, it
                        will install those in the local cache. When using
                        version ranges, it will install the latest version
                        that satisfies the range. Also, if using revisions, it
                        will update to the latest revision for the resolved
                        version range.
  -l [LOCKFILE], --lockfile [LOCKFILE]
                        Path to a lockfile or folder containing 'conan.lock' file.␣
↪Lockfile
                        can be updated if packages change
  -e ENV_HOST, --env ENV_HOST
                        Environment variables that will be set during the package build
                        (host machine). e.g.: -e CXX=/usr/bin/clang++
  -e:b ENV_BUILD, --env:build ENV_BUILD
                        Environment variables that will be set during the package build
                        (build machine). e.g.: -e CXX=/usr/bin/clang++
  -e:h ENV_HOST, --env:host ENV_HOST
                        Environment variables that will be set during the package build
                        (host machine). e.g.: -e CXX=/usr/bin/clang++
  -o OPTIONS_HOST, --options OPTIONS_HOST
                        Define options values (host machine), e.g.: -o Pkg:with_qt=true
  -o:b OPTIONS_BUILD, --options:build OPTIONS_BUILD
                        Define options values (build machine), e.g.: -o Pkg:with_qt=true
  -o:h OPTIONS_HOST, --options:host OPTIONS_HOST
                        Define options values (host machine), e.g.: -o Pkg:with_qt=true
  -pr PROFILE_HOST, --profile PROFILE_HOST
                        Apply the specified profile to the host machine
  -pr:b PROFILE_BUILD, --profile:build PROFILE_BUILD
                        Apply the specified profile to the build machine
  -pr:h PROFILE_HOST, --profile:host PROFILE_HOST
                        Apply the specified profile to the host machine
  -s SETTINGS_HOST, --settings SETTINGS_HOST
                        Settings to build the package, overwriting the defaults (host
                        machine). e.g.: -s compiler=gcc
  -s:b SETTINGS_BUILD, --settings:build SETTINGS_BUILD
                        Settings to build the package, overwriting the defaults (build
                        machine). e.g.: -s compiler=gcc
  -s:h SETTINGS_HOST, --settings:host SETTINGS_HOST
                        Settings to build the package, overwriting the defaults (host
                        machine). e.g.: -s compiler=gcc
  -c CONF_HOST, --conf CONF_HOST
                        Configuration to build the package, overwriting the defaults␣
↪(host machine). e.g.: -c
                        tools.cmake.cmaketoolchain:generator=Xcode
  -c:b CONF_BUILD, --conf:build CONF_BUILD
                        Configuration to build the package, overwriting the defaults␣
↪(build machine). e.g.: -c:b
                        tools.cmake.cmaketoolchain:generator=Xcode
  -c:h CONF_HOST, --conf:host CONF_HOST
                        Configuration to build the package, overwriting the defaults␣
↪(host machine). e.g.: -c:h
```

(continues on next page)

(continued from previous page)

```
                        tools.cmake.cmaketoolchain:generator=Xcode
  -if INSTALL_FOLDER, --install-folder INSTALL_FOLDER
                        Folder where the workspace files will be created (default to
                        current working directory)
```

Note that these arguments, like `settings` and `options` mostly apply to the dependencies, but those packages that are defined as editable in the workspace are in the user space. Those packages won't be built by the command (even with `--build` arguments), as they are built locally. It is the responsibility of the editables layout to match the settings (typically parameterizing the layout with `settings` and `options`)

18.1.4 Misc commands

Other useful commands:

conan profile

```
$ conan profile [-h] {list,show,new,update,get,remove} ...
```

Lists profiles in the '.conan/profiles' folder, or shows profile details.

The 'list' subcommand will always use the default user 'conan/profiles' folder. But the 'show' subcommand can resolve absolute and relative paths, as well as to map names to '.conan/profiles' folder, in the same way as the '–profile' install argument.

```
positional arguments:
  {list,show,new,update,get,remove}
                        sub-command help
    list                List current profiles
    show                Show the values defined for a profile
    new                 Creates a new empty profile
    update              Update a profile with desired value
    get                 Get a profile key
    remove              Remove a profile key

optional arguments:
  -h, --help            show this help message and exit
```

Examples

- List the profiles:

```
$ conan profile list
> myprofile1
> myprofile2
```

- Print profile contents:

```
$ conan profile show myprofile1
Profile myprofile1
[settings]
...
```

- Print profile contents (in the standard directory `.conan/profiles`):

```
$ conan profile show myprofile1
Profile myprofile1
[settings]
...
```

- Print profile contents (in a custom directory):

```
$ conan profile show /path/to/myprofile1
Profile myprofile1
[settings]
...
```

- Update a setting from a profile located in a custom directory:

```
$ conan profile update settings.build_type=Debug /path/to/my/profile
```

- Add a new option to the default profile:

```
$ conan profile update options.zlib:shared=True default
```

- Create a new empty profile:

```
$ conan profile new /path/to/new/profile
```

- Create a new profile detecting the settings:

```
$ conan profile new /path/to/new/profile --detect
```

- Create a new or overwrite an existing profile with detected settings:

```
$ conan profile new /path/to/new/profile --detect --force
```

conan remote

```
$ conan remote [-h]
               {list,add,remove,update,rename,list_ref,add_ref,remove_ref,update_ref,
↪list_pref,add_pref,remove_pref,update_pref,clean,enable,disable}
               ...
```

Manages the remote list and the package recipes associated with a remote.

```
positional arguments:
  {list,add,remove,update,rename,list_ref,add_ref,remove_ref,update_ref,list_pref,add_
↪pref,remove_pref,update_pref,clean,enable,disable}
                        sub-command help
    list                List current remotes
    add                 Add a remote
    remove              Remove a remote
    update              Update the remote url
    rename              Update the remote name
    list_ref            List the package recipes and its associated remotes
    add_ref             Associate a recipe's reference to a remote
    remove_ref          Dissociate a recipe's reference and its remote
```

(continues on next page)

(continued from previous page)

```
    update_ref          Update the remote associated with a package recipe
    list_pref           List the package binaries and its associated remotes
    add_pref            Associate a package reference to a remote
    remove_pref         Dissociate a package's reference and its remote
    update_pref         Update the remote associated with a binary package
    clean               Clean the list of remotes and all recipe-remote
                        associations
    enable              Enable a remote
    disable             Disable a remote

optional arguments:
  -h, --help            show this help message and exit
```

Examples

- List remotes:

```
$ conan remote list
conancenter: https://center.conan.io [Verify SSL: True]
local: http://localhost:9300 [Verify SSL: True, Disabled: True]
```

- List remotes in a format almost valid for the *remotes.txt* to use with *conan config install*, only need to remove the `True` boolean appended to disabled remotes (notice line for `local` one in the output):

```
$ conan remote list --raw
conancenter https://center.conan.io True
local http://localhost:9300 True True
# capture the current remotes in a text file
$ conan remote list --raw > remotes.txt
```

- Add a new remote:

```
$ conan remote add remote_name remote_url [verify_ssl]
```

  Verify SSL option can be True or False (default True). Conan client will verify the SSL certificates.

- Insert a new remote:

  Insert as the first one (position/index 0), so it is the first one to be checked:

```
$ conan remote add remote_name remote_url [verify_ssl] --insert
```

  Insert as the second one (position/index 1), so it is the second one to be checked:

```
$ conan remote add remote_name remote_url [verify_ssl] --insert=1
```

- Add or insert a remote:

Adding the `--force` argument to `conan remote add` will always work, and won't raise an error. If an existing remote exists with that remote name or URL, it will be updated with the new information. The `--insert` works the same. If not specified, the remote will be appended the last one. If specified, the command will insert the remote in the specified position

```
$ conan remote add remote_name remote_url [verify_ssl] --force --insert=1
```

- Remove a remote:

```
$ conan remote remove remote_name
```

- Remove all configured remotes (this will also remove all recipe-remote associations):

```
$ conan remote clean
```

- Update a remote:

```
$ conan remote update remote_name new_url [verify_ssl]
```

- Rename a remote:

```
$ conan remote rename remote_name new_remote_name
```

- Change an existing remote to the first position:

```
$ conan remote update remote_name same_url --insert 0
```

- List the package recipes and its associated remotes:

```
$ conan remote list_ref
bzip2/1.0.6@lasote/stable: conan.io
Boost/1.60.0@lasote/stable: conan.io
zlib/1.2.8@lasote/stable: conan.io
```

- In some cases you may want to list the packages that are not associated with any remote:

```
$ conan remote list_ref --no-remote
spdlog/1.8.0: None
restinio/0.6.10: None
opencv/2.4.13.7: None
```

- Also, you can list the remote association for different binaries of the same Conan package:

```
$ conan remote list_pref zlib/1.2.8@
zlib/1.2.8:f83037eff23ab3a94190d7f3f7b37a2d6d522241: conancenter
zlib/1.2.8:e46341e9b52d3e4c66657dc8fb13ab6cdd5831c6: conan-local-dev
zlib/1.2.8:9de3196f2439d69299f168e3088bbefafe212f38: conan-local-prod
```

- If you want to know if any binaries of a Conan package have no remote association:

```
$ conan remote list_pref zlib/1.2.8@  --no-remote
zlib/1.2.8:a7480322bf53ca215dbba4db77ee500c7c51ee33: None
```

- Associate a recipe's reference to a remote:

```
$ conan remote add_ref openssl/1.0.2u conancenter
```

- Update the remote associated with a package recipe:

```
$ conan remote update_ref openssl/1.0.2t local-remote
```

- Enable or disable remotes (accepts patterns such as * as argument using Unix shell-style wildcards):

```
$ conan remote disable "*"
$ conan remote enable local-remote
```

Note: Check the section *How to manage SSL (TLS) certificates* section to know more about server certificates verification and client certifications management .

conan user

```
$ conan user [-h] [-c] [-p [PASSWORD]] [-r REMOTE] [-j JSON] [-s] [name]
```

Authenticates against a remote with user/pass, caching the auth token.

Useful to avoid the user and password being requested later. e.g. while you're uploading a package. You can have one user for each remote. Changing the user, or introducing the password is only necessary to perform changes in remote packages.

```
positional arguments:
  name                  Username you want to use. If no name is provided it
                        will show the current user

optional arguments:
  -h, --help            show this help message and exit
  -c, --clean           Remove user and tokens for all remotes
  -p [PASSWORD], --password [PASSWORD]
                        User password. Use double quotes if password with
                        spacing, and escape quotes if existing. If empty, the
                        password is requested interactively (not exposed)
  -r REMOTE, --remote REMOTE
                        Use the specified remote server
  -j JSON, --json JSON  json file path where the user list will be written to
  -s, --skip-auth       Skips the authentication with the server if there are
                        local stored credentials. It doesn't check if the
                        current credentials are valid or not
```

After a successful login the auth token is stored in the local database (see *CONAN_LOGIN_ENCRYPTION_KEY* to add a basic level of security).

Examples:

- List my user for each remote:

```
$ conan user
Current user of remote 'conancenter' set to: 'None' (anonymous)
Current user of remote 'myprivateremote' set to: 'danimtb' [Authenticated]
Current user of remote 'otherremote' set to: 'None' (anonymous)
```

- Change **bar** remote user to **foo**:

```
$ conan user foo -r bar
Changed user of remote 'bar' from 'None' (anonymous) to 'foo'
```

- Change **bar** remote user to **foo**, authenticating against the remote and storing the user and authentication token locally, so a later upload won't require entering credentials:

```
$ conan user foo -r bar -p mypassword
```

- Authenticate against the remote only if we don't have credentials stored locally. It will not check if the credentials are valid or not:

```
$ conan user foo -r bar -p mypassword --skip-auth
```

- Clean all local users and tokens:

```
$ conan user --clean
```

- Change **bar** remote user to **foo**, **asking user password** to authenticate against the remote and storing the user and authentication token locally, so a later upload won't require entering credentials:

```
$ conan user foo -r bar -p
Please enter a password for "foo" account:
Change 'bar' user from None (anonymous) to foo
```

Note: The password is not stored in the client computer at any moment. Conan uses JWT, so it gets a token (expirable by the server) checking the password against the remote credentials. If the password is correct, an authentication token will be obtained, and that token is the information cached locally. For any subsequent interaction with the remotes, the Conan client will only use that JWT token.

Using environment variables

The *CONAN_LOGIN_USERNAME* and *CONAN_PASSWORD* environment variables allow defining the user and the password in the environment. If those environment variables are defined, the user input will no be necessary whenever the user or password are requested. Values for user and password will be automatically taken from the environment variables without any interactive input.

This applies also to the `conan user` command, if you want to force the authentication in some scripts, without requiring to put the password in plain text, the following can be done:

```
$ conan user --clean  # remove previous auth tokens
$ export CONAN_PASSWORD=mypassword
$ conan user mysyusername -p -r=myremote
Please enter a password for "mysusername" account: Got password '******' from environment
Changed user of remote 'myremote' from 'None' (anonymous) to 'mysusername'
$ conan upload zlib* -r=myremote --all --confirm
```

In this example, **`conan user mysyusername -p -r=myremote`** will interactively request a password if `CONAN_PASSWORD` is not defined.

The environment variable *CONAN_NON_INTERACTIVE* (or `general.non_interactive` in *conan.conf*) can be defined to guarantee that an error will be raise if user input is required, to avoid stalls in CI builds.

Note that defining `CONAN_LOGIN_USERNAME` and/or `CONAN_PASSWORD` do not perform in any case an authentication request against the server. Only when the server request credentials (or a explicit **`conan user -p`** is done), they will be used as an alternative source rather than interactive user input. This means that for servers like Artifactory that allow enabling *"Hide Existence of Unauthorized Resource"* modes, it will be necessary to explicitly call **`conan user -p`** before downloading or uploading anything from the server, otherwise, Artifactory will return 404 errors instead of requesting authentication.

conan imports

```
$ conan imports [-h] [-if INSTALL_FOLDER] [-imf IMPORT_FOLDER] [-u] path
```

Calls your local conanfile.py or conanfile.txt 'imports' method.

It requires to have been previously installed and have a conanbuildinfo.txt generated file in the –install-folder (defaulted to the current directory).

```
positional arguments:
  path                  Path to a folder containing a conanfile.py or to a
                        recipe file e.g., my_folder/conanfile.py With --undo
                        option, this parameter is the folder containing the
                        conan_imports_manifest.txt file generated in a
                        previous execution. e.g.: conan imports
                        ./imported_files --undo

optional arguments:
  -h, --help            show this help message and exit
  -if INSTALL_FOLDER, --install-folder INSTALL_FOLDER
                        Directory containing the conaninfo.txt and
                        conanbuildinfo.txt files (from previous 'conan
                        install'). Defaulted to --build-folder
  -imf IMPORT_FOLDER, --import-folder IMPORT_FOLDER
                        Directory to copy the artifacts to. By default it will
                        be the current directory
  -u, --undo            Undo imports. Remove imported files
```

The `imports()` method might use *settings*, *options* and *environment variables* from the specified profile and dependencies information from the declared `deps_XXX_info` objects in the conanfile requirements.

All that information is saved automatically in the *conaninfo.txt* and *conanbuildinfo.txt* files respectively, when you run **`conan install`**. Those files have to be located in the specified **`--install-folder`**.

Examples

- Import files from a current conanfile in current directory:

```
$ conan install . --no-imports # Creates the conanbuildinfo.txt
$ conan imports .
```

- Remove the copied files (undo the import):

```
$ conan imports . --undo
```

conan copy

```
$ conan copy [-h] [-p PACKAGE] [--all] [--force] reference user_channel
```

Copies conan recipes and packages to another user/channel.

Useful to promote packages (e.g. from "beta" to "stable") or transfer them from one user to another.

```
positional arguments:
  reference             package reference. e.g., MyPackage/1.2@user/channel
  user_channel          Destination user/channel. e.g., lasote/testing

optional arguments:
  -h, --help            show this help message and exit
  -p PACKAGE, --package PACKAGE
                        copy specified package ID [DEPRECATED: use full
                        reference instead]
  --all                 Copy all packages from the specified package recipe
  --force               Override destination packages and the package recipe
```

Examples

- Promote a package to **stable** from **beta**:

```
$ conan copy mypackage/1.0.0@lasote/beta lasote/stable
```

- Change a package's username:

```
$ conan copy openssl/1.0.2u@ foo/beta
```

conan download

```
$ conan download [-h] [-p PACKAGE] [-r REMOTE] [-re] reference
```

Downloads recipe and binaries to the local cache, without using settings.

It works specifying the recipe reference and package ID to be installed. Not transitive, requirements of the specified reference will NOT be retrieved. Useful together with 'conan copy' to automate the promotion of packages to a different user/channel. Only if a reference is specified, it will download all packages from the specified remote. If no remote is specified, it will use the default remote.

```
positional arguments:
  reference             pkg/version@user/channel

optional arguments:
  -h, --help            show this help message and exit
  -p PACKAGE, --package PACKAGE
                        Force install specified package ID (ignore
                        settings/options) [DEPRECATED: use full reference
                        instead]
  -r REMOTE, --remote REMOTE
                        look in the specified remote server
  -re, --recipe         Downloads only the recipe
```

Examples

- Download all **openssl/1.0.2u** binary packages from the remote **foo**:

```
$ conan download openssl/1.0.2u@ -r foo
```

- Download a single binary package of **openssl/1.0.2u** from the remote **foo**:

```
$ conan download openssl/1.0.2u@:8018a4df6e7d2b4630a814fa40c81b85b9182d2 -r foo
```

- Download only the recipe of package **openssl/1.0.2u** from the remote **foo**:

```
$ conan download openssl/1.0.2u@ -r foo -re
```

conan remove

```
$ conan remove [-h] [-b [BUILDS [BUILDS ...]]] [-f] [-l] [-o]
               [-p [PACKAGES [PACKAGES ...]]] [-q QUERY] [-r REMOTE] [-s]
               [-t]
               [pattern_or_reference]
```

Removes packages or binaries matching pattern from local cache or remote.

It can also be used to remove the temporary source or build folders in the local conan cache. If no remote is specified, the removal will be done by default in the local conan cache.

```
positional arguments:
  pattern_or_reference  Pattern or package recipe reference, e.g., 'boost/*',
                        'MyPackage/1.2@user/channel'

optional arguments:
  -h, --help            show this help message and exit
  -b [BUILDS [BUILDS ...]], --builds [BUILDS [BUILDS ...]]
                        By default, remove all the build folders or select
                        one, specifying the package ID
  -f, --force           Remove without requesting a confirmation
  -l, --locks           Remove locks
  -o, --outdated        Remove only outdated from recipe packages. This flag
                        can only be used with a pattern or a reference
  -p [PACKAGES [PACKAGES ...]], --packages [PACKAGES [PACKAGES ...]]
                        Remove all packages of the specified reference if no
                        specific package ID is provided
  -q QUERY, --query QUERY
                        Packages query: 'os=Windows AND (arch=x86 OR
                        compiler=gcc)'. The 'pattern_or_reference' parameter
                        has to be a reference: MyPackage/1.2@user/channel
  -r REMOTE, --remote REMOTE
                        Will remove from the specified remote
  -s, --src             Remove source folders
  -t, --system-reqs     Remove system_reqs folders
```

The `-q` parameter can't be used along with `-p` nor `-b` parameters.

Examples:

- Remove from the local cache the binary packages (the package recipes will not be removed) from all the recipes matching `openssl/*` pattern:

```
$ conan remove openssl/* --packages
```

- Remove the temporary build folders from all the recipes matching `openssl/*` pattern without requesting confirmation:

```
$ conan remove openssl/* --builds --force
```

- Remove the recipe and the binary packages from a specific remote:

```
$ conan remove openssl/1.0.u@ -r myremote
```

- Remove only Windows openssl packages from local cache:

```
$ conan remove openssl/1.0.u@ -q "os=Windows"
```

- Remove system requirements installation registry for the package name referred globally for all package ids:

```
$ conan remove --system-reqs package/version@user/channel
```

  This command does not remove the system installed packages, but only the Conan lock to indicate they were installed.

- Remove system requirements installation registry for all packages named `package` via a wildcard

```
$ conan remove --system-reqs 'package/*'
```

- Remove system requirements installation registry for all packages via a wildcard

```
$ conan remove --system-reqs '*'
```

- Remove all remote packages only related to a specific recipe revision

```
$ conan remove -r myremote package/version@user/channel#RREV --packages
```

- Remove only a single remote package related to a specific recipe revision and its package ID

```
$ conan remove -r myremote package/version@user/channel#RREV -p package_id

OR

$ conan remove -r myremote package/version@user/channel#RREV:PACKAGE_ID
```

conan alias

```
$ conan alias [-h] reference target
```

Creates and exports an 'alias package recipe'.

An "alias" package is a symbolic name (reference) for another package (target). When some package depends on an alias, the target one will be retrieved and used instead, so the alias reference, the symbolic name, does not appear in the final dependency graph.

```
positional arguments:
  reference   Alias reference. e.g.: mylib/1.X@user/channel
  target      Target reference. e.g.: mylib/1.12@user/channel

optional arguments:
  -h, --help  show this help message and exit
```

The command:

```
$ conan alias hello/0.X@user/testing hello/0.1@user/testing
```

Creates and exports a package recipe for `hello/0.X@user/testing` with the following content:

```
from conans import ConanFile

class AliasConanfile(ConanFile):
    alias = "hello/0.1@user/testing"
```

Such package recipe acts as a "proxy" for the aliased reference. Users depending on `hello/0.X@user/testing` will actually use version `hello/0.1@user/testing`. The alias package reference will not appear in the dependency graph at all. It is useful to define symbolic names, or behaviors like "always depend on the latest minor", but defined upstream instead of being defined downstream with `version-ranges`.

The "alias" package should be uploaded to servers in the same way as regular package recipes, in order to enable usage from servers.

From Conan 1.39, a new **experimental** explicit syntax for requiring alias packages has been introduced, designed to supersede the current one in Conan 2.0, in the form ``requires = "pkg/(latest)@user/testing"``. Read more about it in *this section*.

conan inspect

Warning: This is an **experimental** feature subject to breaking changes in future releases.

```
$ conan inspect [-h] [-a [ATTRIBUTE]] [-r REMOTE] [-j JSON] [--raw RAW]
                path_or_reference
```

Displays conanfile attributes, like name, version, and options. Works locally, in local cache and remote.

```
positional arguments:
  path_or_reference     Path to a folder containing a recipe (conanfile.py) or
                        to a recipe file. e.g., ./my_project/conanfile.py. It
                        could also be a reference

optional arguments:
  -h, --help            show this help message and exit
  -a [ATTRIBUTE], --attribute [ATTRIBUTE]
                        The attribute to be displayed, e.g "name"
  -r REMOTE, --remote REMOTE
                        look in the specified remote server
  -j JSON, --json JSON  json output file
  --raw RAW             Print just the value of the requested attribute
```

Examples:

```
$ conan inspect zlib/1.2.11@ -a=name -a=version -a=options -a default_options -
↪r=conancenter
name: zlib
version: 1.2.11
options
```

(continues on next page)

(continued from previous page)

```
    shared: [True, False]
default_options: shared=False
```

```
$ conan inspect zlib/1.2.11@ -a=license -a=url
license: Zlib
url: https://github.com/conan-io/conan-center-index
```

```
$ conan inspect zlib/1.2.11@ --raw=settings
('os', 'arch', 'compiler', 'build_type')
```

```
$ conan inspect pkg/latest@ -a alias
...
alias: pkg/0.1

$ conan inspect pkg/latest@ -a alias --json=myinspect.json
$ cat myinspect.json
{"alias": "pkg/0.1"}
```

If no specific attributes are defined via `-a`, then, some default attributes will be displayed:

```
$ conan inspect zlib/1.2.11@
name: zlib
version: 1.2.11
url: https://github.com/conan-io/conan-center-index
homepage: https://zlib.net
license: Zlib
author: None
description: A Massively Spiffy Yet Delicately Unobtrusive Compression Library (Also␣
↪Free, Not to Mention Unencumbered by Patents)
topics: None
generators: cmake
exports: None
exports_sources: ['CMakeLists.txt', 'CMakeLists_minizip.txt', 'minizip.patch']
short_paths: False
apply_env: True
build_policy: None
revision_mode: hash
settings: ('os', 'arch', 'compiler', 'build_type')
options:
    fPIC: [True, False]
    minizip: [True, False]
    shared: [True, False]
default_options:
    fPIC: True
    minizip: False
    shared: False
```

conan lock

```
$ conan lock [-h]
             {update,build-order,clean-modified,install,create,bundle}
             ...
```

Generates and manipulates lock files.

```
positional arguments:
  {update,build-order,clean-modified,install,create,bundle}
                        sub-command help
    update              Complete missing information in the first lockfile
                        with information defined in the second lockfile. Both
                        lockfiles must represent the same graph, and have the
                        same topology with the same identifiers, i.e. the
                        second lockfile must be an evolution based on the
                        first one
    build-order         Returns build-order
    clean-modified      Clean modified flags
    install             Install a lockfile
    create              Create a lockfile from a conanfile or a reference
    bundle              Manages lockfile bundles

optional arguments:
  -h, --help            show this help message and exit
```

See also:

read about lockfiles in *Lockfiles*

conan lock create

```
$ conan lock create [-h] [--name NAME] [--version VERSION] [--user USER] [--channel␣
↪CHANNEL] [--reference REFERENCE] [-l LOCKFILE] [--base]
                    [--lockfile-out LOCKFILE_OUT] [-b [BUILD]] [-r REMOTE] [-u] [-e ENV_
↪HOST] [-e:b ENV_BUILD] [-e:h ENV_HOST] [-o OPTIONS_HOST] [-o:b OPTIONS_BUILD]
                    [-o:h OPTIONS_HOST] [-pr PROFILE_HOST] [-pr:b PROFILE_BUILD] [-pr:h␣
↪PROFILE_HOST] [-s SETTINGS_HOST] [-s:b SETTINGS_BUILD] [-s:h SETTINGS_HOST]
                    [-c CONF_HOST] [-c:b CONF_BUILD] [-c:h CONF_HOST]
                    [path]
```

```
positional arguments:
  path                  Path to a conanfile

optional arguments:
  -h, --help            show this help message and exit
  --name NAME           Provide a package name if not specified in conanfile
  --version VERSION     Provide a package version if not specified in conanfile
  --user USER           Provide a user
  --channel CHANNEL     Provide a channel
  --reference REFERENCE
                        Provide a package reference instead of a conanfile
```

(continues on next page)

(continued from previous page)

```
  -l LOCKFILE, --lockfile LOCKFILE
                        Path to lockfile to be used as a base
  --base                Lock only recipe versions and revisions
  --lockfile-out LOCKFILE_OUT
                        Filename of the created lockfile
  -b [BUILD], --build [BUILD]
                        Packages to build from source
  -r REMOTE, --remote REMOTE
                        Look in the specified remote server
  -u, --update          Will check the remote and in case a newer version and/or␣
↪revision of the dependencies exists there, it will install those in the local cache.␣
↪When
                        using version ranges, it will install the latest version that␣
↪satisfies the range. Also, if using revisions, it will update to the latest revision
                        for the resolved version range.
  -e ENV_HOST, --env ENV_HOST
                        Environment variables that will be set during the package build␣
↪(host machine). e.g.: -e CXX=/usr/bin/clang++
  -e:b ENV_BUILD, --env:build ENV_BUILD
                        Environment variables that will be set during the package build␣
↪(build machine). e.g.: -e:b CXX=/usr/bin/clang++
  -e:h ENV_HOST, --env:host ENV_HOST
                        Environment variables that will be set during the package build␣
↪(host machine). e.g.: -e:h CXX=/usr/bin/clang++
  -o OPTIONS_HOST, --options OPTIONS_HOST
                        Define options values (host machine), e.g.: -o Pkg:with_qt=true
  -o:b OPTIONS_BUILD, --options:build OPTIONS_BUILD
                        Define options values (build machine), e.g.: -o:b Pkg:with_
↪qt=true
  -o:h OPTIONS_HOST, --options:host OPTIONS_HOST
                        Define options values (host machine), e.g.: -o:h Pkg:with_qt=true
  -pr PROFILE_HOST, --profile PROFILE_HOST
                        Apply the specified profile to the host machine
  -pr:b PROFILE_BUILD, --profile:build PROFILE_BUILD
                        Apply the specified profile to the build machine
  -pr:h PROFILE_HOST, --profile:host PROFILE_HOST
                        Apply the specified profile to the host machine
  -s SETTINGS_HOST, --settings SETTINGS_HOST
                        Settings to build the package, overwriting the defaults (host␣
↪machine). e.g.: -s compiler=gcc
  -s:b SETTINGS_BUILD, --settings:build SETTINGS_BUILD
                        Settings to build the package, overwriting the defaults (build␣
↪machine). e.g.: -s:b compiler=gcc
  -s:h SETTINGS_HOST, --settings:host SETTINGS_HOST
                        Settings to build the package, overwriting the defaults (host␣
↪machine). e.g.: -s:h compiler=gcc
  -c CONF_HOST, --conf CONF_HOST
                        Configuration to build the package, overwriting the defaults␣
↪(host machine). e.g.: -c
                        tools.cmake.cmaketoolchain:generator=Xcode
  -c:b CONF_BUILD, --conf:build CONF_BUILD
                        Configuration to build the package, overwriting the defaults␣
```

(continues on next page)

(continued from previous page)

```
↪(build machine). e.g.: -c:b
                        tools.cmake.cmaketoolchain:generator=Xcode
  -c:h CONF_HOST, --conf:host CONF_HOST
                        Configuration to build the package, overwriting the defaults␣
↪(host machine). e.g.: -c:h
                        tools.cmake.cmaketoolchain:generator=Xcode
```

conan lock update

```
$ conan lock update [-h] old_lockfile new_lockfile
```

```
positional arguments:
    old_lockfile  Path to lockfile to be updated
    new_lockfile  Path to lockfile containing the new information that is going to be␣
↪updated into the first lockfile

optional arguments:
    -h, --help    show this help message and exit
```

conan lock build-order

```
$ conan lock build-order [-h] [--json JSON] lockfile
```

```
positional arguments:
    lockfile     lockfile file

optional arguments:
    -h, --help   show this help message and exit
    --json JSON  generate output file in json format
```

conan lock clean-modified

```
$ conan lock clean-modified [-h] lockfile
```

```
positional arguments:
    lockfile    Path to the lockfile

optional arguments:
    -h, --help  show this help message and exit
```

conan lock install

```
$ conan lock install [-h] [--recipes] [-g GENERATOR] lockfile
```

```
positional arguments:
lockfile              Path to the lockfile

optional arguments:
-h, --help            show this help message and exit
--recipes             Install only recipes, not binaries
-g GENERATOR, --generator GENERATOR
                        Generators to use
```

conan lock bundle

```
$ conan lock bundle [-h] {create,build-order,update,clean-modified} ...
```

```
positional arguments:
{create,build-order,update,clean-modified}
                        sub-command help
    create              Create lockfile bundle
    build-order         Returns build-order
    update              Complete missing information in the first lockfile with␣
↪information defined in the second lockfile.
                        Both lockfiles must represent the same graph, and have the same␣
↪topology with the same identifiers,
                        i.e. the second lockfile must be an evolution based on the first␣
↪one
    clean-modified      Clean modified flag
```

conan lock bundle create

```
$ conan lock bundle create [-h] [--bundle-out BUNDLE_OUT] lockfiles [lockfiles ...]
```

```
positional arguments:
lockfiles             Path to lockfiles

optional arguments:
-h, --help            show this help message and exit
--bundle-out BUNDLE_OUT
                        Filename of the created bundle
```

conan lock bundle build-order

```
$ conan lock bundle build-order [-h] [--json JSON] bundle
```

```
positional arguments:
bundle       Path to lockfile bundle

optional arguments:
-h, --help   show this help message and exit
--json JSON  generate output file in json format
```

conan lock bundle update

```
$ conan lock bundle update [-h] bundle
```

```
positional arguments:
bundle      Path to lockfile bundle

optional arguments:
-h, --help  show this help message and exit
```

conan lock bundle clean-modified

```
$ conan lock bundle clean-modified [-h] bundle
```

```
positional arguments:
bundle      Path to lockfile bundle

optional arguments:
-h, --help  show this help message and exit
```

conan help

```
$ conan help [-h] [command]
```

Shows help for a specific command.

```
positional arguments:
  command     command

optional arguments:
  -h, --help  show this help message and exit
```

This command is equivalent to the `--help` and `-h` arguments

Example:

```
$ conan help get
> usage: conan get [-h] [-p PACKAGE] [-r REMOTE] [-raw] reference [path]
> Gets a file or list a directory of a given reference or package.

# same as
$ conan get -h
```

Warning: This is an **experimental** feature subject to breaking changes in future releases.

Warning: Some problems regarding the use of BuildInfo with Conan packages have been reported. If the BuildInfo contains artifacts that have the same checksum as other artifacts, this may result in losing the path of the artifact in the BuildInfo in Artifactory and also fail in the promotion process.

We are currently working along with the Artifactory team to solve those problems. Until this issue gets fixed, we do not recommend using BuildInfo's for Conan.

conan_build_info v1

```
usage: conan_build_info [-h] [--output OUTPUT] trace_path

Extracts build-info from a specified conan trace log and return a valid JSON

positional arguments:
  trace_path       Path to the conan trace log file e.g.: /tmp/conan_trace.log

optional arguments:
  -h, --help       show this help message and exit
  --output OUTPUT  Optional file to output the JSON contents, if not specified
                   the JSON will be printed to stdout
```

conan_build_info v2

```
$ conan_build_info --v2 [-h] {start,stop,create,update,publish} ...
```

```
Generates build-info from lockfiles information

positional arguments:
  {start,stop,create,update,publish}
                        sub-command help
    start               Command to incorporate to the artifacts.properties the
                        build name and number
    stop                Command to remove from the artifacts.properties the
                        build name and number
    create              Command to generate a build info json from a lockfile
    update              Command to update a build info json with another one
    publish             Command to publish the build info to Artifactory
```

(continues on next page)

(continued from previous page)

```
optional arguments:
  -h, --help            show this help message and exit
```

start subcommand:

```
usage: conan_build_info --v2 start [-h] build_name build_number

positional arguments:
  build_name    build name to assign
  build_number  build number to assign

optional arguments:
  -h, --help    show this help message and exit
```

stop subcommand:

```
usage: conan_build_info --v2 stop [-h]

optional arguments:
  -h, --help  show this help message and exit
```

create subcommand:

```
usage: conan_build_info --v2 create [-h] --lockfile LOCKFILE [--user [USER]]
                                    [--password [PASSWORD]] [--apikey [APIKEY]]
                                    build_info_file

positional arguments:
  build_info_file       build info json for output

optional arguments:
  -h, --help            show this help message and exit
  --lockfile LOCKFILE   input lockfile
  --user [USER]         user
  --password [PASSWORD]
                        password
  --apikey [APIKEY]     apikey
```

publish subcommand:

```
usage: conan_build_info --v2 publish [-h] --url URL [--user [USER]]
                                     [--password [PASSWORD]] [--apikey [APIKEY]]
                                     buildinfo

positional arguments:
  buildinfo             build info to upload

optional arguments:
  -h, --help            show this help message and exit
  --url URL             url
  --user [USER]         user
  --password [PASSWORD]
```

(continues on next page)

(continued from previous page)

```
                        password
  --apikey [APIKEY]     apikey
```

update subcommand:

```
usage: conan_build_info --v2 update [-h] [--output-file OUTPUT_FILE]
                                    buildinfo [buildinfo ...]

positional arguments:
  buildinfo             buildinfo files to merge

optional arguments:
  -h, --help            show this help message and exit
  --output-file OUTPUT_FILE
                        path to generated build info file
```

18.1.5 JSON Output

JSON documents generated by the commands:

Install and Create output

Warning: This is an **experimental** feature subject to breaking changes in future releases.

The **conan install** and **conan create** provide a `--json` parameter to generate a file containing the information of the installation process.

The output JSON contains a two first level keys:

- **error**: `True` if the install completed without error, `False` otherwise.
- **installed**: A list of installed packages. Each element contains:
  - **recipe**: Document representing the downloaded recipe.
    - **remote**: remote URL if the recipe has been downloaded. `null` otherwise.
    - **cache**: `true/false`. Retrieved from cache (not downloaded).
    - **downloaded**: `true/false`. Downloaded from a remote (not in cache).
    - **time**: `ISO 8601` string with the time the recipe was downloaded/retrieved.
    - **error**: `true/false`.
    - **id**: Reference. E.g., "openssl/1.0.2u"
    - **name**: name of the packaged library. E.g., "openssl"
    - **version**: version of the packaged library. E.g., "1.0.2u"
    - **user**: user of the packaged library. E.g., "conan"
    - **channel**: channel of the packaged library. E.g., "stable"
    - **dependency**: `true/false`. Is the package being installed/created or a dependency. Same as *develop conanfile attribute*.

- **packages**: List of elements, representing the binary packages downloaded for the recipe. Normally there will be only 1 element in this list, only in special cases with build requires, private dependencies and settings overridden this list could have more than one element.
  - **remote**: remote URL if the recipe has been downloaded. `null` otherwise.
  - **cache**: `true`/`false`. Retrieved from cache (not downloaded).
  - **downloaded**: `true`/`false`. Downloaded from a remote (not in cache).
  - **time**: ISO 8601 string with the time the recipe was downloaded/retrieved.
  - **error**: `true`/`false`.
  - **id**: Package ID. E.g., "8018a4df6e7d2b4630a814fa40c81b85b9182d2b"
  - **cpp_info**: dictionary containing the build information defined in the `package_info` method on the recipe.

Example:

```
$ conan install openssl/1.0.2u@ --json install.json
```

Listing 2: install.json


```
{
    "error": false,
    "installed": [{
        "recipe": {
            "id": "openssl/1.0.2u",
            "downloaded": true,
            "exported": false,
            "error": null,
            "remote": "https://center.conan.io",
            "time": "2020-01-30T19:19:21.217923",
            "dependency": true,
            "name": "openssl",
            "version": "1.0.2u",
            "user": null,
            "channel": null
        },
        "packages": [{
            "id": "f99afdbf2a1cc98ba2029817b35103455b6a9b77",
            "downloaded": true,
            "exported": false,
            "error": null,
            "remote": "https://center.conan.io",
            "time": "2020-01-30T19:19:27.662199",
            "built": false,
            "cpp_info": {
                "name": "openssl",
                "names": {
                    "cmake_find_package": "OpenSSL",
                    "cmake_find_package_multi": "OpenSSL"
                },
                "includedirs": ["include"],
                "libdirs": ["lib"],
```


(continues on next page)

(continued from previous page)


```
                "resdirs": ["res"],
                "bindirs": ["bin"],
                "builddirs": [""],
                "frameworkdirs": ["Frameworks"],
                "libs": ["ssl", "crypto", "dl", "pthread"],
                "rootpath": "/home/user/.conan/data/openssl/1.0.2u/_/_/package/
↪f99afdbf2a1cc98ba2029817b35103455b6a9b77",
                "version": "1.0.2u",
                "description": "A toolkit for the Transport Layer Security (TLS) and␣
↪Secure Sockets Layer (SSL) protocols",
                "filter_empty": true,
                "public_deps": ["zlib"]
            }
        }]
    }, {
        "recipe": {
            "id": "zlib/1.2.11#1cd4a227e1b846f961bf91fcb6f3980f",
            "downloaded": false,
            "exported": false,
            "error": null,
            "remote": null,
            "time": "2020-01-30T19:19:21.237131",
            "dependency": true,
            "name": "zlib",
            "version": "1.2.11",
            "user": null,
            "channel": null
        },
        "packages": [{
            "id": "6af9cc7cb931c5ad942174fd7838eb655717c709",
            "downloaded": false,
            "exported": false,
            "error": null,
            "remote": null,
            "time": "2020-01-30T19:19:22.061885",
            "built": false,
            "cpp_info": {
                "name": "ZLIB",
                "includedirs": ["include"],
                "libdirs": ["lib"],
                "resdirs": ["res"],
                "bindirs": ["bin"],
                "builddirs": [""],
                "frameworkdirs": ["Frameworks"],
                "libs": ["z"],
                "rootpath": "/home/user/.conan/data/zlib/1.2.11/_/_/package/
↪6af9cc7cb931c5ad942174fd7838eb655717c709",
                "version": "1.2.11",
                "description": "A Massively Spiffy Yet Delicately Unobtrusive␣
↪Compression Library (Also Free, Not to Mention Unencumbered by Patents)",
                "filter_empty": true
            }
```


(continues on next page)

(continued from previous page)

```
        }]
    }]
}
```

Note: As this is a marked as *experimental*, some fields may be removed or added: fields `version` and `description` inside `cpp_info` will eventually be removed and paths may be changed for absolute ones.

Search output

Warning: This is an **experimental** feature subject to breaking changes in future releases.

The **conan search** provides a **--json** parameter to generate a file containing the information of the search process.

The output JSON contains a two first level keys:

- **error**: `True` if the upload completed without error, `False` otherwise.
- **results**: A list of the remotes with the packages found. Each element contains:
  - **remote**: Name of the remote.
  - **items: List of the items found in that remote. For each item there will always be a** recipe and optionally also packages when searching them.
    - **recipe: Document representing the uploaded recipe.**
      - **id**: Reference, e.g., "openssl/1.0.2u"
    - **packages: List of elements representing the binary packages found for the recipe.**
      - **id**: Package ID, e.g., "8018a4df6e7d2b4630a814fa40c81b85b9182d2b"
      - **options**: Dictionary of options of the package.
      - **settings**: Dictionary with settings of the package.
      - **requires**: List of requires of the package.
      - **outdated**: Boolean to show whether package is outdated from recipe or not.

Examples:

- Search references in all remotes: **conan search eigen* -r all**

```
{
    "error": false,
    "results": [{
        "remote": "conancenter",
        "items": [{
            "recipe": {
                "id": "eigen/3.3.4@conan/stable"
            }
        }, {
            "recipe": {
                "id": "eigen/3.3.5@conan/stable"
```

(continues on next page)

(continued from previous page)

```
            }
        }, {
            "recipe": {
                "id": "eigen/3.3.7"
            }
        }, {
            "recipe": {
                "id": "eigen/3.3.7@conan/stable"
            }
        }]
    }, {
        "remote": "otherremote",
        "items": [{
            "recipe": {
                "id": "eigen/3.3.4@conan/stable"
            }
        }, {
            "recipe": {
                "id": "eigen/3.3.5@conan/stable"
            }
        }, {
            "recipe": {
                "id": "eigen/3.3.7@conan/stable"
            }
        }]
    }]
}
```

- Search packages of a reference in a remote: **conan search paho-c/1.2.0@conan/stable -r conancenter --json search.json**

```
{
    "error":false,
    "results":[
        {
            "remote":"conancenter",
            "items":[
                {
                    "recipe":{
                        "id":"paho-c/1.2.0@conan/stable"
                    },
                    "packages":[
                        {
                            "id":"0000193ac313953e78a4f8e82528100030ca70ee",
                            "options":{
                                "shared":"False",
                                "asynchronous":"False",
                                "SSL":"False"
                            },
                            "settings":{
                                "os":"Linux",
                                "arch":"x86_64",
```

(continues on next page)

(continued from previous page)


```
                            "compiler":"gcc",
                            "build_type":"Debug",
                            "compiler.version":"4.9"
                        },
                        "requires":[

                        ],
                        "outdated":false
                    },
                    {
                        "id":"014be746b283391f79d11e4e8af3154344b58223",
                        "options":{
                            "shared":"False",
                            "asynchronous":"False",
                            "SSL":"False"
                        },
                        "settings":{
                            "os":"Windows",
                            "compiler.threads":"posix",
                            "compiler.exception":"seh",
                            "arch":"x86_64",
                            "compiler":"gcc",
                            "build_type":"Debug",
                            "compiler.version":"5"
                        },
                        "requires":[

                        ],
                        "outdated":false
                    },
                    {
                      "id":"0188020dbfd167611b967ad2fa0e30710d23e920",
                        "options":{
                            "shared":"True",
                            "asynchronous":"False",
                            "SSL":"False"
                        },
                        "settings":{
                            "os":"Macos",
                            "arch":"x86_64",
                            "compiler":"apple-clang",
                            "build_type":"Debug",
                            "compiler.version":"9.1"
                        },
                        "requires":[

                        ],
                        "outdated":false
                    },
                    {
                        "id":"03369b0caf8c0c8d4bb84d5136112596bde4652d",
                        "options":{
```


(continues on next page)

(continued from previous page)


```
                                "shared":"True",
                                "asynchronous":"False",
                                "SSL":"False"
                            },
                            "settings":{
                                "os":"Linux",
                                "arch":"x86",
                                "compiler":"gcc",
                                "build_type":"Release",
                                "compiler.version":"5"
                            },
                            "requires":[

                            ],
                            "outdated":false
                        }
                    ]
                }
            ]
        }
    ]
}
```


- Search references in local cache: **conan search paho-c* --json search.json**


```
{
    "error":false,
    "results":[
        {
            "remote":"None",
            "items":[
                {
                    "recipe":{
                        "id":"paho-c/1.2.0@danimtb/testing"
                    }
                }
            ]
        }
    ]
}
```


- Search packages of a reference in local cache: **conan search paho-c/1.2.0@danimtb/testing --json search.json**


```
{
    "error":false,
    "results":[
        {
            "remote":"None",
            "items":[
                {
                    "recipe":{
```


(continues on next page)

(continued from previous page)


```
            "id":"paho-c/1.2.0@danimtb/testing"
        },
        "packages":[
            {
                "id":"6cc50b139b9c3d27b3e9042d5f5372d327b3a9f7",
                "options":{
                    "SSL":"False",
                    "asynchronous":"False",
                    "shared":"False"
                },
                "settings":{
                    "arch":"x86_64",
                    "build_type":"Release",
                    "compiler":"Visual Studio",
                    "compiler.runtime":"MD",
                    "compiler.version":"15",
                    "os":"Windows"
                },
                "requires":[

                ],
                "outdated":false
            },
            {
                "id":"95cd13dfc3f6b80d3ccb2a38441e3a1ad88e5a15",
                "options":{
                    "SSL":"False",
                    "asynchronous":"True",
                    "shared":"True"
                },
                "settings":{
                    "arch":"x86_64",
                    "build_type":"Release",
                    "compiler":"Visual Studio",
                    "compiler.runtime":"MD",
                    "compiler.version":"15",
                    "os":"Windows"
                },
                "requires":[

                ],
                "outdated":true
            },
            {
                "id":"970e773c5651dc2560f86200a4ea56c23f568ff9",
                "options":{
                    "SSL":"False",
                    "asynchronous":"False",
                    "shared":"True"
                },
                "settings":{
                    "arch":"x86_64",
```


(continues on next page)

(continued from previous page)


```
                            "build_type":"Release",
                            "compiler":"Visual Studio",
                            "compiler.runtime":"MD",
                            "compiler.version":"15",
                            "os":"Windows"
                        },
                        "requires":[

                        ],
                        "outdated":true
                    },
                    {
                        "id":"c4c0a49b09575515ce1dd9841a48de0c508b9d7c",
                        "options":{
                            "SSL":"True",
                            "asynchronous":"False",
                            "shared":"True"
                        },
                        "settings":{
                            "arch":"x86_64",
                            "build_type":"Release",
                            "compiler":"Visual Studio",
                            "compiler.runtime":"MD",
                            "compiler.version":"15",
                            "os":"Windows"
                        },
                        "requires":[
                            "openssl/1.0.2n@conan/
↪stable:606fdb601e335c2001bdf31d478826b644747077",
                            "zlib/1.2.11@conan/
↪stable:6cc50b139b9c3d27b3e9042d5f5372d327b3a9f7"
                        ],
                        "outdated":true
                    },
                    {
                        "id":"db9d6ba7004592ed2598f2c369484d4a01269110",
                        "options":{
                            "SSL":"True",
                            "asynchronous":"False",
                            "shared":"True"
                        },
                        "settings":{
                            "arch":"x86_64",
                            "build_type":"Release",
                            "compiler":"gcc",
                            "compiler.exception":"seh",
                            "compiler.threads":"posix",
                            "compiler.version":"7",
                            "os":"Windows"
                        },
                        "requires":[
                            "openssl/1.0.2n@conan/
```


(continues on next page)

(continued from previous page)

```
↪stable:f761d91cef7988eafb88c6b6179f4cf261609f26",
                                "zlib/1.2.11@conan/
↪stable:6dc82da13f94df549e60f9c1ce4c5d11285a4dff"
                            ],
                            "outdated":true
                        }
                    ]
                }
            ]
        }
    ]
}
```

Upload output

Warning: This is an **experimental** feature subject to breaking changes in future releases.

The **conan upload** provides a `--json` parameter to generate a file containing the information of the upload process.

The output JSON contains a two first level keys:

- **error**: `True` if the upload completed without error, `False` otherwise.
- **uploaded**: A list of uploaded packages. Each element contains:
  - **recipe**: Document representing the uploaded recipe.
    - **id**: Reference, e.g., "openssl/1.0.2u@"
    - **remote_name**: Remote name where the recipe was uploaded.
    - **remote_url**: Remote URL where the recipe was uploaded.
    - **time**: `ISO 8601` string with the time the recipe was uploaded.
  - **packages: List of elements, representing the binary packages uploaded for the recipe.**
    - **id**: Package ID, e.g., "8018a4df6e7d2b4630a814fa40c81b85b9182d2b"
    - **time**: ISO 8601 string with the time the recipe was uploaded.

Example:

```
$ conan upload "h*" -all -r conancenter --json upload.json
```

Listing 3: upload.json

```
{
    "error":false,
    "uploaded":[
        {
            "recipe":{
                "id":"hello/0.1@conan/testing",
                "remote_name":"conancenter",
                "remote_url":"https://center.conan.io",
```

(continues on next page)

(continued from previous page)


```
                "time":"2018-04-30T11:18:19.204728"
            },
            "packages":[
                {
                    "id":"3f3387d49612e03a5306289405a2101383b861f0",
                    "remote_name":"conancenter",
                    "remote_url":"https://center.conan.io",
                    "time":"2018-04-30T11:18:21.534877"
                },
                {
                    "id":"6cc50b139b9c3d27b3e9042d5f5372d327b3a9f7",
                    "remote_name":"conancenter",
                    "remote_url":"https://center.conan.io",
                    "time":"2018-04-30T11:18:23.934152"
                },
                {
                    "id":"889d5d7812b4723bd3ef05693ffd190b1106ea43",
                    "remote_name":"conancenter",
                    "remote_url":"https://center.conan.io",
                    "time":"2018-04-30T11:18:28.195266"
                },
                {
                    "id":"e98aac15065fc710dffd1b4fbee382b087c3ad1d",
                    "remote_name":"conancenter",
                    "remote_url":"https://center.conan.io",
                    "time":"2018-04-30T11:18:30.495989"
                }
            ]
        },
        {
            "recipe":{
                "id":"hello0/1.2.1@conan/testing",
                "remote_name":"conancenter",
                "remote_url":"https://center.conan.io",
                "time":"2018-04-30T11:18:32.688651"
            },
            "packages":[
                {
                    "id":"5ab84d6acfe1f23c4fae0ab88f26e3a396351ac9",
                    "remote_name":"conancenter",
                    "remote_url":"https://center.conan.io",
                    "time":"2018-04-30T11:18:34.991721"
                }
            ]
        },
        {
            "recipe":{
                "id":"hello_app/0.1@conan/testing",
                "remote_name":"conancenter",
                "remote_url":"https://center.conan.io",
                "time":"2018-04-30T11:18:36.901333"
            },
```


(continues on next page)

(continued from previous page)


```
            "packages":[
                {
                    "id":"6cc50b139b9c3d27b3e9042d5f5372d327b3a9f7",
                    "remote_name":"conancenter",
                    "remote_url":"https://center.conan.io",
                    "time":"2018-04-30T11:18:39.243895"
                }
            ]
        },
        {
            "recipe":{
                "id":"hello_python_conan/0.1@conan/testing",
                "remote_name":"conancenter",
                "remote_url":"https://center.conan.io",
                "time":"2018-04-30T11:18:41.181543"
            },
            "packages":[
                {
                    "id":"5ab84d6acfe1f23c4fae0ab88f26e3a396351ac9",
                    "remote_name":"conancenter",
                    "remote_url":"https://center.conan.io",
                    "time":"2018-04-30T11:18:43.749422"
                }
            ]
        },
        {
            "recipe":{
                "id":"hello_python_reuse_conan/0.1@conan/testing",
                "remote_name":"conancenter",
                "remote_url":"https://center.conan.io",
                "time":"2018-04-30T11:18:45.614096"
            },
            "packages":[
                {
                    "id":"6a051b2648c89dbd1f8ada0031105b287deea9d2",
                    "remote_name":"conancenter",
                    "remote_url":"https://center.conan.io",
                    "time":"2018-04-30T11:18:47.942491"
                }
            ]
        },
        {
            "recipe":{
                "id":"hdf5/1.8.20@acri/testing",
                "remote_name":"conancenter",
                "remote_url":"https://center.conan.io",
                "time":"2018-04-30T11:18:48.291756"
            },
            "packages":[

            ]
        },
```


(continues on next page)

(continued from previous page)

```
        {
            "recipe":{
                "id":"http_parser/2.9.2",
                "remote_name":"conancenter",
                "remote_url":"https://center.conan.io",
                "time":"2018-04-30T11:18:48.637576"
            },
            "packages":[
                {
                    "id":"6cc50b139b9c3d27b3e9042d5f5372d327b3a9f7",
                    "remote_name":"conancenter",
                    "remote_url":"https://center.conan.io",
                    "time":"2018-04-30T11:18:51.125189"
                }
            ]
        }
    ]
}
```

User output

Warning: This is an **experimental** feature subject to breaking changes in future releases.

The **conan user** provides a **--json** parameter to generate a file containing the information of the users configured per remote.

The output JSON contains a two first level keys:

- **error**: Boolean indicating whether command completed with error.
- **remotes**: A list of the remotes with the packages found. Each element contains:
  - **name**: Name of the remote.
  - **user_name**: Name of the user set for that remote.
  - **authenticated**: Boolean indicating if user is authenticated or not.

Example:

List users per remote: **conan user --json user.json**

Listing 4: *user.json*

```
{
    "error":false,
    "remotes":[
        {
            "name":"conancenter",
            "user_name":"danimtb",
            "authenticated":true
        },
        {
```

(continues on next page)

(continued from previous page)

```
            "name":"bincrafters",
            "user_name":null,
            "authenticated":false
        },
        {
            "name":"the_remote",
            "user_name":"foo",
            "authenticated":false
        }
    ]
}
```

Info output

Warning: This is an **experimental** feature subject to breaking changes in future releases.

The **conan info** provides a **--json** parameter to generate a file containing the output of the command.

There are several possible outputs depending on other arguments:

Build order

Warning: The command `conan info --build-order` is deprecated in favor of *conan lock build-order*.

The build order printed with the argument **--build-order** can be formatted as JSON. It will show a list of lists where the references inside each nested one can be built in parallel.

Listing 5: build_order.json

```
{
    "groups":[
        [
            "liba/0.1@lasote/stable",
            "libe/0.1@lasote/stable",
            "libf/0.1@lasote/stable"
        ],
        [
            "libb/0.1@lasote/stable",
            "libc/0.1@lasote/stable"
        ]
    ]
}
```

Nodes to build

When called with the argument **--build** it will retrieve the list of nodes to be built according to the build policy. Output will be just a list of references.

Listing 6: nodes_to_build.json

```
[
    "h0/0.1@lu/st",
    "h1a/0.1@lu/st",
    "h1c/0.1@lu/st",
    "h2a/0.1@lu/st",
    "h2c/0.1@lu/st"
]
```

Info output

The output of a **conan info** call over a reference or a path gives information about all the nodes involved in its build graph; the generated JSON file will contain a list with the information for each of the nodes.

Listing 7: info.json

```
[
    {
        "reference":"liba/0.1@lasote/stable",
        "is_ref":true,
        "display_name":"liba/0.1@lasote/stable",
        "id":"8da7d879f40d12efabc9a1f26ab12f1b6cafb6ad",
        "build_id":null,
        "context": "host",
        "url":"myurl",
        "license":[
            "MIT"
        ],
        "description": "project A",
        "recipe":"No remote",
        "binary":"Missing",
        "creation_date":"2019-01-29 17:22:41",
        "required_by":[
            "libc/0.1@lasote/stable",
            "libb/0.1@lasote/stable"
        ]
    },
    {
        "reference":"libb/0.1@lasote/stable",
        "is_ref":true,
        "display_name":"libb/0.1@lasote/stable",
        "id":"c4ec2bf350e2a02405029ab366535e26372a4f63",
        "build_id":null,
        "context": "host",
        "url":"myurl",
        "license":[
```

(continues on next page)

(continued from previous page)

```
            "MIT"
        ],
        "description": "project C",
        "recipe":"No remote",
        "binary":"Missing",
        "creation_date":"2019-01-29 17:22:41",
        "required_by":[
            "conanfile.py (libd/0.1@None/None)"
        ],
        "requires":[
            "liba/0.1@lasote/stable",
            "libe/0.1@lasote/stable"
        ]
    },
    { "...": "..."}
]
```

Note: As this is a marked as *experimental*, some fields may be removed or added.

Config output

Warning: This is an **experimental** feature subject to breaking changes in future releases.

The **conan config home** provides a `--json` parameter to generate a file containing the information of the conan home directory.

```
$ conan config home --json home.json
```

It will create a JSON file like:

Listing 8: home.json

```
{
    "home": "/path/to/conan/home"
}
```

18.1.6 Return codes

Return Codes

The Conan client returns different exit codes for every command depending on the situation:

Success

Return code: `0`

Execution terminated successfully

General error

Return code: `1`

Execution terminated with a general error, normally caused by a `ConanException`.

Migration error

Return code: `2`

Execution terminated with an error migrating configuration files to new format.

User Ctrl+C

Return code: `3`

Execution terminated due to manually stopping the process with `Ctrl+C` key combination.

User Ctrl+Break

Return code: `4`

Execution terminated due to manually stopping the profess with `Ctrl+Break` key combination.

SIGTERM

Return code: `5`

Execution terminated due to `SIGTERM` signal.

Invalid configuration

Return code: `6`

Execution terminated due to an exception caused by a `ConanInvalidConfiguration`. This exit code can be considered a success as it is expected for *configurations not supported by the recipe*.

18.2 conanfile.txt

Reference for *conanfile.txt* sections: requires, generators, etc.

18.2.1 Sections

[requires]

List of requirements, specifying the full reference.

```
[requires]
poco/1.9.4
zlib/1.2.11
```

This section supports references with *version ranges*:

```
[requires]
poco/[>1.0,<1.9]
zlib/1.2.11
```

[tool_requires]

List of tool requirements specifying the full reference.

```
[tool_requires]
7zip/16.00
```

This section supports references with *version ranges*.

In practice the `[tool_requires]` will be always installed (same as `[requires]`) as installing from a *conanfile.txt* means that something is going to be built, so the tool requirements are indeed needed.

It is useful and conceptually cleaner to have them in separate sections, so users of this *conanfile.txt* might quickly identify some dev-tools that they have already installed on their machine, differentiating them from the required libraries to link with.

[generators]

List of *generators*.

```
[requires]
poco/1.9.4
zlib/1.2.11

[generators]
xcode
cmake
qmake
```

[options]

List of *options* scoped for each package like **package_name:option = Value**.

```
[requires]
poco/1.9.4
zlib/1.2.11

[generators]
cmake

[options]
poco:shared=True
openssl:shared=True
```

[imports]

List of files to be imported to a local directory. Read more: *imports*.

```
[requires]
poco/1.9.4
zlib/1.2.11

[generators]
cmake

[options]
poco:shared=True
openssl:shared=True

[imports]
bin, *.dll -> ./bin # Copies all dll files from packages bin folder to my local "bin"␣
↪folder
lib, *.dylib* -> ./bin # Copies all dylib files from packages lib folder to my local "bin
↪" folder
```

The first item is the subfolder of the packages (could be the root "." one), the second is the pattern to match. Both relate to the local cache. The third (after the arrow) item, is the destination folder, living in user space, not in the local cache.

The `[imports]` section also support the same arguments as the equivalent `imports()` method in *conanfile.py*, separated with an `@`.

Note: If your previous folders use an `@` in the path name, use a trailing (even if empty) `@` so the parser correctly gets the folders paths, e.g: `lib, * -> /home/jenkins/workspace/conan_test@2/g/install/lib @`

- **root_package** (Optional, Defaulted to *all packages in deps*): fnmatch pattern of the package name ("OpenCV", "Boost") from which files will be copied.
- **folder**: (Optional, Defaulted to `False`). If enabled, it will copy the files from the local cache to a subfolder named as the package containing the files. Useful to avoid conflicting imports of files with the same name (e.g. License).
- **ignore_case**: (Optional, Defaulted to `False`). If enabled will do a case-insensitive pattern matching.

- **excludes**: (Optional, Defaulted to `None`). Allows defining a list of patterns (even a single pattern) to be excluded from the copy, even if they match the main `pattern`.
- **keep_path** (Optional, Defaulted to `True`): Means if you want to keep the relative path when you copy the files from the **src** folder to the **dst** one. Useful to ignore (`keep_path=False`) path of *library.dll* files in the package it is imported from.

Example to collect license files from dependencies into a *licenses* folder, excluding (just an example) *.html* and *.jpeg* files:

```
[imports]
., license* -> ./licenses @ folder=True, ignore_case=True, excludes=*.html *.jpeg
```

[layout]

This is a feature introduced in Conan 1.49

You can specify one name of a predefined layout. The available values are:

- *cmake_layout*
- *vs_layout*
- *bazel_layout*

```
[layout]
cmake_layout
```

Comments

A comment starts with a hash character (#) and ends at the end of the physical line. Comments are ignored by the syntax; they are not tokens.

18.3 conanfile.py

Reference for *conanfile.py*: attributes, methods, etc.

Important: *conanfile.py* recipes uses a variety of attributes and methods to operate. In order to avoid collisions and conflicts, follow these rules:

- Public attributes and methods, like `build()`, `self.package_folder`, are reserved for Conan. Don't use public members for custom fields or methods in the recipes.
- Use "protected" access for your own members, like `self._my_data` or `def _my_helper(self):`. Conan only reserves "protected" members starting with `_conan`.

Contents:

18.3.1 Attributes

name

This is a string, with a minimum of 2 and a maximum of 50 characters (though shorter names are recommended), that defines the package name. It will be the `<pkgName>/version@user/channel` of the package reference. It should match the following regex `^[a-zA-Z0-9_][a-zA-Z0-9_\+\.-]{1,50}$`, so start with alphanumeric or underscore, then alphanumeric, underscore, +, ., - characters.

The name is only necessary for `export`-ing the recipe into the local cache (`export` and `create` commands), if they are not defined in the command line. It might take its value from an environment variable, or even any python code that defines it (e.g. a function that reads an environment variable, or a file from disk). However, the most common and suggested approach would be to define it in plain text as a constant, or provide it as command line arguments.

version

The version attribute will define the version part of the package reference: `pkgName/<version>@user/channel` It is a string, and can take any value, matching the same constraints as the `name` attribute. In case the version follows semantic versioning in the form `X.Y.Z-pre1+build2`, that value might be used for requiring this package through version ranges instead of exact versions.

The version is only strictly necessary for `export`-ing the recipe into the local cache (`export` and `create` commands), if they are not defined in the command line. It might take its value from an environment variable, or even any python code that defines it (e.g. a function that reads an environment variable, or a file from disk). Please note that this value might be used in the recipe in other places (as in `source()` method to retrieve code from elsewhere), making this value not constant means that it may evaluate differently in different contexts (e.g., on different machines or for different users) leading to unrepeatable or unpredictable results. The most common and suggested approach would be to define it in plain text as a constant, or provide it as command line arguments.

description

This is an optional, but strongly recommended text field, containing the description of the package, and any information that might be useful for the consumers. The first line might be used as a short description of the package.

```
class HelloConan(ConanFile):
    name = "hello"
    version = "0.1"
    description = """This is a Hello World library.
                     A fully featured, portable, C++ library to say Hello World in the␣
↪stdout,
                     with incredible iostreams performance"""
```

homepage

Use this attribute to indicate the home web page of the library being packaged. This is useful to link the recipe to further explanations of the library itself like an overview of its features, documentation, FAQ as well as other related information.

```
class EigenConan(ConanFile):
    name = "eigen"
    version = "3.3.4"
    homepage = "http://eigen.tuxfamily.org"
```

url

It is possible, even typical, if you are packaging a third party lib, that you just develop the packaging code. Such code is also subject to change, often via collaboration, so it should be stored in a VCS like git, and probably put on GitHub or a similar service. If you do indeed maintain such a repository, please indicate it in the `url` attribute, so that it can be easily found.

```
class HelloConan(ConanFile):
    name = "hello"
    version = "0.1"
    url = "https://github.com/conan-io/hello.git"
```

The `url` is the url **of the package** repository, i.e. not necessarily the original source code. It is optional, but highly recommended, that it points to GitHub, Bitbucket or your preferred code collaboration platform. Of course, if you have the conanfile inside your library source, you can point to it, and afterwards use the `url` in your `source()` method.

This is a recommended, but not mandatory attribute.

license

This field is intended for the license of the **target** source code and binaries, i.e. the code that is being packaged, not the `conanfile.py` itself. This info is used to be displayed by the **`conan info`** command and possibly other search and report tools.

```
class HelloConan(ConanFile):
    name = "hello"
    version = "0.1"
    license = "MIT"
```

This attribute can contain several, comma separated licenses. It is a text string, so it can contain any text, including hyperlinks to license files elsewhere.

However, we strongly recommend packagers of Open-Source projects to use SPDX identifiers from the SPDX license list instead of free-formed text. This will help people wanting to automate license compatibility checks, like consumers of your package, or you if your package has Open-Source dependencies.

This is a recommended, but not mandatory attribute.

author

Intended to add information about the author, in case it is different from the Conan user. It is possible that the Conan user is the name of an organization, project, company or group, and many users have permissions over that account. In this case, the author information can explicitly define who is the creator/maintainer of the package

```
class HelloConan(ConanFile):
    name = "hello"
    version = "0.1"
    author = "John J. Smith (john.smith@company.com)"
```

This is an optional attribute.

topics

Topics provide a useful way to group related tags together and to quickly tell developers what a package is about. Topics also make it easier for customers to find your recipe. It could be useful to filter packages by topics.

The `topics` attribute should be a tuple with the needed topics inside.

```
class ProtocInstallerConan(ConanFile):
    name = "protoc_installer"
    version = "0.1"
    topics = ("protocol-buffers", "protocol-compiler", "serialization", "rpc")
```

This is an optional attribute.

user, channel

These fields are optional in a Conan reference, they could be useful to identify a forked recipe from the community with changes specific for your company. Using these fields you may keep the same `name` and `version` and use the `user/channel` to disambiguate your recipe.

The value of these fields can be accessed from within a `conanfile.py`:

```
from conans import ConanFile

class HelloConan(ConanFile):
    name = "hello"
    version = "0.1"

    def requirements(self):
        self.requires("common-lib/version")
        if self.user and self.channel:
            # If the recipe is using them, I want to consume my fork.
            self.requires("say/0.1@%s/%s" % (self.user, self.channel))
        else:
            # otherwise, I'll consume the community one
            self.requires("say/0.1")
```

Only packages that have already been exported (packages in the local cache or in a remote server) can have a user/channel assigned. For package recipes working in the user space, there is no current user/channel by default, although they can be defined at **`conan install`** time with:

```
$ conan install <path to conanfile.py> user/channel
```

See also:

FAQ: *Is there any recommendation regarding which <user> or <channel> to use in a reference?*

> **Warning:** Environment variables `CONAN_USERNAME` and `CONAN_CHANNEL` that were used to assign a value to these fields are now deprecated and will be removed in Conan 2.0. Don't use them to populate the value of `self.user` and `self.channel`.

default_user, default_channel

For package recipes working in the user space, with local methods like **conan install .** and **conan build .**, there is no current user/channel. If you are accessing to `self.user` or `self.channel` in your recipe, you need to declare the environment variables `CONAN_USERNAME` and `CONAN_CHANNEL` or you can set the attributes `default_user` and `default_channel`. You can also use python `@property`:

```
from conans import ConanFile

class HelloConan(ConanFile):
    name = "hello"
    version = "0.1"
    default_user = "myuser"

    @property
    def default_channel(self):
        return "mydefaultchannel"

    def requirements(self):
        self.requires("pkg/0.1@%s/%s" % (self.user, self.channel))
```

settings

There are several things that can potentially affect a package being created, i.e. the final package will be different (a different binary, for example), if some input is different.

Development project-wide variables, like the compiler, its version, or the OS itself. These variables have to be defined, and they cannot have a default value listed in the conanfile, as it would not make sense.

It is obvious that changing the OS produces a different binary in most cases. Changing the compiler or compiler version changes the binary too, which might have a compatible ABI or not, but the package will be different in any case.

For these reasons, the most common convention among Conan recipes is to distinguish binaries by the following four settings, which is reflected in the *conanfile.py* template used in the *conan new* command:

```
settings = "os", "compiler", "build_type", "arch"
```

When Conan generates a compiled binary for a package with a given combination of the settings above, it generates a unique ID for that binary by hashing the current values of these settings.

But what happens for example to **header only libraries**? The final package for such libraries is not binary and, in most cases it will be identical, unless it is automatically generating code. We can indicate that in the conanfile:

```
from conans import ConanFile

 class HelloConan(ConanFile):
     name = "hello"
     version = "0.1"
     # We can just omit the settings attribute too
     settings = None

     def build(self):
         #empty too, nothing to build in header only
```

You can restrict existing settings and accepted values as well, by redeclaring the settings attribute:

```
class HelloConan(ConanFile):
    settings = {"os": ["Windows"],
        "compiler": {"Visual Studio": {"version": [11, 12]}},
        "arch": None}
```

In this example we have just defined that this package only works in Windows, with VS 10 and 11. Any attempt to build it in other platforms with other settings will throw an error saying so. We have also defined that the runtime (the MD and MT flags of VS) is irrelevant for us (maybe we using a universal one?). Using None as a value means, *maintain the original values* in order to avoid re-typing them. Then, "arch": None is totally equivalent to "arch": ["x86", "x86_64", "arm"] Check the reference or your ~/.conan/settings.yml file.

As re-defining the whole settings attribute can be tedious, it is sometimes much simpler to remove or tune specific fields in the `configure()` method. For example, if our package is runtime independent in VS, we can just remove that setting field:

```
settings = "os", "compiler", "build_type", "arch"

def configure(self):
    self.settings.compiler["Visual Studio"].remove("runtime")
```

It is possible to check the settings to implement conditional logic, with attribute syntax:

```
def build(self):
    if self.settings.os == "Windows" and self.settings.compiler.version == "15":
        # do some special build commands
    elif self.settings.arch == "x86_64":
        # Other different commands
```

Those comparisons do content checking, for example if you do a typo like `self.settings.os == "Windos"`, Conan will fail and tell you that is not a valid `settings.os` value, and the possible range of values.

Likewise, if you try to access some setting that doesn't exist, like `self.settings.compiler.libcxx` for the `Visual Studio` setting, Conan will fail telling that `libcxx` does not exist for that compiler.

If you want to do a safe check of settings values, you could use the `get_safe()` method:

```
def build(self):
    # Will be None if doesn't exist
    arch = self.settings.get_safe("arch")
    # Will be None if doesn't exist
    compiler_version = self.settings.get_safe("compiler.version")
    # Will be the default version if the return is None
    build_type = self.settings.get_safe("build_type", default="Release")
```

The `get_safe()` method will return `None` if that setting or subsetting doesn't exist and there is no default value assigned.

options

Conan provides this attribute to declare traits which will affect only one reference, unlike the settings that are typically the same for all the recipes in a Conan graph. Options are declared per recipe, this attribute consist on a dictionary where the key is the option name and the value is the list of different values that the option can take.

Important: All the options with their values are encoded into the package ID, as everything that affects the generated binary. See *configure(), config_options()* and *package_id()* methods for information about removing certain options for some configurations.

A very common one is the option `shared` with allowed values of `[True, False]` that many recipes declare and use to configure the build system to produce a static library or a shared library.

Values for each option can be typed or plain strings (`"value"`, `True`, `None`, `42`,...) and there is a special value, `"ANY"`, for options that can take any value.

```
class MyPkg(ConanFile):
    ...
    options = {
        "shared": [True, False],
        "option1": ["value1", "value2"],
        "option2": "ANY",
        "option3": [None, "value1", "value2"],
        "option4": [True, False, "value"],
    }
```

Every option in a recipe needs to be assigned a value from the ones declared in the `options` attribute. The consumer can define options using different methods: command line, profile or consumer recipes; **an uninitialized option will get the value `None` and it will be a valid value if it is contained in the list of valid values**. Invalid values will produce an error. See attribute *default_options* for a way to declare a default value for options in a recipe.

Tip:

- You can inspect available package options reading the package recipe, which can be done with the command **`conan inspect mypkg/0.1@user/channel`**.

Tip:

- Options `"shared": [True, False]` and `"fPIC": [True, False]` are automatically managed in *CMake* and *AutoToolsBuildEnvironment (configure/make)* build helpers.

Define the value of an option

As we mentioned before, values for options in a recipe can be defined using different ways, let's go over all of them for the example recipe `mypkg` defined above:

- In the recipe that declares the option:
  - Using the attribute `default_options` in the recipe itself.
  - In the `config_options()` method of the recipe.
  - In the `configure()` method of the recipe itself (**this one has the highest precedence**, this value can't be overriden)

```
class MyPkg(ConanFile):

    options = {
        "shared": [True, False],
        "option1": ["value1", "value2"],
        "option2": "ANY",
    }

    def configure(self):
        if some_condition:
            self.options.shared = False
```

- From a recipe that requires this one:
  - using the `default_options` attribute of the consumer:

```
class OtherPkg(ConanFile):
    requires = "mypkg/0.1@user/channel"
    default_options = {"mypkg:shared": False}
```

  - in the `configure()` method of the consumer (**highest precedence after `configure()` in the recipe itself**):

```
class OtherPkg(ConanFile):
    requires = "mypkg/0.1@user/channel"

    def configure(self):
        self.options['mypkg'].shared = False
```

    This method allows to assign values based on other conditions, it can have some drawbacks as it is explainded in the *mastering section*.

- In the *conanfile.txt* file:

```
[requires]
mypkg/0.1@user/channel

[options]
mypkg:shared=False
```

- It is also possible to define default values for the options of a recipe using *profiles*. They will apply whenever that recipe is used:

```
[settings]
setting=value

[options]
mypkg:shared=False
```

- Last way of defining values for options is to pass these values using the command argument **`-o,--option`** in the command line:

```
$ conan install . -o mypkg:shared=True
```

Regarding the precedence of all these ways of assigning a value to an option, it works like any other configuration element in Conan: the closer to the consumer and the command line the higher the precedence. The list above is

ordered from the less priority to the highest one (with the exceptional assignment in `configure()` method which cannot be overridden).

Get the value of an option

Values from options can be retrieved after they are assigned. For options that belong to the same recipe, the value can be retrieved in any method to run logic conditional to their values. **Options from required packages can be retrieved only after the full graph has been resolved**, this means that the value will be available in the methods `validate()`, `build()`, `package()`, `package_info()`. Accessing those values in other methods can lead to unexpected results.

```
class OtherPkg(ConanFile):
    requires = "mypkg/0.1@user/channel"

    def validate(self):
        if self.options['mypkg'].shared:
            raise ConanInvalidConfiguration("Cannot use shared library of requirement
↪'mypkg'")
```

If you want to retrieve the value of an option and fallback to a known value if the option doesn't exist you can use the `get_safe()` method:

```
def build(self):
    # Will return None if doesn't exist
    fpic = self.options.get_safe("fPIC")
    # Will return the default value if the return is None
    shared = self.options.get_safe("shared", default=False)
```

The `get_safe()` method will return `None` if that option doesn't exist and there is no default value assigned.

Evaluate options

It is very important to know how the options are evaluated in conditional expressions and how the comparison operator works with them:

- Evaluation for the typed value and the string one is the same, so all these inputs would behave the same:
  - `default_options = {"shared": True, "option": None}`
  - `default_options = {"shared": "True", "option": "None"}`
  - `mypkg:shared=True`, `mypkg:option=None` on profiles, command line or *conanfile.txt*
- **Implicit conversion to boolean is case insensitive**, so the expression `bool(self.options.option)`:
  - equals `True` for the values `True`, `"True"` and `"true"`, and any other value that would be evaluated the same way in Python code.
  - equals `False` for the values `False`, `"False"` and `"false"`, also for the empty string and for `0` and `"0"` as expected.
- Comparison using `is` is always equals to `False` because the types would be different as the option value is encapsulated inside a Python class.
- Explicit comparisons with the == symbol **are case sensitive**, so:
  - `self.options.option = "False"` satisfies `assert self.options.option == False`, `assert self.options.option == "False"`, but `assert self.options.option != "false"`.
- A different behavior has `self.options.option = None`, because `assert self.options.option != None`.

default_options

The attribute `default_options` has the purpose of defining the default values for the options if the consumer (consuming recipe, project, profile or the user through the command line) does not define them. This attribute should be defined as a python dictionary:

```
class MyPkg(ConanFile):
    ...
    options = {"build_tests": [True, False],
               "option1": ["value1", "value2"],
               "option2": "ANY"}
    default_options = {"build_tests": True,
                       "option1": "value1",
                       "option2": 42}

    def build(self):
        cmake = CMake(self)
        cmake.definitions['BUILD_TESTS'] = self.options.build_tests
        cmake.configure()
        ...
```

Remember that you can also assign default values for options of your requirements as we've seen in the attribute *options*.

You can also set the options conditionally to a final value with `configure()` instead of using `default_options`:

```
class OtherPkg(ConanFile):
    settings = "os", "arch", "compiler", "build_type"
    options = {"some_option": [True, False]}
    # Do NOT declare 'default_options', use 'config_options()'

    def configure(self):
        if self.options.some_option == None:
            if self.settings.os == 'Android':
                self.options.some_option = True
            else:
                self.options.some_option = False
```

Take into account that if a value is assigned in the `configure()` method it cannot be overridden.

Important: Default options can be specified as a dictionary only for Conan version >= 1.8.

See also:

Read more about the *config_options()* method.

requires

Specify package dependencies as a list or tuple of other packages:

```
class MyLibConan(ConanFile):
    requires = "hello/1.0@user/stable", "OtherLib/2.1@otheruser/testing"
```

You can specify further information about the package requirements:

```
class MyLibConan(ConanFile):
    requires = [("hello/0.1@user/testing"),
                ("say/0.2@dummy/stable", "override"),
                ("bye/2.1@coder/beta", "private")]
```

```
class MyLibConan(ConanFile):
    requires = (("hello/1.0@user/stable", "private"), )
```

Requirements can be complemented by 2 different parameters:

private: a dependency can be declared as private if it is going to be fully embedded and hidden from consumers of the package. It might be necessary in some extreme cases, like having to use two different versions of the same library (provided that they are totally hidden in a shared library, for example), but it is mostly discouraged otherwise.

override: packages can define overrides of their dependencies, if they require the definition of specific versions of the upstream required libraries, but not necessarily direct dependencies. For example, a package can depend on A(v1.0), which in turn could conditionally depend on Zlib(v2), depending on whether the compression is enabled or not. Now, if you want to force the usage of Zlib(v3) you can:

```
class HelloConan(ConanFile):
    requires = ("ab/1.0@user/stable", ("zlib/3.0@other/beta", "override"))
```

This **will not introduce a new dependency**, it will just change `zlib/2.0` to `zlib/3.0` if `ab` actually requires it. Otherwise zlib will not be a dependency of your package.

Note: To prevent accidental override of transitive dependencies, check the config variable *general.error_on_override* or the environment variable *CONAN_ERROR_ON_OVERRIDE*.

---

version ranges

The syntax is using brackets:

```
class HelloConan(ConanFile):
    requires = "pkg/[>1.0 <1.8]@user/stable"
```

Expressions are those defined and implemented by [python node-semver](https://pypi.org/project/node-semver/). Accepted expressions would be:

```
>1.1 <2.1    # In such range
2.8          # equivalent to =2.8
~=3.0        # compatible, according to semver
>1.1 || 0.8  # conditions can be OR'ed
```

Go to *Mastering/Version Ranges* if you want to learn more about version ranges.

tool_requires

Tool requirements are requirements that are only installed and used when the package is built from sources. If there is an existing pre-compiled binary, then the tool requirements for this package will not be retrieved.

They can be specified as a comma separated tuple in the package recipe:

```
class MyPkg(ConanFile):
    tool_requires = "tool_a/0.2@user/testing", "tool_b/0.2@user/testing"
```

Read more: *Tool requirements*

exports

This **optional attribute** declares the set of files that should be exported and stored side by side with the *conanfile.py* file to make the recipe work: other python files that the recipe will import, some text file with data to read,...

The `exports` field can declare one single file or pattern, or a list of any of the previous elements. Patterns use fnmatch formatting to declare files to include or exclude.

For example, if we have some python code that we want the recipe to use in a `helpers.py` file, and have some text file *info.txt* we want to read and display during the recipe evaluation we would do something like:

```
exports = "helpers.py", "info.txt"
```

Exclude patterns are also possible, with the `!` prefix:

```
exports = "*.py", "!*tmp.py"
```

exports_sources

This **optional attribute** declares the set of files that should be exported together with the recipe and will be available to generate the package. Unlike `exports` attribute, these files shouldn't be used by the *conanfile.py* Python code, but to compile the library or generate the final package. And, due to its purpose, these files will only be retrieved if requested binaries are not available or the user forces Conan to compile from sources.

The `exports_sources` attribute can declare one single file or pattern, or a list of any of the previous elements. Patterns use fnmatch formatting to declare files to include or exclude.

Together with the `source()` and `imports()` methods, and the *SCM feature*, this is another way to retrieve the sources to create a package. Unlike the other methods, files declared in `exports_sources` will be exported together with the *conanfile.py* recipe, so, if nothing else is required, it can create a self-contained package with all the sources (like a snapshot) that will be used to generate the final artifacts.

Some examples for this attribute are:

```
exports_sources = "include*", "src*"
```

Exclude patterns are also possible, with the `!` prefix:

```
exports_sources = "include*", "src*", "!src/build/*"
```

Note, if the recipe defines the `layout()` method and specifies a `self.folders.source = "src"` it won't affect where the files (from the `exports_sources`) are copied. They will be copied to the base source folder. So, if you want to replace some file that got into the `source()` method, you need to explicitly copy it from the parent folder or even better, from `self.export_sources_folder`.

```
import os, shutil
from conan import ConanFile
from conan.tools.files import save, load

class Pkg(ConanFile):
   ...
   exports_sources = "CMakeLists.txt"

   def layout(self):
       self.folders.source = "src"
       self.folders.build = "build"

   def source(self):
       # emulate a download from web site
       save(self, "CMakeLists.txt", "MISTAKE: Very old CMakeLists to be replaced")
       # Now I fix it with one of the exported files
       shutil.copy("../CMakeLists.txt", ".")
       shutil.copy(os.path.join(self.export_sources_folder, "CMakeLists.txt", "."))
```

generators

Generators specify which is the output of the **install** command in your project folder. By default, a *conanbuildinfo.txt* file is generated, but you can specify different generators and even use more than one.

```
class MyLibConan(ConanFile):
    generators = "cmake", "gcc"
```

You can also set the generators conditionally in the *configure() method* like in the example below.

```
class MyLibConan(ConanFile):
    settings = "os", "compiler", "arch", "build_type"
    def configure(self):
        if self.settings.os == "Windows":
            self.generators = ["msbuild"]
```

Check the full *generators list*.

should_configure, should_build, should_install, should_test

Read only variables defaulted to `True`.

These variables allow you to control the build stages of a recipe during a **conan build** command with the optional arguments **--configure**/**--build**/**--install**/**--test**. For example, consider this `build()` method:

```
def build(self):
    cmake = CMake(self)
    cmake.configure()
    cmake.build()
    cmake.install()
    cmake.test()
```

If nothing is specified, all four methods will be called. But using command line arguments, this can be changed:

```
$ conan build . --configure  # only run cmake.configure(). Other methods will do nothing
$ conan build . --build      # only run cmake.build(). Other methods will do nothing
$ conan build . --install    # only run cmake.install(). Other methods will do nothing
$ conan build . --test       # only run cmake.test(). Other methods will do nothing
# They can be combined
$ conan build . -c -b # run cmake.configure() + cmake.build(), but not cmake.install()␣
↪nor cmake.test()
```

Autotools and Meson helpers already implement the same functionality. For other build systems, you can use these variables in the `build()` method:

```
def build(self):
    if self.should_configure:
        # Run my configure stage
    if self.should_build:
        # Run my build stage
    if self.should_install: # If my build has install, otherwise use package()
        # Run my install stage
    if self.should_test:
        # Run my test stage
```

Note that the `should_configure`, `should_build`, `should_install`, `should_test` variables will always be `True` while building in the cache and can be only modified for the local flow with **`conan build`**.

build_policy

With the `build_policy` attribute the package creator can change conan's build behavior. The allowed `build_policy` values are:

- `missing`: If this package is not found as a binary package, Conan will build it from source.
- `always`: This package will always be built from source, also **retrieving the source code each time** by executing the "source" method.

```
class PocoTimerConan(ConanFile):
    build_policy = "always" # "missing"
```

short_paths

This attribute is specific to Windows, and ignored on other operating systems. It tells Conan to workaround the limitation of 260 chars in Windows paths.

Important: Since Windows 10 (ver. 10.0.14393), it is possible to enable long paths at the system level. Latest python 2.x and 3.x installers enable this by default. With the path limit removed both on the OS and on Python, the `short_paths` functionality becomes unnecessary, and can be disabled explicitly through the `CONAN_USER_HOME_SHORT` environment variable.

Enabling short paths management will "link" the `source` and `build` directories of the package to a different location, in Windows it will be `C:\.conan\tmpdir`. All the folder layout in the local cache is maintained.

Set `short_paths=True` in your *conanfile.py*:

```
from conans import ConanFile

class ConanFileTest(ConanFile):
    ...
    short_paths = True
```

See also:

There is an *environment variable* `CONAN_USE_ALWAYS_SHORT_PATHS` to force activate this behavior for all packages.

This behavior will also work in Cygwin, the short folder directory will be `/home/<user>/.conan_short` by default, but it can be modified as we've explained before.

no_copy_source

The attribute `no_copy_source` tells the recipe that the source code will not be copied from the `source` folder to the `build` folder. This is mostly an optimization for packages with large source codebases or header-only, to avoid extra copies. It is **mandatory** that the source code must not be modified at all by the configure or build scripts, as the source code will be shared among all builds.

To be able to use it, the package recipe can access the `self.source_folder` attribute, which will point to the `build` folder when `no_copy_source=False` or not defined, and will point to the `source` folder when `no_copy_source=True`.

When this attribute is set to True, the `self.copy()` lines will be called twice, one copying from the `source` folder and the other copying from the `build` folder.

Read *header-only* section for an example using `no_copy_source` attribute.

source_folder

The folder in which the source code lives.

When a package is built in the Conan local cache its value is the same as the `build` folder by default. This is due to the fact that the source code is copied from the `source` folder to the `build` folder to ensure isolation and avoiding modifications of shared common source code among builds for different configurations. Only when `no_copy_source=True` this folder will actually point to the package `source` folder in the local cache.

When executing Conan commands in the *Package development flow* like **`conan source`**, this attribute will be pointing to the folder specified in the command line.

install_folder

The folder in which the installation of packages outputs the generator files with the information of dependencies. By default in the the local cache its value is the same as `self.build_folder` one.

When executing Conan commands in the *Package development flow* like **`conan install`** or **`conan build`**, this attribute will be pointing to the folder specified in the command line.

build_folder

The folder used to build the source code. In the local cache a build folder is created with the name of the package ID that will be built.

When executing Conan commands in the *Package development flow* like **conan build**, this attribute will be pointing to the folder specified in the command line.

package_folder

The folder to copy the final artifacts for the binary package. In the local cache a package folder is created for every different package ID.

When executing Conan commands in the *Package development flow* like **conan package**, this attribute will be pointing to the folder specified in the command line.

recipe_folder

Available since: 1.28.0

The folder where the recipe *conanfile.py* is stored, either in the local folder or in the cache. This is useful in order to access files that are exported along with the recipe.

cpp_info

Important: This attribute is only defined inside `package_info()` method being *None* elsewhere.

The `self.cpp_info` attribute is responsible for storing all the information needed by consumers of a package: include directories, library names, library paths... There are some default values that will be applied automatically if not indicated otherwise.

This object should be filled in `package_info()` method.

NAME	DESCRIPTION
self.cpp_info.includedirs	Ordered list with include paths. Defaulted to `["include"]`
self.cpp_info.libdirs	Ordered list with lib paths. Defaulted to `["lib"]`
self.cpp_info.resdirs	Ordered list of resource (data) paths. Defaulted to `["res"]`
self.cpp_info.bindirs	Ordered list with paths to binaries (executables, dynamic libraries,...). Defaulted to `["bin"]`
self.cpp_info.builddirs	Ordered list with build scripts directory paths. Defaulted to `[""]` (Package folder directory) CMake generators will search in these dirs for files like *findXXX.cmake*
self.cpp_info.libs	Ordered list with the library names, Defaulted to `[]` (empty)
self.cpp_info.defines	Preprocessor definitions. Defaulted to `[]` (empty)
self.cpp_info.cflags	Ordered list with pure C flags. Defaulted to `[]` (empty)
self.cpp_info.cppflags	[DEPRECATED: Use cxxflags instead]
self.cpp_info.cxxflags	Ordered list with C++ flags. Defaulted to `[]` (empty)
self.cpp_info.sharedlinkflags	Ordered list with linker flags (shared libs). Defaulted to `[]` (empty)
self.cpp_info.exelinkflags	Ordered list with linker flags (executables). Defaulted to `[]` (empty)
self.cpp_info.frameworks	Ordered list with the framework names (OSX), Defaulted to `[]` (empty)
self.cpp_info.frameworkdirs	Ordered list with frameworks search paths (OSX). Defaulted to `["Frameworks"]`
self.cpp_info.rootpath	Filled with the root directory of the package, see `deps_cpp_info`
self.cpp_info.name	**[DEPRECATED]** Use `self.cpp_info.names` instead.
self.cpp_info.names[“generator”]	Alternative name for the package used by an specific generator to create files or variables. If set for a generator it will override the information provided by self.cpp_info.name. Like the cpp_info.name, this is only supported by *cmake*, *cmake_multi*, *cmake_find_package*, *cmake_find_package_multi*, *cmake_paths* and *pkg_config* generators.
self.cpp_info.filenames[“generator”]	Alternative name for the filename produced by a specific generator. If set for a generator it will override the “names” value. This is only supported by the *cmake_find_package* and *cmake_find_package_multi* generators.
self.cpp_info.system_libs	Ordered list with the system library names. Defaulted to `[]` (empty)
self.cpp_info.build_modules	Dictionary of lists per generator containing relative paths to build system related utility module files created by the package. Used by CMake

The paths of the directories in the directory variables indicated above are relative to the *self.package_folder* directory.

> **Warning:** Components is a **experimental** feature subject to breaking changes in future releases.

Using components you can achieve a more fine-grained control over individual libraries available in a single Conan package. Components allow you define a `cpp_info` like object per each of those libraries and also requirements between them and to components of other packages (the following case is not a real example):

```
def package_info(self):
    self.cpp_info.names["cmake_find_package"] = "OpenSSL"
    self.cpp_info.names["cmake_find_package_multi"] = "OpenSSL"
    self.cpp_info.components["crypto"].names["cmake_find_package"] = "Crypto"
    self.cpp_info.components["crypto"].libs = ["libcrypto"]
    self.cpp_info.components["crypto"].defines = ["DEFINE_CRYPTO=1"]
    self.cpp_info.components["crypto"].requires = ["zlib::zlib"]  # Depends on all␣
↪components in zlib package

    self.cpp_info.components["ssl"].names["cmake"] = "SSL"
    self.cpp_info.components["ssl"].includedirs = ["include/headers_ssl"]
    self.cpp_info.components["ssl"].libs = ["libssl"]
    self.cpp_info.components["ssl"].requires = ["crypto",
                                                "boost::headers"]  # Depends on headers␣
↪component in boost package
    self.cpp_info.components["ssl"].names["cmake"] = "SSL"

    obj_ext = "obj" if platform.system() == "Windows" else "o"
    self.cpp_info.components["ssl-objs"].objects = [os.path.join("lib", "ssl-object.{}".
↪format(obj_ext))]
```

The interface of the `Component` object is the same as the one used by the `cpp_info` object and has **the same default directories**.

> **Warning:** Using components and global `cpp_info` non-default values or release/debug configurations at the same time is not allowed (except for `self.cpp_info.names`).

Dependencies among components and to components of other requirements can be defined using the `requires` attribute and the name of the component. The dependency graph for components will be calculated and values will be aggregated in the correct order for each field.

New properties model for the cpp_info new tools

> **Warning:** Using `set_property` and `get_property` methods for `cpp_info` is an **experimental** feature subject to breaking changes in future releases.

Using `.names`, `.filenames` and `.build_modules` will not work any more for new generators, like *CMakeDeps* and *PkgConfigDeps*. They have a new way of setting this information using `set_property` and `get_property` methods of the `cpp_info` object (available since Conan 1.36).

```
def set_property(self, property_name, value)
def get_property(self, property_name):
```

To read more about the new `set_property` and `get_property` methods for `cpp_info` please check the dedicated section in the *Conan 2.0 migration guide*.

deps_cpp_info

Contains the `cpp_info` object of the requirements of the recipe. In addition of the above fields, there are also properties to obtain the absolute paths, and `name` and `version` attributes:

NAME	DESCRIPTION
self.deps_cpp_info["dep"].include_paths	"dep" package `includedirs` but transformed to absolute paths
self.deps_cpp_info["dep"].lib_paths	"dep" package `libdirs` but transformed to absolute paths
self.deps_cpp_info["dep"].bin_paths	"dep" package `bindirs` but transformed to absolute paths
self.deps_cpp_info["dep"].build_paths	"dep" package `builddirs` but transformed to absolute paths
self.deps_cpp_info["dep"].res_paths	"dep" package `resdirs` but transformed to absolute paths
self.deps_cpp_info["dep"].framework_paths	"dep" package `frameworkdirs` but transformed to absolute paths
self.deps_cpp_info["dep"].build_modules_paths	"dep" package `build_modules` but transformed to absolute paths
self.deps_cpp_info["dep"].get_name("<generator>")	Get the name declared for the given generator
self.deps_cpp_info["dep"].version	Get the version of the "dep" package
self.deps_cpp_info["dep"].components	**[Experimental]** Dictionary with different components a package may have: libraries, executables...

To get a list of all the dependency names from `deps_cpp_info`, you can call the *deps* member:

```
class PocoTimerConan(ConanFile):
    ...
    def build(self):
        # deps is a list of package names: ["poco", "zlib", "openssl"]
        deps = self.deps_cpp_info.deps
```

It can be used to get information about the dependencies, like used compilation flags or the root folder of the package:

```
class PocoTimerConan(ConanFile):
    ...
    requires = "zlib/1.2.11", "openssl/1.0.2u"
    ...

    def build(self):
        # Get the directory where zlib package is installed
        self.deps_cpp_info["zlib"].rootpath

        # Get the absolute paths to zlib include directories (list)
```

(continues on next page)

(continued from previous page)

```
        self.deps_cpp_info["zlib"].include_paths

        # Get the sharedlinkflags property from OpenSSL package
        self.deps_cpp_info["openssl"].sharedlinkflags
```

Note: If using the experimental feature *with different context for host and build*, this attribute will contain only information from packages in the *host* context.

env_info

This attribute is only defined inside `package_info()` method, being None elsewhere, so please use it only inside this method.

The `self.env_info` object can be filled with the environment variables to be declared in the packages reusing the recipe.

See also:

Read *package_info() method docs* for more info.

deps_env_info

You can access to the declared environment variables of the requirements of the recipe.

Note: The environment variables declared in the requirements of a recipe are automatically applied and it can be accessed with the python `os.environ` dictionary. Nevertheless if you want to access to the variable declared by some specific requirement you can use the `self.deps_env_info` object.

```
import os

class RecipeConan(ConanFile):
    ...
    requires = "package1/1.0@conan/stable", "package2/1.2@conan/stable"
    ...

    def build(self):
        # Get the SOMEVAR environment variable declared in the "package1"
        self.deps_env_info["package1"].SOMEVAR

        # Access to the environment variables globally
        os.environ["SOMEVAR"]
```

Note: If using the experimental feature *with different context for host and build*, this attribute will contain only information from packages in the *build* context.

user_info

This attribute is only defined inside `package_info()` method, being None elsewhere, so please use it only inside this method.

The `self.user_info` object can be filled with any custom variable to be accessed in the packages reusing the recipe.

See also:

Read *package_info() method docs* for more info.

deps_user_info

You can access the declared `user_info.XXX` variables of the requirements through the `self.deps_user_info` object like this:

```
import os

class RecipeConan(ConanFile):
    ...
    requires = "package1/1.0@conan/stable"
    ...

    def build(self):
        self.deps_user_info["package1"].SOMEVAR
```

Note: If using the experimental feature *with different context for host and build*, this attribute will contain only information from packages in the *host* context. Use *user_info_build* to access information from packages that belong to *build* context.

user_info_build

Warning: This section refers to the **experimental feature** that is activated when using `--profile:build` and `--profile:host` in the command-line. It is currently under development, features can be added or removed in the following versions.

This attribute offers the information declared in the `user_info.XXXX` variables of the requirements that belong to the *build* context, it is available only if Conan is invoked with two profiles (see *this section* to know more about this feature.

```
import os

class RecipeConan(ConanFile):
    ...
    tool_requires = "tool/1.0"
    ...

    def build(self):
        self.user_info_build["tool"].SOMEVAR
```

info

Object used to control the unique ID for a package. Check the *package_id()* to see the details of the `self.info` object.

apply_env

When `True` (Default), the values from `self.deps_env_info` (corresponding to the declared `env_info` in the `requires` and `tool_requires`) will be automatically applied to the `os.environ`.

Disable it setting `apply_env` to False if you want to control by yourself the environment variables applied to your recipes.

You can apply manually the environment variables from the requires and tool_requires:

```
import os
from conans import tools

class RecipeConan(ConanFile):
    apply_env = False

    def build(self):
        with tools.environment_append(self.env):
            # The same if we specified apply_env = True
            pass
```

in_local_cache

A boolean attribute useful for conditional logic to apply in user folders local commands. It will return *True* if the conanfile resides in the local cache (we are installing the package) and *False* if we are running the conanfile in a user folder (local Conan commands).

```
import os

class RecipeConan(ConanFile):
    ...

    def build(self):
        if self.in_local_cache:
            # we are installing the package
        else:
            # we are building the package in a local directory
```

develop

A boolean attribute useful for conditional logic. It will be `True` if the package is created with **`conan create`**, or if the *conanfile.py* is in user space:

```
class RecipeConan(ConanFile):

    def build(self):
        if self.develop:
            self.output.info("Develop mode")
```

It can be used for conditional logic in other methods too, like `requirements()`, `package()`, etc.

This recipe will output "Develop mode" if:

```
$ conan create . user/testing
# or
$ mkdir build && cd build && conan install ..
$ conan build ..
```

But it will not output that when it is a transitive requirement or installed with **`conan install`**.

keep_imports

Just before the `build()` method is executed, if the conanfile has an `imports()` method, it is executed into the build folder, to copy binaries from dependencies that might be necessary for the `build()` method to work. After the method finishes, those copied (imported) files are removed, so they are not later unnecessarily repackaged.

This behavior can be avoided declaring the `keep_imports=True` attribute. This can be useful, for example to *repackage artifacts*

scm

> **Warning:** This is an **experimental** feature subject to breaking changes in future releases. Although this is an experimental feature, the use of the feature using `scm_to_conandata` is considered stable.

Used to clone/checkout a repository. It is a dictionary with the following possible values:

```
from conans import ConanFile, CMake, tools

class HelloConan(ConanFile):
     scm = {
        "type": "git",
        "subfolder": "hello",
        "url": "https://github.com/conan-io/hello.git",
        "revision": "master"
     }
    ...
```

- **type** (Required): Currently only `git` and `svn` are supported. Others can be added eventually.
- **url** (Required): URL of the remote or `auto` to capture the remote from the local working copy (credentials will be removed from it). When type is `svn` it can contain the peg_revision.
- **revision** (Required): id of the revision or `auto` to capture the current working copy one. When type is `git`, it can also be the branch name or a tag.
- **subfolder** (Optional, Defaulted to `.`): A subfolder where the repository will be cloned.
- **username** (Optional, Defaulted to `None`): When present, it will be used as the login to authenticate with the remote.
- **password** (Optional, Defaulted to `None`): When present, it will be used as the password to authenticate with the remote.
- **verify_ssl** (Optional, Defaulted to `True`): Verify SSL certificate of the specified **url**.

- **shallow** (Optional, Defaulted to `True`): Use shallow clone for Git repositories.
- **submodule (Optional, Defaulted to `None`):**
  - `shallow`: Will sync the git submodules using `submodule sync`
  - `recursive`: Will sync the git submodules using `submodule sync --recursive`

Attributes `type`, `url` and `revision` are required to upload the recipe to a remote server.

SCM attributes are evaluated in the working directory where the *conanfile.py* is located before exporting it to the Conan cache, so these values can be returned from arbitrary functions that depend on the local directory. Nevertheless, all the other code in the recipe must be able to run in the export folder inside the cache, where it has access only to the files exported (see attribute *exports* and *conandata.yml*) and to any other functionality from a *python_requires* package.

Warning: By default, in Conan v1.x the information after evaluating the attribute `scm` will be stored in the *conanfile.py* file (the recipe will be modified when exported to the Conan cache) and any value will be written in plain text (watch out about passwords). However, you can activate the *scm_to_conandata* config option, the *conanfile.py* won't be modified (data is stored in a different file) and the fields `username` and `password` won't be stored, so these one will be computed each time the recipe is loaded.

Note: In case of git, by default conan will try to perform shallow clone of the repository, and will fallback to the full clone in case shallow fails (e.g. not supported by the server).

To know more about the usage of `scm` check:

- *Creating packages/Recipe and sources in a different repo*
- *Creating packages/Recipe and sources in the same repo*

revision_mode

Warning: This attribute is part of the *package revisions* feature, so it is also an **experimental** feature subject to breaking changes in future releases.

This attribute allow each recipe to declare how the revision for the recipe itself should be computed. It can take two different values:

- `"hash"` (by default): Conan will use the checksum hash of the recipe manifest to compute the revision for the recipe.
- `"scm"`: the commit ID will be used as the recipe revision if it belongs to a known repository system (Git or SVN). If there is no repository it will raise an error.

python_requires (legacy)

Warning: This attribute has been superseded by the new *Python requires*. Even if this is an **experimental** feature subject to breaking changes in future releases, this legacy `python_requires` syntax has not been removed yet, but it will be removed in Conan 2.0.

Python requires are associated with the `ConanFile` declared in the recipe file, data from those imported recipes is accessible using the `python_requires` attribute in the recipe itself. This attribute is a dictionary where the key is the name of the *python requires* reference and the value is a dictionary with the following information:

- `ref`: full reference of the python requires.
- `exports_folder`: directory in the cache where the exported files are located.
- `exports_sources_folder`: directory in the cache where the files exported using the `exports_sources` attribute of the python requires recipe are located.

You can use this information to copy files accompanying a python requires to the consumer workspace.:

```
from conans import ConanFile

class PyReq(ConanFile):
    name = "pyreq"
    exports_sources = "CMakeLists.txt"

    def source(self):
        pyreq = self.python_requires['pyreq']
        path = os.path.join(pyreq.exports_sources_folder, "CMakeLists.txt")
        shutil.copy(src=path, dst=self.source_folder)
```

python_requires

Warning: This is an **experimental** feature subject to breaking changes in future releases.

This class attribute allows to define a dependency to another Conan recipe and reuse its code. Its basic syntax is:

```
from conans import ConanFile

class Pkg(ConanFile):
    python_requires = "pyreq/0.1@user/channel"  # recipe to reuse code from

    def build(self):
        self.python_requires["pyreq"].module # access to the whole conanfile.py module
        self.python_requires["pyreq"].module.myvar  # access to a variable
        self.python_requires["pyreq"].module.myfunct()  # access to a global function
        self.python_requires["pyreq"].path # access to the folder where the reused file␣
↪is
```

Read more about this attribute in *Python requires*

python_requires_extend

Warning: This is an **experimental** feature subject to breaking changes in future releases.

This class attribute defines one or more classes that will be injected in runtime as base classes of the recipe class. Syntax for each of these classes should be a string like `pyreq.MyConanfileBase` where the `pyreq` is the name of a `python_requires` and `MyConanfileBase` is the name of the class to use.

```
from conans import ConanFile

class Pkg(ConanFile):
    python_requires = "pyreq/0.1@user/channel", "utils/0.1@user/channel"
    python_requires_extend = "pyreq.MyConanfileBase", "utils.UtilsBase"  # class/es to␣
↪inject
```

Read more about this attribute in *Python requires*

conan_data

This attribute is a dictionary with the keys and values provided in a *conandata.yml* file format placed next to the *conanfile.py*. This YAML file is automatically exported with the recipe and automatically loaded with it too.

You can declare information in the *conandata.yml* file and then access it inside any of the methods of the recipe. For example, a *conandata.yml* with information about sources that looks like this:

```
sources:
  "1.1.0":
    url: "https://www.url.org/source/mylib-1.0.0.tar.gz"
    sha256: "8c48baf3babe0d505d16cfc0cf272589c66d3624264098213db0fb00034728e9"
  "1.1.1":
    url: "https://www.url.org/source/mylib-1.0.1.tar.gz"
    sha256: "15b6393c20030aab02c8e2fe0243cb1d1d18062f6c095d67bca91871dc7f324a"
```

```
def source(self):
    tools.get(**self.conan_data["sources"][self.version])
```

deprecated

Warning: This is an **experimental** feature subject to breaking changes in future releases.

Available since: 1.28.0

This attribute declares that the recipe is deprecated, causing a user-friendly warning message to be emitted whenever it is used. For example, the following code:

```
from conans import ConanFile

class Pkg(ConanFile):
    name = "cpp-taskflow"
```

(continues on next page)

(continued from previous page)

```
    version = "1.0"
    deprecated = True
```

may emit a warning like:

```
cpp-taskflow/1.0: WARN: Recipe 'cpp-taskflow/1.0' is deprecated. Please, consider␣
↪changing your requirements.
```

Optionally, the attribute may specify the name of the suggested replacement:

```
from conans import ConanFile

class Pkg(ConanFile):
    name = "cpp-taskflow"
    version = "1.0"
    deprecated = "taskflow"
```

This will emit a warning like:

```
cpp-taskflow/1.0: WARN: Recipe 'cpp-taskflow/1.0' is deprecated in favor of 'taskflow'.␣
↪Please, consider changing your requirements.
```

If the value of the attribute evaluates to `False`, no warning is printed.

provides

> **Warning:** This is an **experimental** feature subject to breaking changes in future releases.

Available since: 1.28.0

This attribute declares that the recipe provides the same functionality as other recipe(s). The attribute is usually needed if two or more libraries implement the same API to prevent link-time and run-time conflicts (ODR violations). One typical situation is forked libraries.

Some examples are:

- LibreSSL, BoringSSL and OpenSSL
- libav and ffmpeg
- MariaDB client and MySQL client

If Conan encounters two or more libraries providing the same functionality within a single graph, it raises an error:

```
At least two recipes provides the same functionality:
 - 'libjpeg' provided by 'libjpeg/9d', 'libjpeg-turbo/2.0.5'
```

The attribute value should be a string with a recipe name or a tuple of such recipe names.

For example, to declare that `libjpeg-turbo` recipe offers the same functionality as `libjpeg` recipe, the following code could be used:

```
from conans import ConanFile

class LibJpegTurbo(ConanFile):
    name = "libjpeg-turbo"
    version = "1.0"
    provides = "libjpeg"
```

To declare that a recipe provides the functionality of several different recipes at the same time, the following code could be used:

```
from conans import ConanFile

class OpenBLAS(ConanFile):
    name = "openblas"
    version = "1.0"
    provides = "cblas", "lapack"
```

If the attribute is omitted, the value of the attribute is assumed to be equal to the current package name. Thus, it's redundant for `libjpeg` recipe to declare that it provides `libjpeg`, it's already implicitly assumed by Conan.

conf

This is an **experimental** feature introduced in Conan 1.47.

In the `self.conf` attribute we can find all the conf entries declared in the *[conf] section of the profiles* in addition of the declared *self.conf_info* entries from the first level tool requirements. The profile entries have priority.

This is an example of a recipe for a tool called `my_android_ndk` with the following recipe:

```
from conan import ConanFile

class MyAndroidNDKRecipe(ConanFile):

  name="my_android_ndk"
  version="1.0"

  def package_info(self):
      self.conf_info.define("tools.android:ndk_path", "bar")
```

If we require that tool:

```
from conan import ConanFile

class MyConsumer(ConanFile):

  tool_requires = "my_android_ndk/1.0"

  def generate(self):
      self.output.info("NDK host: %s" % self.conf.get("tools.android:ndk_path"))
      self.output.info("Custom var1: %s" % self.conf.get("user.custom.var1"))
```

And we install it applying this profile:

Listing 9: myprofile

```
include(default)
[conf]
tools.android:ndk_path=foo
user.custom.var1=hello
```

```
$ conan install . --profile myprofile
...
conanfile.py: Applying build-requirement: my_android_ndk/1.0
conanfile.py: Generator txt created conanbuildinfo.txt
conanfile.py: Calling generate()
conanfile.py: NDK host: foo
conanfile.py: Custom var1: hello
...
```

Without the profile:

```
$ conan install .
...
conanfile.py: Applying build-requirement: my_android_ndk/1.0
conanfile.py: Generator txt created conanbuildinfo.txt
conanfile.py: Calling generate()
conanfile.py: NDK host: bar
conanfile.py: Custom var1: None
```

win_bash

This is an **experimental** feature introduced in Conan 1.39.

When `True` it enables the new run in a subsystem bash in Windows mechanism. *Read more here*.

```
from conans import ConanFile

class FooRecipe(ConanFile):
    ...
    win_bash = True
```

test_type

Warning: Test type is an **experimental** feature subject to breaking changes in future releases.

Available since: 1.44.0

This attribute allows testing requirements and build requiments explicitly on test package. In Conan 2.0 the `test_type` attribute will be ignored, the behavior will be always explicit, so declaring `test_type="explicit"` will make the test recipe compatible with Conan 2.0. The possible values are:

- `requires` (default): The package being tested will be consumed as a regular requirement automatically.

- `build_requires`: The package being tested will be consumed as a build requirement automatically. It can be combined with `requires` to be required in both ways.
- `explicit`: The test package will not solve its dependencies automatically, you need to declare it explicitly using the reference at `self.tested_reference_str`. This will be the only behavior for Conan 2.0. The additional values `requires` and `build_requires` (if specified) will be ignored.

To solve build requirements and requirements automatically as regularly on Conan 1.0

```
from conans import ConanFile, CMake
import os

class TestPackage(ConanFile):
    test_type = "build_requires", "requires"

    def build(self):
        cmake = CMake(self)
        cmake.configure()
        cmake.build()

    def test(self):
        bin_path = os.path.join("bin", "test_package")
        self.run(bin_path, run_environment=True)
```

To explicitly declare the required dependencies as required on Conan 2.0:

```
from conans import ConanFile, CMake
import os

class TestPackage(ConanFile):
    test_type = "explicit"

    def requirements(self):
        self.requires(self.tested_reference_str)
        # and, or
        self.build_requires(self.tested_reference_str)

    def build(self):
        cmake = CMake(self)
        cmake.configure()
        cmake.build()

    def test(self):
        bin_path = os.path.join("bin", "test_package")
        self.run(bin_path, run_environment=True)
```

For more information see *explicit test package requirement* and *testing tool requirements*.

18.3.2 Methods

source()

Method used to retrieve the source code from any other external origin like github using `$ git clone` or just a regular download.

For example, "exporting" the source code files, together with the *conanfile.py* file, can be handy if the source code is not under version control. But if the source code is available in a repository, you can directly get it from there:

```
from conans import ConanFile

class HelloConan(ConanFile):
    name = "hello"
    version = "0.1"
    settings = "os", "compiler", "build_type", "arch"

    def source(self):
        self.run("git clone https://github.com/conan-io/hello.git")
        # You can also change branch, commit or whatever
        # self.run("cd hello && git checkout 2fe5...")
        #
        # Or using the Git class:
        # git = tools.Git(folder="hello")
        # git.clone("https://github.com/conan-io/hello.git")
```

The current working directory where the `source()` method runs is the `self.source_folder`. Note, however, that this folder can be different if the recipe defines the `layout()` method and specifies a `self.folders.source = "src"`. In that case, the `self.source_folder` and the current working directory will be the composition of the base folder (typically where the recipe is) and the user specified `"src"` subfolder.

This will work, as long as git is in your current path (so in Win you probably want to run things in msysgit, cmder, etc). You can also use another VCS or direct download/unzip. For that purpose, we have provided some helpers, but you can use your own code or origin as well. This is a snippet of the conanfile of the Poco library:

```
from conans import ConanFile
from conans.tools import download, unzip, check_md5, check_sha1, check_sha256
import os
import shutil

class PocoConan(ConanFile):
    name = "poco"
    version = "1.6.0"

    def source(self):
        zip_name = "poco-1.6.0-release.zip"
        download("https://github.com/pocoproject/poco/archive/poco-1.6.0-release.zip",␣
↪zip_name)
        # check_md5(zip_name, "51e11f2c02a36689d6ed655b6fff9ec9")
        # check_sha1(zip_name, "8d87812ce591ced8ce3a022beec1df1c8b2fac87")
        # check_sha256(zip_name,
↪"653f983c30974d292de58444626884bee84a2731989ff5a336b93a0fef168d79")
        unzip(zip_name)
        shutil.move("poco-poco-1.6.0-release", "poco")
        os.unlink(zip_name)
```

The download, unzip utilities can be imported from conan, but you can also use your own code here to retrieve source code from any origin. You can even create packages for pre-compiled libraries you already have, even if you don't have the source code. You can download the binaries, skip the `build()` method and define your `package()` and `package_info()` accordingly.

You can also use `check_md5()`, `check_sha1()` and `check_sha256()` from the *tools* module to verify that a package is downloaded correctly.

Note: It is very important to recall that the `source()` method will be executed just once, and the source code will be shared for all the package builds. So it is not a good idea to conditionally use settings or options to make changes or patches on the source code. Maybe the only setting that makes sense is the OS `self.settings.os`, if not doing cross-building, for example to retrieve different sources:

```
def source(self):
    if platform.system() == "Windows":
        # download some Win source zip
    else:
        # download sources from Nix systems in a tgz
```

If you need to patch the source code or build scripts differently for different variants of your packages, you can do it in the `build()` method, which uses a different folder and source code copy for each variant.

```
def build(self):
    tools.patch(patch_file="0001-fix.patch")
```

---

build()

This method is used to build the source code of the recipe using the desired commands. You can use your command line tools to invoke your build system or any of the build helpers provided with Conan.

```
def build(self):
    cmake = CMake(self)
    self.run("cmake . %s" % (cmake.command_line))
    self.run("cmake --build . %s" % cmake.build_config)
```

Build helpers

You can use these classes to prepare your build system's command invocation:

- **CMake**: Prepares the invocation of cmake command with your settings.
- **AutoToolsBuildEnvironment**: If you are using configure/Makefile to build your project you can use this helper. Read more: *Building with Autotools*.
- **MSBuild**: If you are using Visual Studio compiler directly to build your project you can use this helper *MSBuild()*. For lower level control, the **VisualStudioBuildEnvironment** can also be used: *VisualStudioBuildEnvironment*.

(Unit) Testing your library

We have seen how to run package tests with conan, but what if we want to run full unit tests on our library before packaging, so that they are run for every build configuration? Nothing special is required here. We can just launch the tests from the last command in our `build()` method:

```
def build(self):
    cmake = CMake(self)
    cmake.configure()
    cmake.build()
    # here you can run CTest, launch your binaries, etc
    cmake.test()
```

package()

The actual creation of the package, once that it is built, is done in the `package()` method. Using the `self.copy()` method, artifacts are copied from the build folder to the package folder.

The syntax of `self.copy` inside `package()` is as follows:

```
self.copy(pattern, dst="", src="", keep_path=True, symlinks=None, excludes=None, ignore_
↪case=True)
```

Returns: A list with absolute paths of the files copied in the destination folder.

Parameters:

- **pattern** (Required): A pattern following fnmatch syntax of the files you want to copy, from the build to the package folders. Typically something like `*.lib` or `*.h`.
- **src** (Optional, Defaulted to `""`): The folder where you want to search the files in the build folder. If you know that your libraries when you build your package will be in *build/lib*, you will typically use `build/lib` in this parameter. Leaving it empty means the root build folder in local cache.
- **dst** (Optional, Defaulted to `""`): Destination folder in the package. They will typically be `include` for headers, `lib` for libraries and so on, though you can use any convention you like. Leaving it empty means the root package folder in local cache.
- **keep_path** (Optional, Defaulted to `True`): Means if you want to keep the relative path when you copy the files from the **src** folder to the **dst** one. Typically headers are packaged with relative path.
- **symlinks** (Optional, Defaulted to `None`): Set it to True to activate symlink copying, like typical lib.so->lib.so.9.
- **excludes** (Optional, Defaulted to `None`): Single pattern or a tuple of patterns to be excluded from the copy. If a file matches both the include and the exclude pattern, it will be excluded.
- **ignore_case** (Optional, Defaulted to `True`): If enabled, it will do a case-insensitive pattern matching.

For example:

```
self.copy("*.h", "include", "build/include") #keep_path default is True
```

The final path in the package will be: `include/mylib/path/header.h`, and as the *include* is usually added to the path, the includes will be in the form: `#include "mylib/path/header.h"` which is something desired.

`keep_path=False` is something typically desired for libraries, both static and dynamic. Some compilers as MSVC, put them in paths as *Debug/x64/MyLib/Mylib.lib*. Using this option, we could write:

```
self.copy("*.lib", "lib", "", keep_path=False)
```

And it will copy the lib to the package folder *lib/Mylib.lib*, which can be linked easily.

Note: If you are using CMake and you have an install target defined in your CMakeLists.txt, you might be able to reuse it for this `package()` method. Please check *How to reuse cmake install for package() method*.

This method copies files from build/source folder to the package folder depending on two situations:

- **Build folder and source folder are the same**: Normally during **`conan create`** source folder content is copied to the build folder. In this situation `src` parameter of `self.copy()` will be relative to the build folder in the local cache.
- **Build folder is different from source folder**: When *developing a package recipe* and source and build folder are different (**`conan package . --source-folder=source --build-folder=build`**) or when *no_copy_source* is defined, every `self.copy()` is internally called twice: One will copy from the source folder (`src` parameter of `self.copy()` will point to the source folder), and the other will copy from the build folder (`src` parameter of `self.copy()` will point to the build folder).

package_info()

cpp_info

Each package has to specify certain build information for its consumers. This can be done in the `cpp_info` attribute within the `package_info()` method.

The *cpp_info* attribute has the following properties you can assign/append to:

```
self.cpp_info.names["generator_name"] = "<PKG_NAME>"
self.cpp_info.includedirs = ['include']  # Ordered list of include paths
self.cpp_info.libs = []  # The libs to link against
self.cpp_info.system_libs = []  # System libs to link against
self.cpp_info.libdirs = ['lib']  # Directories where libraries can be found
self.cpp_info.resdirs = ['res']  # Directories where resources, data, etc. can be found
self.cpp_info.bindirs = ['bin']  # Directories where executables and shared libs can be␣
↪found
self.cpp_info.srcdirs = []  # Directories where sources can be found (debugging, reusing␣
↪sources)
self.cpp_info.build_modules = {}  # Build system utility module files
self.cpp_info.defines = []  # preprocessor definitions
self.cpp_info.cflags = []  # pure C flags
self.cpp_info.cxxflags = []  # C++ compilation flags
self.cpp_info.sharedlinkflags = []  # linker flags
self.cpp_info.exelinkflags = []  # linker flags
self.cpp_info.components  # Dictionary with the different components a package may have
self.cpp_info.requires = None  # List of components from requirements
```

- **names**: Alternative name(s) for the package to be used by generators.
- **includedirs**: List of relative paths (starting from the package root) of directories where headers can be found. By default it is initialized to `['include']`, and it is rarely changed.
- **libs**: Ordered list of libs the client should link against. Empty by default, it is common that different configurations produce different library names. For example:

```
def package_info(self):
    if not self.settings.os == "Windows":
        self.cpp_info.libs = ["libzmq-static.a"] if self.options.static else [
↪"libzmq.so"]
    else:
        ...
```

- **libdirs**: List of relative paths (starting from the package root) of directories in which to find library object binaries (*.lib, *.a, *.so, *.dylib). By default it is initialized to `['lib']`, and it is rarely changed.
- **resdirs**: List of relative paths (starting from the package root) of directories in which to find resource files (images, xml, etc). By default it is initialized to `['res']`, and it is rarely changed.
- **bindirs**: List of relative paths (starting from the package root) of directories in which to find library runtime binaries (like Windows .dlls). By default it is initialized to `['bin']`, and it is rarely changed.
- **srcdirs**: List of relative paths (starting from the package root) of directories in which to find sources (like .c, .cpp). By default it is empty. It might be used to store sources (for later debugging of packages, or to reuse those sources building them in other packages too).
- **build_modules**: Dictionary of lists per generator containing relative paths to build system related utility module files created by the package. Used by CMake generators to include *.cmake* files with functions for consumers. e.g: `self.cpp_info.build_modules["cmake_find_package"].append("cmake/myfunctions.cmake")`. Those files will be included automatically in *cmake/cmake_multi* generators when using *conan_basic_setup()* and will be automatically added in *cmake_find_package/cmake_find_package_multi* generators when *find_package()* is used.
- **defines**: Ordered list of preprocessor directives. It is common that the consumers have to specify some sort of defines in some cases, so that including the library headers matches the binaries.
- **system_libs**: Ordered list of system libs the consumer should link against. Empty by default.
- **cflags**, **cxxflags**, **sharedlinkflags**, **exelinkflags**: List of flags that the consumer should activate for proper behavior. Usage of C++11 could be configured here, for example, although it is true that the consumer may want to do some flag processing to check if different dependencies are setting incompatible flags (c++11 after c++14).

```
if self.options.static:
    if self.settings.compiler == "Visual Studio":
        self.cpp_info.libs.append("ws2_32")
    self.cpp_info.defines = ["ZMQ_STATIC"]

    if not self.settings.os == "Windows":
        self.cpp_info.cxxflags = ["-pthread"]
```

  Note that due to the way that some build systems, like CMake, manage forward and back slashes, it might be more robust passing flags for Visual Studio compiler with dash instead. Using `"/NODEFAULTLIB:MSVCRT"`, for example, might fail when using CMake targets mode, so the following is preferred and works both in the global and targets mode of CMake:

```
def package_info(self):
    self.cpp_info.exelinkflags = ["-NODEFAULTLIB:MSVCRT",
                                  "-DEFAULTLIB:LIBCMT"]
```

- **components**: **[Experimental]** Dictionary with names as keys and a component object as value to model the different components a package may have: libraries, executables... Read more about this feature at *Using Components*.

- **requires**: **[Experimental]** List of components from the requirements this package (and its consumers) should link with. It will be used by generators that add support for components features (*Using Components*).

If your recipe has requirements, you can access to the information stored in the `cpp_info` of your requirements using the `deps_cpp_info` object:

```
class OtherConan(ConanFile):
    name = "OtherLib"
    version = "1.0"
    requires = "mylib/1.6.0@conan/stable"

    def build(self):
        self.output.warn(self.deps_cpp_info["mylib"].libdirs)
```

Note: Please take into account that defining `self.cpp_info.bindirs` directories, does not have any effect on system paths, PATH environment variable, nor will be directly accessible by consumers. `self.cpp_info` information is translated to build-systems information via generators, for example for CMake, it will be a variable in `conanbuildinfo.cmake`. If you want a package to make accessible its executables to its consumers, you have to specify it with `self.env_info` as described in *env_info*.

env_info

Each package can also define some environment variables that the package needs to be reused. It's specially useful for *installer packages*, to set the path with the "bin" folder of the packaged application. This can be done in the `env_info` attribute within the `package_info()` method.

```
self.env_info.path.append("ANOTHER VALUE") # Append "ANOTHER VALUE" to the path variable
self.env_info.othervar = "OTHER VALUE" # Assign "OTHER VALUE" to the othervar variable
self.env_info.thirdvar.append("some value") # Every variable can be set or appended a␣
↪new value
```

One of the most typical usages for the PATH environment variable, would be to add the current binary package directories to the path, so consumers can use those executables easily:

```
# assuming the binaries are in the "bin" subfolder
self.env_info.PATH.append(os.path.join(self.package_folder, "bin"))
```

The *virtualenv* generator will use the `self.env_info` variables to prepare a script to activate/deactivate a virtual environment. However, this could be directly done using the *virtualrunenv* generator.

They will be automatically applied before calling the consumer *conanfile.py* methods `source()`, `build()`, `package()` and `imports()`.

If your recipe has requirements, you can access to your requirements `env_info` as well using the `deps_env_info` object.

```
class OtherConan(ConanFile):
    name = "OtherLib"
    version = "1.0"
    requires = "mylib/1.6.0@conan/stable"

    def build(self):
        self.output.warn(self.deps_env_info["mylib"].othervar)
```

user_info

If you need to declare custom variables not related with C/C++ (`cpp_info`) and the variables are not environment variables (`env_info`), you can use the `self.user_info` object.

Currently only the `cmake`, `cmake_multi` and `txt` generators supports `user_info` variables.

```
class MyLibConan(ConanFile):
    name = "mylib"
    version = "1.6.0"

    # ...

    def package_info(self):
        self.user_info.var1 = 2
```

For the example above, in the `cmake` and `cmake_multi` generators, a variable `CONAN_USER_MYLIB_var1` will be declared. If your recipe has requirements, you can access to your requirements `user_info` using the `deps_user_info` object.

```
class OtherConan(ConanFile):
    name = "otherlib"
    version = "1.0"
    requires = "mylib/1.6.0@conan/stable"

    def build(self):
        self.out.warn(self.deps_user_info["mylib"].var1)
```

Important: Both `env_info` and `user_info` objects store information in a "key <-> value" form and the values are always considered strings. This is done for serialization purposes to *conanbuildinfo.txt* files and to avoid the deserialization of complex structures. It is up to the consumer to convert the string to the expected type:

```
# In a dependency
self.user_info.jars="jar1.jar, jar2.jar, jar3.jar"  # Use a string, not a list
...

# In the dependent conanfile
jars = self.deps_user_info["pkg"].jars
jar_list = jars.replace(" ", "").split(",")
```

---

set_name(), set_version()

Dynamically define `name` and `version` attributes in the recipe with these methods. The following example defines the package name reading it from a *name.txt* file and the version from the branch and commit of the recipe's repository.

These functions are executed after assigning the values of the `name` and `version` if they are provided from the command line.

```
from conans import ConanFile, tools

class HelloConan(ConanFile):
```

(continues on next page)

(continued from previous page)

```
    def set_name(self):
        # Read the value from 'name.txt' if it is not provided in the command line
        self.name = self.name or tools.load("name.txt")

    def set_version(self):
        git = tools.Git()
        self.version = "%s_%s" % (git.get_branch(), git.get_revision())
```

The `set_name()` and `set_version()` methods should respectively set the `self.name` and `self.version` attributes. These methods are only executed when the recipe is in a user folder (**`export`**, **`create`** and **`install <path>`** commands).

The above example uses the current working directory as the one to resolve the relative "name.txt" path and the git repository. That means that the "name.txt" should exist in the directory where conan was launched. To define a relative path to the *conanfile.py*, irrespective of the current working directory it is necessary to do:

```
import os
from conans import ConanFile, tools

class HelloConan(ConanFile):
    def set_name(self):
        f = os.path.join(self.recipe_folder, "name.txt")
        self.name = tools.load(f)

    def set_version(self):
        git = tools.Git(folder=self.recipe_folder)
        self.version = "%s_%s" % (git.get_branch(), git.get_revision())
```

Warning: The `set_name()` and `set_version()` methods are alternatives to the `name` and `version` attributes. It is not advised or supported to define both a `name` attribute and a `set_name()` method. Likewise, it is not advised or supported to define both a `version` attribute and a `set_version()` method. If you define both, you may experience unexpected behavior.

See also:

See more examples *in this howto*.

configure(), config_options()

If the package options and settings are related, and you want to configure either, you can do so in the `configure()` and `config_options()` methods.

```
class MyLibConan(ConanFile):
    name = "MyLib"
    version = "2.5"
    settings = "os", "compiler", "build_type", "arch"
    options = {"static": [True, False],
               "header_only": [True False]}

    def configure(self):
        # If header only, the compiler, etc, does not affect the package!
```

(continues on next page)

(continued from previous page)

```
        if self.options.header_only:
            self.settings.clear()
            del self.options.static
```

The package has 2 options set, to be compiled as a static (as opposed to shared) library, and also not to involve any builds, because header-only libraries will be used. In this case, the settings that would affect a normal build, and even the other option (static vs shared) do not make sense, so we just clear them. That means, if someone consumes MyLib with the `header_only=True` option, the package downloaded and used will be the same, irrespective of the OS, compiler or architecture the consumer is building with.

You can also restrict the settings used deleting any specific one. For example, it is quite common for C libraries to delete the `compiler.libcxx` and `compiler.cppstd` as your library does not depend on any C++ standard library:

```
def configure(self):
    del self.settings.compiler.libcxx
    del self.settings.compiler.cppstd
```

The most typical usage would be the one with `configure()` while `config_options()` should be used more sparingly. `config_options()` is used to configure or constraint the available options in a package, **before** they are given a value. So when a value is tried to be assigned it will raise an error. For example, let's suppose that a certain package library cannot be built as shared library in Windows, it can be done:

```
def config_options(self):
    if self.settings.os == "Windows":
        del self.options.shared
```

This will be executed before the actual assignment of `options` (then, such `options` values cannot be used inside this function), so the command **`conan install -o pkg:shared=True`** will raise an exception in Windows saying that `shared` is not an option for such package.

These methods can also be used to assign values to options as seen in *options*. Values assigned in the `configure()` method cannot be overriden, while values assigned in `config_options()` can.

Invalid configuration

Conan allows the recipe creator to declare invalid configurations, those that are known not to work with the library being packaged. There is an especial kind of exception that can be raised from the `validate()` method to state this situation: `conans.errors.ConanInvalidConfiguration`. Here it is an example of a recipe for a library that doesn't support Windows operating system:

```
def validate(self):
    if self.settings.os != "Windows":
        raise ConanInvalidConfiguration("Library MyLib is only supported for Windows")
```

This exception will be propagated and Conan application will finish with a *special return code*.

Note: For managing invalid configurations, please check the new experimental `validate()` method (*validate()*).

validate()

Warning: This is an **experimental** feature subject to breaking changes in future releases.

Available since: 1.32.0

The `validate()` method can be used to mark a binary as "impossible" or invalid for a given configuration. For example, if a given library does not build or work at all in Windows it can be defined as:

```
from conans import ConanFile
from conans.errors import ConanInvalidConfiguration

class Pkg(ConanFile):
    settings = "os"

    def validate(self):
        if self.settings.os == "Windows":
            raise ConanInvalidConfiguration("Windows not supported")
```

If you try to use, consume or build such a package, it will raise an error, returning exit code *exit code*:

```
$ conan create . pkg/0.1@ -s os=Windows
...
Packages
    pkg/0.1:INVALID - Invalid
...
> ERROR: There are invalid packages (packages that cannot exist for this configuration):
> pkg/0.1: Invalid ID: Windows not supported
```

A major difference with `configure()` is that this information can be queried with the `conan info` command, for example this is possible without getting an error:

```
$ conan export . test/0.1@user/testing
...
> test/0.1@user/testing: Exported revision: ...

$ conan info test/0.1@user/testing
>test/0.1@user/testing
    ID: INVALID
    BuildID: None
    Remote: None
    ...
```

Another important difference with the `configure()` method, is that `validate()` is evaluated after the graph has been computed and the information has been propagated downstream. So the values used in `validate()` are guaranteed to be final real values, while values at `configure()` time are not. This might be important, for example when checking values of options of dependencies:

```
from conans import ConanFile
from conans.errors import ConanInvalidConfiguration

class Pkg(ConanFile):
    requires = "dep/0.1"
```

(continues on next page)

(continued from previous page)

```
    def validate(self):
        if self.options["dep"].myoption == 2:
            raise ConanInvalidConfiguration("Option 2 of 'dep' not supported")
```

If a package uses `compatible_packages` feature, it should not add to those compatible packages configurations that will not be valid, for example:

```
from conans import ConanFile
from conans.errors import ConanInvalidConfiguration

class Pkg(ConanFile):
    settings = "os", "build_type"

    def validate(self):
        if self.settings.os == "Windows":
            raise ConanInvalidConfiguration("Windows not supported")

    def package_id(self):
        if self.settings.build_type == "Debug" and self.settings.os != "Windows":
            compatible_pkg = self.info.clone()
            compatible_pkg.settings.build_type = "Release"
            self.compatible_packages.append(compatible_pkg)
```

Note the `self.settings.os != "Windows"` in the `package_id()`. If this is not provided, the `validate()` might still work and raise an error, but in the best case it will be wasted resources (compatible packages do more API calls to check them), so it is strongly recommended to properly define the `package_id()` method to no include incompatible configurations.

requirements()

Besides the `requires` field, more advanced requirement logic can be defined in the `requirements()` optional method, using for example values from the package `settings` or `options`:

```
def requirements(self):
    if self.options.myoption:
        self.requires("zlib/1.2@drl/testing")
    else:
        self.requires("opencv/2.2@drl/stable")
```

This is a powerful mechanism for handling **conditional dependencies**.

When you are inside the method, each call to `self.requires()` will add the corresponding requirement to the current list of requirements. It also has optional parameters that allow defining the special cases, as is shown below:

```
def requirements(self):
    self.requires("zlib/1.2@drl/testing", private=True, override=False)
```

`self.requires()` parameters:

- **override** (Optional, Defaulted to `False`): True means that this is not an actual requirement, but something to be passed upstream and override possible existing values.

- **private** (Optional, Defaulted to `False`): True means that this requirement will be somewhat embedded, and totally hidden. It might be necessary in some extreme cases, like having to use two different versions of the same library (provided that they are totally hidden in a shared library, for example), but it is mostly discouraged otherwise.

Note: To prevent accidental override of transitive dependencies, check the config variable *general.error_on_override* or the environment variable *CONAN_ERROR_ON_OVERRIDE*.

---

build_requirements()

The requires specified in this method are only installed and used when the package is built from sources. If there is an existing pre-compiled binary, then the tool requirements for this package will not be retrieved.

This method is useful for defining conditional tool requirements, for example:

```
class MyPkg(ConanFile):

    def build_requirements(self):
        if self.settings.os == "Windows":
            self.tool_requires("tool_win/0.1@user/stable")
```

See also:

Tool requirements

system_requirements()

It is possible to install system-wide packages from Conan. Just add a `system_requirements()` method to your conanfile and specify what you need there.

For a special use case you can use also `conans.tools.os_info` object to detect the operating system, version and distribution (Linux):

- `os_info.is_linux`: True if Linux.
- `os_info.is_windows`: True if Windows.
- `os_info.is_macos`: True if macOS.
- `os_info.is_freebsd`: True if FreeBSD.
- `os_info.is_solaris`: True if SunOS.
- `os_info.os_version`: OS version.
- `os_info.os_version_name`: Common name of the OS (Windows 7, Mountain Lion, Wheezy...).
- `os_info.linux_distro`: Linux distribution name (None if not Linux).
- `os_info.bash_path`: Returns the absolute path to a bash in the system.
- `os_info.uname(options=None)`: Runs the "uname" command and returns the output. You can pass arguments with the *options* parameter.
- `os_info.detect_windows_subsystem()`: Returns "MSYS", "MSYS2", "CYGWIN" or "WSL" if any of these Windows subsystems are detected.

Warning: The values returned from some of these variables (`linux_distro`, `os_version` and `os_version_name`) use the external dependency distro, values returned might be different from one version to another, please check their changelog for bugfixes and new features.

You can also use `SystemPackageTool` class, that will automatically invoke the right system package tool: **apt**, **yum**, **dnf**, **pkg**, **pkgutil**, **brew** and **pacman** depending on the system we are running.

```
from conans.tools import os_info, SystemPackageTool

def system_requirements(self):
    pack_name = None
    if os_info.linux_distro == "ubuntu":
        if os_info.os_version > "12":
            pack_name = "package_name_in_ubuntu_10"
        else:
            pack_name = "package_name_in_ubuntu_12"
    elif os_info.linux_distro == "fedora" or os_info.linux_distro == "centos":
        pack_name = "package_name_in_fedora_and_centos"
    elif os_info.is_macos:
        pack_name = "package_name_in_macos"
    elif os_info.is_freebsd:
        pack_name = "package_name_in_freebsd"
    elif os_info.is_solaris:
        pack_name = "package_name_in_solaris"

    if pack_name:
        installer = SystemPackageTool()
        installer.install(pack_name) # Install the package, will update the package␣
↪database if pack_name isn't already installed
```

On Windows, there is no standard package manager, however **choco** can be invoked as an optional:

```
from conans.tools import os_info, SystemPackageTool, ChocolateyTool

def system_requirements(self):
    if os_info.is_windows:
        pack_name = "package_name_in_windows"
        installer = SystemPackageTool(tool=ChocolateyTool()) # Invoke choco package␣
↪manager to install the package
        installer.install(pack_name)
```

SystemPackageTool

Warning: SystemPackageTool will dissapear in Conan 2.0, there's already a new implementation of these wrappers in *conan.tools.system.package_manager* that will be the default in Conan 2.0.

```
def SystemPackageTool(runner=None, os_info=None, tool=None, recommends=False,␣
↪output=None, conanfile=None, default_mode="enabled")
```

Available tool classes: **AptTool**, **YumTool**, **DnfTool**, **BrewTool**, **PkgTool**, **PkgUtilTool**, **ChocolateyTool**, **PacManTool**.

Methods:

- **add_repository(repository, repo_key=None)**: Add `repository` address in your current repo list.
- **update()**: Updates the system package manager database. It's called automatically from the `install()` method by default.
- **install(packages, update=True, force=False)**: Installs the `packages` (could be a list or a string). If `update` is True it will execute `update()` first if it's needed. The packages won't be installed if they are already installed at least of `force` parameter is set to True. If `packages` is a list the first available package will be picked (short-circuit like logical **or**). **Note**: This list of packages is intended for providing **alternative** names for the same package, to account for small variations of the name for the same package in different distros. To install different packages, one call to `install()` per package is necessary.
- **install_packages(packages, update=True, force=False, arch_names=None)**: Installs all `packages` (could be a list or a string). If `update` is True it will execute `update()` first if it's needed. The packages won't be installed if they are already installed at least of `force` parameter is set to True. If `packages` has a nested list or tuple, the first available package will be picked (short-circuit like logical **or**).
- **installed(package_name)**: Verify if `package_name` is actually installed. It returns `True` if it is installed, otherwise `False`.

The use of `sudo` in the internals of the `install()` and `update()` methods is controlled by the `CONAN_SYSREQUIRES_SUDO` environment variable, so if the users don't need sudo permissions, it is easy to opt-in/out.

When the environment variable `CONAN_SYSREQUIRES_SUDO` is not defined, Conan will try to use **sudo** if the following conditions are met:

- **sudo** is available in the `PATH`.
- The platform name is `posix` and the UID (user id) is not `0`

Also, when the environment variable *CONAN_SYSREQUIRES_MODE* is not defined, Conan will work as if its value was `enabled` unless you pass the `default_mode` argument to the constructor of `SystemPackageTool`. In that case, it will work as if `CONAN_SYSREQUIRES_MODE` had been defined to that value. If `CONAN_SYSREQUIRES_MODE` is defined, it will take preference and the `default_mode` parameter will not affect. This can be useful when a recipe has system requirements but we don't want to automatically install them if the user has not defined `CONAN_SYSREQUIRES_MODE` but to warn him about the missing requirements and allowing him to install them.

Conan will keep track of the execution of this method, so that it is not invoked again and again at every Conan command. The execution is done per package, since some packages of the same library might have different system dependencies. If you are sure that all your binary packages have the same system requirements, just add the following line to your method:

```
def system_requirements(self):
    self.global_system_requirements=True
    if ...
```

To install multi-arch packages it is possible passing the desired architecture manually according your package manager:

```
name = "foobar"
platforms = {"x86_64": "amd64", "x86": "i386"}
installer = SystemPackageTool(tool=AptTool())
installer.install("%s:%s" % (name, platforms[self.settings.arch]))
```

However, it requires a boilerplate which could be automatically solved by your settings in ConanFile:

```
installer = SystemPackageTool(conanfile=self)
installer.install(name)
```

The `SystemPackageTool` is adapted to support possible prefixes and suffixes, according to the instance of the package manager. It validates whether your current settings are configured for cross-building, and if so, it will update the package name to be installed according to `self.settings.arch`.

To install more than one package at once:

```
def system_requirements(self):
    packages = [("vim", "nano", "emacs"), "firefox", "chromium"]
    installer = SystemPackageTool()
    installer.install_packages(packages)
    # e.g. apt-get install -y --no-recommends vim firefox chromium
```

The `install_packages` will install the first text editor available (only one) following the tupple order, while it will install both web browsers.

imports()

Importing files copies files from the local store to your project. This feature is handy for copying shared libraries (*dylib* in Mac, *dll* in Win) to the directory of your executable, so that you don't have to mess with your PATH to run them. But there are other use cases:

- Copy an executable to your project, so that it can be easily run. A good example is the **Google's protobuf** code generator.
- Copy package data to your project, like configuration, images, sounds... A good example is the **OpenCV** demo, in which face detection XML pattern files are required.

Importing files is also very convenient in order to redistribute your application, as many times you will just have to bundle your project's bin folder.

A typical `imports()` method for shared libs could be:

```
def imports(self):
   self.copy("*.dll", "", "bin")
   self.copy("*.dylib", "", "lib")
```

The `self.copy()` method inside `imports()` supports the following arguments:

```
def copy(pattern, dst="", src="", root_package=None, folder=False, ignore_case=True,␣
↪excludes=None, keep_path=True)
```

Parameters:

- **pattern** (Required): An fnmatch file pattern of the files that should be copied.
- **dst** (Optional, Defaulted to ""): Destination local folder, with reference to current directory, to which the files will be copied.
- **src** (Optional, Defaulted to ""): Source folder in which those files will be searched. This folder will be stripped from the dst parameter. E.g., *lib/Debug/x86*. It accepts symbolic folder names like `@bindirs` and `@libdirs` which will map to the `self.cpp_info.bindirs` and `self.cpp_info.libdirs` of the source package, instead of a hardcoded name.
- **root_package** (Optional, Defaulted to *all packages in deps*): An fnmatch pattern of the package name ("OpenCV", "Boost") from which files will be copied.

- **folder** (Optional, Defaulted to `False`): If enabled, it will copy the files from the local cache to a subfolder named as the package containing the files. Useful to avoid conflicting imports of files with the same name (e.g. License).
- **ignore_case** (Optional, Defaulted to `True`): If enabled, it will do a case-insensitive pattern matching.
- **excludes** (Optional, Defaulted to `None`): Allows defining a list of patterns (even a single pattern) to be excluded from the copy, even if they match the main `pattern`.
- **keep_path** (Optional, Defaulted to `True`): Means if you want to keep the relative path when you copy the files from the **src** folder to the **dst** one. Useful to ignore (`keep_path=False`) path of *library.dll* files in the package it is imported from.

Example to collect license files from dependencies:

```
def imports(self):
    self.copy("license*", dst="licenses", folder=True, ignore_case=True)
```

If you want to be able to customize the output user directory to work with both the `cmake` and `cmake_multi` generators, then you can do:

```
def imports(self):
    dest = os.getenv("CONAN_IMPORT_PATH", "bin")
    self.copy("*.dll", dst=dest, src="bin")
    self.copy("*.dylib*", dst=dest, src="lib")
```

And then use, for example: **conan install . -e CONAN_IMPORT_PATH=Release -g cmake_multi**

To import files from packages that have different layouts, for example a package uses folder `libraries` instead of `lib`, or to import files from packages that could be in editable mode, a symbolic `src` argument can be provided:

```
def imports(self):
    self.copy("*", src="@bindirs", dst="bin")
    self.copy("*", src="@libdirs", dst="lib")
```

This will import all files from all the dependencies `self.cpp_info.bindirs` folders to the local "bin" folder, and all files from the dependencies `self.cpp_info.libdirs` folders to the local "lib" folder. This include packages that are in *editable* mode and declares `[libdirs]` and `[bindirs]` in their editable layouts.

When a conanfile recipe has an `imports()` method and it builds from sources, it will do the following:

- Before running `build()` it will execute `imports()` in the build folder, copying dependencies artifacts
- Run the `build()` method, which could use such imported binaries.
- Remove the copied (imported) artifacts after `build()` is finished.

You can use the *keep_imports* attribute to keep the imported artifacts, and maybe *repackage* them.

package_id()

Creates a unique ID for the package. Default package ID is calculated using `settings`, `options` and `requires` properties. When a package creator specifies the values for any of those properties, it is telling that any value change will require a different binary package.

However, sometimes a package creator would need to alter the default behavior, for example, to have only one binary package for several different compiler versions. In that case you can set a custom `self.info` object implementing this method and the package ID will be computed with the given information:

```
def package_id(self):
    v = Version(str(self.settings.compiler.version))
    if self.settings.compiler == "gcc" and (v >= "4.5" and v < "5.0"):
        self.info.settings.compiler.version = "GCC 4 between 4.5 and 5.0"
```

Please, check the section *Defining Package ABI Compatibility* to get more details.

self.info

This `self.info` object stores the information that will be used to compute the package ID.

This object can be manipulated to reflect the information you want in the computation of the package ID. For example, you can delete any setting or option:

```
def package_id(self):
    del self.info.settings.compiler
    del self.info.options.shared
```

self.info.header_only()

The package will always be the same, irrespective of the settings (OS, compiler or architecture), options and dependencies.

```
def package_id(self):
    self.info.header_only()
```

self.info.shared_library_package_id()

When a shared library links with a static library, the binary code of the later one is "embedded" or copied into the shared library. That means that any change in the static library basically requires a new binary re-build of the shared one to integrate those changes. Note that this doesn't happen in the static-static and shared-shared library dependencies.

Use this `shared_library_package_id()` helper in the `package_id()` method:

```
def package_id(self):
    self.info.shared_library_package_id()
```

This helper automatically detects if the current package has the `shared` option and it is `True` and if it is depending on static libraries in other packages (having a `shared` option equal `False` or not having it, which means a header-only library). Only then, any change in the dependencies will affect the `package_id` of this package, (internally, `package_revision_mode` is applied to the dependencies). It is recommended its usage in packages that have the `shared` option.

If you want to have in your dependency graph all static libraries or all shared libraries, (but not shared with embedded static ones) it can be defined with a `*:shared=True` option in command line or profiles, but can also be defined in recipes like:

```
def configure(self):
    if self.options.shared:
        self.options["*"].shared = True
```

self.info.vs_toolset_compatible() / self.info.vs_toolset_incompatible()

By default (`vs_toolset_compatible()` mode) Conan will generate the same binary package when the compiler is Visual Studio and the `compiler.toolset` matches the specified `compiler.version`. For example, if we install some packages specifying the following settings:

```
def package_id(self):
    self.info.vs_toolset_compatible()
    # self.info.vs_toolset_incompatible()
```

```
compiler="Visual Studio"
compiler.version=14
```

And then we install again specifying these settings:

```
compiler="Visual Studio"
compiler.version=15
compiler.toolset=v140
```

The compiler version is different, but Conan will not install a different package, because the used `toolchain` in both cases are considered the same. You can deactivate this default behavior using calling `self.info.vs_toolset_incompatible()`.

This is the relation of Visual Studio versions and the compatible toolchain:

Visual Studio Version	Compatible toolset
15	v141
14	v140
12	v120
11	v110
10	v100
9	v90
8	v80

self.info.discard_build_settings() / self.info.include_build_settings()

By default (`discard_build_settings()`) Conan will generate the same binary when you change the `os_build` or `arch_build` when the `os` and `arch` are declared respectively. This is because `os_build` represent the machine running Conan, so, for the consumer, the only setting that matters is where the built software will run, not where is running the compilation. The same applies to `arch_build`.

With `self.info.include_build_settings()`, Conan will generate different packages when you change the `os_build` or `arch_build`.

```
def package_id(self):
    self.info.discard_build_settings()
    # self.info.include_build_settings()
```

self.info.default_std_matching() / self.info.default_std_non_matching()

By default (`default_std_matching()`) Conan will detect the default C++ standard of your compiler to not generate different binary packages.

For example, you already built some `gcc 6.1` packages, where the default std is `gnu14`. If you specify a value for the setting `compiler.cppstd` equal to the default one, `gnu14`, Conan won't generate new packages, because it was already the default of your compiler.

With `self.info.default_std_non_matching()`, Conan will generate different packages when you specify the `compiler.cppstd` even if it matches with the default of the compiler being used:

```
def package_id(self):
    self.info.default_std_non_matching()
    # self.info.default_std_matching()
```

Same behavior applies if you use the deprecated setting `cppstd`.

Compatible packages

The `package_id()` method serves to define the "canonical" binary package ID, the identifier of the binary that correspond to the input configuration of settings and options. This canonical binary package ID will be always computed, and Conan will check for its existence to be downloaded and installed.

If the binary of that package ID is not found, Conan lets the recipe writer define an ordered list of compatible package IDs, of other configurations that should be binary compatible and can be used as a fallback. The syntax to do this is:

```
from conans import ConanFile

class Pkg(ConanFile):
    settings = "os", "compiler", "arch", "build_type"

    def package_id(self):
        if self.settings.compiler == "gcc" and self.settings.compiler.version == "4.9":
            compatible_pkg = self.info.clone()
            compatible_pkg.settings.compiler.version = "4.8"
            self.compatible_packages.append(compatible_pkg)
```

This will define that, if we try to install this package with `gcc 4.9` and there isn't a binary available for that configuration, Conan will check if there is one available built with `gcc 4.8` and use it. But not the other way round.

See also:

For more information about *compatible packages read this*

build_id()

In the general case, there is one build folder for each binary package, with the exact same hash/ID of the package. However this behavior can be changed, there are a couple of scenarios that this might be interesting:

- You have a build script that generates several different configurations at once, like both debug and release artifacts, but you actually want to package and consume them separately. Same for different architectures or any other setting.
- You build just one configuration (like release), but you want to create different binary packages for different consuming cases. For example, if you have created tests for the library in the build step, you might want to create

two packages: one just containing the library for general usage, and another one also containing the tests. First package could be used as a reference and the other one as a tool to debug errors.

In both cases, if using different settings, the system will build twice (or more times) the same binaries, just to produce a different final binary package. With the `build_id()` method this logic can be changed. `build_id()` will create a new package ID/hash for the build folder, and you can define the logic you want in it. For example:

```
settings = "os", "compiler", "arch", "build_type"

def build_id(self):
    self.info_build.settings.build_type = "Any"
```

So this recipe will generate a final different package for each debug/release configuration. But as the `build_id()` will generate the same ID for any `build_type`, then just one folder and one build will be done. Such build should build both debug and release artifacts, and then the `package()` method should package them accordingly to the `self.settings.build_type` value. Different builds will still be executed if using different compilers or architectures. This method is basically an optimization of build time, avoiding multiple re-builds.

Other information like custom package options can also be changed:

```
def build_id(self):
    self.info_build.options.myoption = 'MyValue' # any value possible
    self.info_build.options.fullsource = 'Always'
```

If the `build_id()` method does not modify the `build_id`, and produce a different one than the `package_id`, then the standard behavior will be applied. Consider the following:

```
settings = "os", "compiler", "arch", "build_type"

def build_id(self):
    if self.settings.os == "Windows":
        self.info_build.settings.build_type = "Any"
```

This will only produce a build ID different if the package is for Windows. So the behavior in any other OS will be the standard one, as if the `build_id()` method was not defined: the build folder will be wiped at each **`conan create`** command and a clean build will be done.

deploy()

This method can be used in a *conanfile.py* to install in the system or user folder artifacts from packages.

```
def deploy(self):
    self.copy("*.exe")  # copy from current package
    self.copy_deps("*.dll") # copy from dependencies
```

Where:

- `self.copy()` is the `self.copy()` method executed inside *package() method*.
- `self.copy_deps()` is the same as `self.copy()` method inside *imports() method*.

Both methods allow the definition of absolute paths (to install in the system), in the `dst` argument. By default, the `dst` destination folder will be the current one.

The `deploy()` method is designed to work on a package that is installed directly from its reference, as:

```
$ conan install pkg/0.1@user/channel
> ...
> pkg/0.1@user/testing deploy(): Copied 1 '.dll' files: mylib.dll
> pkg/0.1@user/testing deploy(): Copied 1 '.exe' files: myexe.exe
```

All other packages and dependencies, even transitive dependencies of `pkg/0.1@user/testing` will not be deployed, it is the responsibility of the installed package to deploy what it needs from its dependencies.

See also:

For a different approach to deploy package files in the user space folders, check the *deploy generator*.

init()

This is an optional method for initializing conanfile values, designed for inheritance from *python requires*. Assuming we have a `base/1.1@user/testing` recipe:

```
class MyConanfileBase(object):
    license = "MyLicense"
    settings = "os", # tuple!


class PyReq(ConanFile):
    name = "base"
    version = "1.1"
```

We could reuse and inherit from it with:

```
class PkgTest(ConanFile):
    license = "MIT"
    settings = "arch", # tuple!
    python_requires = "base/1.1@user/testing"
    python_requires_extend = "base.MyConanfileBase"

    def init(self):
        base = self.python_requires["base"].module.MyConanfileBase
        self.settings = base.settings + self.settings  # Note, adding 2 tuples = tuple
        self.license = base.license  # License is overwritten
```

The final `PkgTest` conanfile will have both `os` and `arch` as settings, and `MyLicense` as license.

This method can also be useful if you need to unconditionally initialize class attributes like `license` or `description` or any other *attributes* from datafiles other than *conandata.yml*. For example, you have a *json* file containing the information about the `license`, `description` and `author` for the library:

Listing 10: *data.json*

```
{"license": "MIT", "description": "This is my awesome library.", "author": "Me"}
```

Then, you can load that information from the `init()` method:

```
import os
import json
from conans import ConanFile, load
```

(continues on next page)

(continued from previous page)

```
class Lib(ConanFile):
    exports = "data.json"
    def init(self):
        data = load(os.path.join(self.recipe_folder, "data.json"))
        d = json.loads(data)
        self.license = d["license"]
        self.description = d["description"]
        self.author = d["author"]
```

export()

Equivalent to the `exports` attribute, but in method form. It supports the `self.copy()` to do pattern based copy of files from the local user folder (the folder containing the *conanfile.py*) to the cache `export_folder`

```
from conans import ConanFile

class Pkg(ConanFile):

    def export(self):
        self.copy("LICENSE.md")
```

The current folder (`os.getcwd()`) and the `self.export_folder` can be used in the method:

```
import os
from conans import ConanFile
from conans.tools import save, load

class Pkg(ConanFile):

    def export(self):
        # we can load files in the user folder
        content = load(os.path.join(os.getcwd(), "data.txt"))
        # We have access to the cache export_folder
        save(os.path.join(self.export_folder, "myfile.txt"), "some content")
```

The `self.copy` support `src` and `dst` subfolder arguments. The `src` is relative to the current folder (the one containing the *conanfile.py*). The `dst` is relative to the cache `export_folder`.

```
from conans import ConanFile

class Pkg(ConanFile):

    def export(self):
        self.output.info("Executing export() method")
        # will copy all .txt files from the local "subfolder" folder to the cache "mydata
↪" one
        self.copy("*.txt", src="mysubfolder", dst="mydata")
```

export_sources()

Equivalent to the `exports_sources` attribute, but in method form. It supports the `self.copy()` to do pattern based copy of files from the local user folder (the folder containing the *conanfile.py*) to the cache `export_sources_folder`

```
from conans import ConanFile

class Pkg(ConanFile):

    def export_sources(self):
        self.copy("LICENSE.md")
```

The current folder (`os.getcwd()`) and the `self.export_sources_folder` can be used in the method:

```
import os
from conans import ConanFile
from conans.tools import save, load

class Pkg(ConanFile):

    def export_sources(self):
        content = load(os.path.join(os.getcwd(), "data.txt"))
        save(os.path.join(self.export_sources_folder, "myfile.txt"), content)
```

Note, if the recipe defines the `layout()` method and specifies a `self.folders.source = "src"` it won't change the current folder in the `export_sources` method. The current dir will be the base source folder (`self.export_sources_folder`).

The `self.copy` support `src` and `dst` subfolder arguments. The `src` is relative to the current folder (the one containing the *conanfile.py*). The `dst` is relative to the cache `export_sources_folder`.

```
from conans import ConanFile

class Pkg(ConanFile):

    def export_sources(self):
        self.output.info("Executing export_sources() method")
        # will copy all .txt files from the local "subfolder" folder to the cache "mydata
↪" one
        self.copy("*.txt", src="mysubfolder", dst="mydata")
```

generate()

Warning: This is an **experimental** feature subject to breaking changes in future releases.

Available since: 1.32.0

This method will run after the computation and installation of the dependency graph. This means that it will run after a **`conan install`** command, or when a package is being built in the cache, it will be run before calling the `build()` method.

The purpose of `generate()` is to prepare the build, generating the necessary files. These files would typically be:

- Files containing information to locate the dependencies, as `xxxx-config.cmake` CMake config scripts, or `xxxx.props` Visual Studio property files.
- Environment activation scripts, like `conanbuildenv.bat` or `conanbuildenv.sh`, that define all the necessary environment variables necessary for the build.
- Toolchain files, like `conan_toolchain.cmake`, that contains a mapping between the current Conan settings and options, and the build system specific syntax.
- General purpose build information, as a `conanbuild.conf` file that could contain information like the CMake generator or CMake toolchain file to be used in the `build()` method.
- Specific build system files, like `conanvcvars.bat`, that contains the necessary Visual Studio vcvars.bat call for certain build systems like Ninja when compiling with the Microsoft compiler.

The idea is that the `generate()` method implements all the necessary logic, making both the user manual builds after a **`conan install`** very straightforward, and also the `build()` method logic simpler. The build produced by a user in their local flow should result exactly the same one as the build done in the cache with a `conan create` without effort.

In many cases, the `generate()` method might not be necessary, and declaring the `generators` attribute could be enough:

```
from conans import ConanFile

class Pkg(ConanFile):
    generators = "CMakeDeps", "CMakeToolchain"
```

But the `generate()` method can explicitly instantiate those generators, customize them, or provide a complete custom generation. For custom integrations, putting code in a common `python_require` would be a good way to avoid repetition in multiple recipes.

```
from conans import ConanFile
from conan.tools.cmake import CMakeToolchain

class Pkg(ConanFile):

    def generate(self):
        tc = CMakeToolchain(self)
        # customize toolchain "tc"
        tc.generate()
        # Or provide your own custom logic
```

layout()

Warning: This is an **experimental** feature subject to breaking changes in future releases. The `layout()` feature will be fully functional only in the new build system integrations (*in the conan.tools space*). If you are using other integrations, they might not fully support this feature.

Available since: 1.37.0

Read about the feature *here*.

In the layout() method you can adjust `self.folders` and `self.cpp`.

self.folders

- **self.folders.source** (Defaulted to ""): Specifies a subfolder where the sources are. The `self.source_folder` attribute inside the `source(self)` and `build(self)` methods will be set with this subfolder. But the *current working directory* in the `source(self)` method will not include this subfolder, because it is intended to describe where the sources are after downloading (zip, git...) them, not to force where the sources should be. As well, the *export_sources*, *exports* and *scm* sources will be copied to the root source directory, being the **self.folders.source** variable the way to describe if the fetched sources are still in a subfolder. It is used in the cache when running **conan create** (relative to the cache source folder) as well as in a local folder when running **conan build** (relative to the local current folder).
- **self.folders.build** (Defaulted to ""): Specifies a subfolder where the files from the build are. The `self.build_folder` attribute and the *current working directory* inside the `build(self)` method will be set with this subfolder. It is used in the cache when running **conan create** (relative to the cache source folder) as well as in a local folder when running **conan build** (relative to the local current folder).
- **self.folders.generators** (Defaulted to ""): Specifies a subfolder where to write the files from the generators and the toolchains. In the cache, when running the **conan create**, this subfolder will be relative to the root build folder and when running the **conan install** command it will be relative to the current working directory.
- **self.folders.imports** (Defaulted to ""): Specifies a subfolder where to write the files copied when using the `imports(self)` method in a `conanfile.py`. In the cache, when running the **conan create**, this subfolder will be relative to the root build folder and when running the **conan imports** command it will be relative to the current working directory.

self.cpp

The `layout()` method allows to declare `cpp_info` objects not only for the final package (like the classic approach with the `self.cpp_info` in the `package_info(self)` method) but for the `self.source_folder` and `self.build_folder`.

The fields of the cpp_info objects at `self.cpp.build` and `self.cpp.source` are the same described *here*. Components are also supported.

test()

The `test()` method is only used for *test_package/conanfile.py* recipes. It will execute immediately after `build()` has been called, and its goal is to run some executable or tests on binaries to prove the package is correctly created. Note that it is intended to be used as a test of the package: the headers are there, the libraries are there, it is possible to link, etc., but not to run unit, integration or functional tests.

It usually takes the form:

```
def test(self):
    if not tools.cross_building(self):
        self.run(os.path.sep.join([".", "bin", "example"]))
```

Note the `tools.cross_building()` check, as it is not possible to run executables different to the build machine architecture. In this case, it would make sense to check the existence of the binary, or inspect it with tools like `dumpbin`, `lipo`, etc to do basic checks about it.

The `self.run()` might need some environment help, in case the execution needs for example shared libraries location.

18.3.3 tools

Tools are all things that can be imported and used in Conan recipes.

> **Warning:** These tools are **experimental** and subject to breaking changes.

Important: This is the current design for Conan 2.0, and these will be the supported tools. Only the tools documented in this section will be available in Conan 2.0.

Most of the utilities defined in "conan.tools" will require very soon to define both the "host" and "build" profiles. It is very recommended to start defining both profiles immediately to avoid future breaking. Furthermore, some features, like trying to cross-compile might not work at all if the "build" profile is not provided.

The import path is always like:

```
from conan.tools.cmake import CMakeToolchain, CMakeDeps, CMake
form conan.tools.microsoft import MSBuildToolchain, MSBuildDeps, MSBuild
```

The main guidelines are:

- Everything that recipes can import belong to `from conan.tools`. Any other thing is private implementation and shouldn't be used in recipes.
- Only documented, public (not preceded by _) tools can be used in recipes.

Contents:

conan.tools.cmake

> **Warning:** These tools are still **experimental** (so subject to breaking changes) but with very stable syntax. We encourage the usage of it to be prepared for Conan 2.0.

You can use `conan new hello/0.1 --template=cmake_lib` and `conan new hello/0.1 --template=cmake_exe` templates to try this CMake integration.

CMakeDeps

> **Warning:** These tools are still **experimental** (so subject to breaking changes) but with very stable syntax. We encourage the usage of it to be prepared for Conan 2.0.

Available since: 1.33.0

The `CMakeDeps` helper will generate one **xxxx-config.cmake** file per dependency, together with other necessary *.cmake* files like version, flags and directory data or configuration. It can be used like:

```
from conans import ConanFile

class App(ConanFile):
```

(continues on next page)

(continued from previous page)

```
    settings = "os", "arch", "compiler", "build_type"
    requires = "hello/0.1"
    generators = "CMakeDeps"
```

The full instantiation, that allows custom configuration can be done in the `generate()` method:

```
from conans import ConanFile
from conan.tools.cmake import CMakeDeps

class App(ConanFile):
    settings = "os", "arch", "compiler", "build_type"
    requires = "hello/0.1"

    def generate(self):
        cmake = CMakeDeps(self)
        cmake.generate()
```

Important: This class will require very soon to define both the "host" and "build" profiles. It is very recommended to start defining both profiles immediately to avoid future breaking. Furthermore, some features, like trying to cross-compile might not work at all if the "build" profile is not provided.

The `CMakeDeps` is a *multi-configuration* generator, it can correctly create files for Release/Debug configurations to be simultaneously used by IDEs like Visual Studio. In single configuration environments, it is necessary to have a configuration defined, which must be provided via the `cmake ... -DCMAKE_BUILD_TYPE=<build-type>` argument in command line (Conan will do it automatically when necessary, in the `CMake.configure()` helper).

There are some attributes you can adjust in the created `CMakeDeps` object to change the default behavior:

configuration

Allows to define custom user CMake configuration besides the standard Release, Debug, etc ones.

```
def generate(self):
    deps = CMakeDeps(self)
    # By default, ``deps.configuration`` will be ``self.settings.build_type``
    if self.options["hello"].shared:
        # Assuming the current project ``CMakeLists.txt`` defines the ReleasedShared␣
↪configuration.
        deps.configuration = "ReleaseShared"
    deps.generate()
```

build_context_activated

When you have a **build-require**, by default, the config files (*xxx-config.cmake*) files are not generated. But you can activate it using the **build_context_activated** attribute:

```
tool_requires = ["my_tool/0.0.1"]

def generate(self):
    cmake = CMakeDeps(self)
    # generate the config files for the tool require
    cmake.build_context_activated = ["my_tool"]
    cmake.generate()
```

Warning: The `build_context_activated` feature will fail if no "build" profile is used. This feature only work when using the two host and build profiles.

build_context_suffix

When you have the same package as a **build-require** and as a **regular require** it will cause a conflict in the generator because the file names of the config files will collide as well as the targets names, variables names etc.

For example, this is a typical situation with some requirements (capnproto, protobuf...) that contain a tool used to generate source code at build time (so it is a **build_require**), but also providing a library to link to the final application, so you also have a **regular require**. Solving this conflict is specially important when we are cross-building because the tool (that will run in the building machine) belongs to a different binary package than the library, that will "run" in the host machine.

You can use the **build_context_suffix** attribute to specify a suffix for a requirement, so the files/targets/variables of the requirement in the build context (tool require) will be renamed:

```
tool_requires = ["my_tool/0.0.1"]
requires = ["my_tool/0.0.1"]

def generate(self):
    cmake = CMakeDeps(self)
    # generate the config files for the tool require
    cmake.build_context_activated = ["my_tool"]
    # disambiguate the files, targets, etc
    cmake.build_context_suffix = {"my_tool": "_BUILD"}
    cmake.generate()
```

Warning: The `build_context_suffix` feature will fail if no "build" profile is used. This feature only work when using the two host and build profiles.

build_context_build_modules

Also there is another issue with the **build_modules**. As you may know, the recipes of the requirements can declare a *cppinfo.build_modules* entry containing one or more **.cmake** files. When the requirement is found by the cmake `find_package()` function, Conan will include automatically these files.

By default, Conan will include only the build modules from the `host` context (regular requires) to avoid the collision, but you can change the default behavior.

Use the **build_context_build_modules** attribute to specify require names to include the **build_modules** from **tool_requires**:

```
tool_requires = ["my_tool/0.0.1"]

def generate(self):
    cmake = CMakeDeps(self)
    # generate the config files for the tool require
    cmake.build_context_activated = ["my_tool"]
    # Choose the build modules from "build" context
    cmake.build_context_build_modules = ["my_tool"]
    cmake.generate()
```

Warning: The `build_context_build_modules` feature will fail if no "build" profile is used. This feature only work when using the two host and build profiles.

Properties

The following properties affect the CMakeDeps generator:

- **cmake_file_name**: The config file generated for the current package will follow the `<VALUE>-config.cmake` pattern, so to find the package you write `find_package(<VALUE>)`.
- **cmake_target_name**: Name of the target to be consumed.
- **cmake_target_aliases**: List of aliases that Conan will create for an already existing target.
- **cmake_find_mode**: Defaulted to `config`. Possible values are:
  - `config`: The CMakeDeps generator will create config scripts for the dependency.
  - `module`: Will create module config (FindXXX.cmake) scripts for the dependency.
  - `both`: Will generate both config and modules.
  - `none`: Won't generate any file. It can be used, for instance, to create a system wrapper package so the consumers find the config files in the CMake installation config path and not in the generated by Conan (because it has been skipped).
- **cmake_module_file_name**: Same as **cmake_file_name** but when generating modules with `cmake_find_mode=module/both`. If not specified it will default to **cmake_file_name**.
- **cmake_module_target_name**: Same as **cmake_target_name** but when generating modules with `cmake_find_mode=module/both`. If not specified it will default to **cmake_target_name**.
- **cmake_build_modules**: List of `.cmake` files (route relative to root package folder) that are automatically included when the consumer run the `find_package()`.

- **cmake_set_interface_link_directories**: boolean value that should be only used by dependencies that don't declare *self.cpp_info.libs* but have `#pragma comment(lib, "foo")` (automatic link) declared at the public headers. Those dependencies should add this property to their *conanfile.py* files at root `cpp_info` level (components not supported for now).

Example:

```
def package_info(self):
    ...
    # MyFileName-config.cmake
    self.cpp_info.set_property("cmake_file_name", "MyFileName")
    # Names for targets are absolute, Conan won't add any namespace to the target names␣
↪automatically
    self.cpp_info.set_property("cmake_target_name", "Foo::Foo")

    # Create a new target "MyFooAlias" that is an alias to the "Foo::Foo" target
    self.cpp_info.set_property("cmake_target_aliases", ["MyFooAlias"])

    self.cpp_info.components["mycomponent"].set_property("cmake_target_name", "Foo::Var")
    # Automatically include the lib/mypkg.cmake file when calling find_package()
    self.cpp_info.components["mycomponent"].set_property("cmake_build_modules", [os.path.
↪join("lib", "mypkg.cmake")])

    # Create a new target "VarComponent" that is an alias to the "Foo::Var" component␣
↪target
    self.cpp_info.components["mycomponent"].set_property("cmake_target_aliases", [
↪"VarComponent"])

    # Skip this package when generating the files for the whole dependency tree in the␣
↪consumer
    # note: it will make useless the previous adjustements.
    # self.cpp_info.set_property("cmake_find_mode", "none")

    # Generate both MyFileNameConfig.cmake and FindMyFileName.cmake
    self.cpp_info.set_property("cmake_find_mode", "both")
```

CMakeToolchain

Warning: These tools are still **experimental** (so subject to breaking changes) but with very stable syntax. We encourage the usage of it to be prepared for Conan 2.0.

The `CMakeToolchain` is the toolchain generator for CMake. It will generate toolchain files that can be used in the command line invocation of CMake with the `-DCMAKE_TOOLCHAIN_FILE=conan_toolchain.cmake`. This generator translates the current package configuration, settings, and options, into CMake toolchain syntax.

Important: This class will require very soon to define both the "host" and "build" profiles. It is very recommended to start defining both profiles immediately to avoid future breaking. Furthermore, some features, like trying to cross-compile might not work at all if the "build" profile is not provided.

It can be declared as:

```
from conans import ConanFile

class Pkg(ConanFile):
    generators = "CMakeToolchain"
```

Or fully instantiated in the `generate()` method:

```
from conans import ConanFile
from conan.tools.cmake import CMakeToolchain

class App(ConanFile):
    settings = "os", "arch", "compiler", "build_type"
    requires = "hello/0.1"
    generators = "CMakeDeps"
    options = {"shared": [True, False], "fPIC": [True, False]}
    default_options = {"shared": False, "fPIC": True}

    def generate(self):
        tc = CMakeToolchain(self)
        tc.variables["MYVAR"] = "MYVAR_VALUE"
        tc.preprocessor_definitions["MYDEFINE"] = "MYDEF_VALUE"
        tc.generate()
```

This will generate the following files after a `conan install` (or when building the package in the cache) with the information provided in the `generate()` method as well as information translated from the current `settings`:

- *conan_toolchain.cmake* file, containing the translation of Conan settings to CMake variables. Some things that will be defined in this file:
  - Definition of the CMake generator platform and generator toolset
  - Definition of the `CMAKE_POSITION_INDEPENDENT_CODE`, based on `fPIC` option.
  - Definition of the C++ standard as necessary
  - Definition of the standard library used for C++
  - Deactivation of rpaths in OSX
- *CMakePresets.json*: The toolchain also generates a `CMakePresets.json` standard file, check the documentation here. It is currently using the version "3" of the JSON schema. Conan creates a `default` configure preset with the information:
  - The `generator` to be used.
  - The path to the `conan_toolchain.cmake`
  - Some cache variables corresponding to the specified settings cannot work if specified in the toolchain.
  - The `CMAKE_BUILD_TYPE` variable when using a single-configuration generators.
- *CMakeUserPresets.json*: If you declare a `layout()` in the recipe and your `CMakeLists.txt` file is found at the `conanfile.source_folder` folder, a `CMakeUserPresets.json` file will be generated (if doesn't exist already) including automatically the `CMakePresets.json` (at the `conanfile.generators_folder`) to allow your IDE (Visual Studio, Visual Studio Code, CLion...) or `cmake` tool to locate the `CMakePresets.json`. The version schema of the generated `CMakeUserPresets.json` is "4" and requires CMake >= 3.23.
- *conanvcvars.bat*: In some cases, the Visual Studio environment needs to be defined correctly for building, like when using the Ninja or NMake generators. If necessary, the `CMakeToolchain` will generate this script, so defining the correct Visual Studio prompt is easier.

constructor

```
def __init__(self, conanfile):
```

- `conanfile`: the current recipe object. Always use `self`.

preprocessor_definitions

This attribute allows defining compiler preprocessor definitions, for multiple configurations (Debug, Release, etc).

```
def generate(self):
    tc = CMakeToolchain(self)
    tc.preprocessor_definitions["MYDEF"] = "MyValue"
    tc.preprocessor_definitions.debug["MYCONFIGDEF"] = "MyDebugValue"
    tc.preprocessor_definitions.release["MYCONFIGDEF"] = "MyReleaseValue"
    tc.generate()
```

This will be translated to:

- One `add_definitions()` definition for `MYDEF` in `conan_toolchain.cmake` file.
- One `add_definitions()` definition, using a cmake generator expression in `conan_toolchain.cmake` file, using the different values for different configurations.

variables

This attribute allows defining CMake variables, for multiple configurations (Debug, Release, etc).

```
def generate(self):
    tc = CMakeToolchain(self)
    tc.variables["MYVAR"] = "MyValue"
    tc.variables.debug["MYCONFIGVAR"] = "MyDebugValue"
    tc.variables.release["MYCONFIGVAR"] = "MyReleaseValue"
    tc.generate()
```

This will be translated to:

- One `set()` definition for `MYVAR` in `conan_toolchain.cmake` file.
- One `set()` definition, using a cmake generator expression in `conan_toolchain.cmake` file, using the different values for different configurations.

The booleans assigned to a variable will be translated to `ON` and `OFF` symbols in CMake:

```
def generate(self):
    tc = CMakeToolchain(self)
    tc.variables["FOO"] = True
    tc.variables["VAR"] = False
    tc.generate()
```

Will generate the sentences: `set(FOO ON ...)` and `set(VAR OFF ...)`.

Generators

The `CMakeToolchain` is intended to run with the `CMakeDeps` dependencies generator. Please do not use other CMake legacy generators (like `cmake`, or `cmake_paths`) with it.

Using a custom toolchain file

There are two ways of providing custom CMake toolchain files:

- The `conan_toolchain.cmake` file can be completely skipped and replaced by a user one, defining the `tools.cmake.cmaketoolchain:toolchain_file=<filepath>` configuration value.
- A custom user toolchain file can be added (included from) to the `conan_toolchain.cmake` one, by using the `user_toolchain` block described below, and defining the `tools.cmake.cmaketoolchain:user_toolchain=["<filepath>"]` configuration value.

  The configuration `tools.cmake.cmaketoolchain:user_toolchain=["<filepath>"]` can be defined in the *global.conf* but also creating a Conan package for your toolchain and using `self.conf_info` to declare the toolchain file:

  ```
  import os
  from conans import ConanFile
  class MyToolchainPackage(ConanFile):
      ...
      def package_info(self):
          f = os.path.join(self.package_folder, "mytoolchain.cmake")
          self.conf_info.define("tools.cmake.cmaketoolchain:user_toolchain",␣
  ↪[f])
  ```

  If you declare the previous package as a `tool_require`, the toolchain will be automatically applied.

- If you have more than one `tool_requires` defined, you can easily append all the user toolchain values together using the `append` method in each of them, for instance:

  ```
  import os
  from conans import ConanFile
  class MyToolRequire(ConanFile):
      ...
      def package_info(self):
          f = os.path.join(self.package_folder, "mytoolchain.cmake")
          # Appending the value to any existing one
          self.conf_info.append("tools.cmake.cmaketoolchain:user_toolchain",␣
  ↪f)
  ```

  So, they'll be automatically applied by your `CMakeToolchain` generator without writing any extra code:

  ```
  from conans import ConanFile
  from conan.tools.cmake import CMake
  class Pkg(ConanFile):
      settings = "os", "compiler", "arch", "build_type"
      exports_sources = "CMakeLists.txt"
      tool_requires = "toolchain1/0.1", "toolchain2/0.1"
      generators = "CMakeToolchain"
  ```

(continues on next page)

(continued from previous page)

```
    def build(self):
        cmake = CMake(self)
        cmake.configure()
```

Using the toolchain in developer flow

One of the advantages of using Conan toolchains is that they can help to achieve the exact same build with local development flows, than when the package is created in the cache.

```
# Lets start in the folder containing the conanfile.py
$ mkdir build && cd build
# Install both debug and release deps and create the toolchain
$ conan install ..
$ conan install .. -s build_type=Debug
# the conan_toolchain.cmake is common for both configurations
```

If you are using a multi-configuration generator:

```
# Need to pass the generator WITHOUT the platform, that matches your default settings
$ cmake .. -G "Visual Studio 15" -DCMAKE_TOOLCHAIN_FILE=conan_toolchain.cmake
# Now you can open the IDE, select Debug or Release config and build
# or, in the command line
$ cmake --build . --config Release
$ cmake --build . --config Debug
```

NOTE: The platform (Win64), is already encoded in the toolchain. The command line shouldn't pass it, so using `-G "Visual Studio 15"` instead of the `-G "Visual Studio 15 Win64"`

If you are using a single-configuration generator:

```
$ cmake ..  -DCMAKE_TOOLCHAIN_FILE=conan_toolchain.cmake -DCMAKE_BUILD_TYPE=Release
$ cmake --build
```

It is recommended to use the `cmake_layout(self)` in the `layout()` method of your `conanfile.py`. If a layout is declared, the `CMakeUserPresets.json` file will be generated in the same folder of your `CMakeLists.txt` file, so you can use the `--preset` argument from `cmake >= 3.23` or use an IDE:

```
# The conan_toolchain.cmake is common for both configurations and will be located at
↪"build/generators"
$ conan install .
$ conan install . -s build_type=Debug

# For single-configuration generator
$ cmake --preset Debug
$ cmake --build --preset Debug
$ cmake --preset Release
$ cmake --build --preset Release

# For multi-configuration generator
$ cmake --preset default
$ cmake --build --preset Debug
$ cmake --build --preset Release
```

conf

CMakeToolchain is affected by these *[conf]* variables:

- tools.cmake.cmaketoolchain:toolchain_file user toolchain file to replace the conan_toolchain.cmake one.
- tools.cmake.cmaketoolchain:user_toolchain list of user toolchains to be included from the conan_toolchain.cmake file.
- tools.android:ndk_path value for ANDROID_NDK_PATH.
- tools.cmake.cmaketoolchain:system_name is not necessary in most cases and is only used to force-define CMAKE_SYSTEM_NAME.
- tools.cmake.cmaketoolchain:system_version is not necessary in most cases and is only used to force-define CMAKE_SYSTEM_VERSION.
- tools.cmake.cmaketoolchain:system_processor is not necessary in most cases and is only used to force-define CMAKE_SYSTEM_PROCESSOR.
- tools.cmake.cmaketoolchain:toolset_arch: Will add the ,host=xxx specifier in the CMAKE_GENERATOR_TOOLSET variable of conan_toolchain.cmake file.
- tools.build:cxxflags list of extra C++ flags that will be appended to CMAKE_CXX_FLAGS_INIT.
- tools.build:cflags list of extra of pure C flags that will be appended to CMAKE_C_FLAGS_INIT.
- tools.build:sharedlinkflags list of extra linker flags that will be appended to CMAKE_SHARED_LINKER_FLAGS_INIT.
- tools.build:exelinkflags list of extra linker flags that will be appended to CMAKE_EXE_LINKER_FLAGS_INIT.
- tools.build:defines list of preprocessor definitions that will be used by add_definitions().
- tools.build:tools.apple:enable_bitcode boolean value to enable/disable Bitcode Apple Clang flags, e.g., CMAKE_XCODE_ATTRIBUTE_ENABLE_BITCODE.
- tools.build:tools.apple:enable_arc boolean value to enable/disable ARC Apple Clang flags, e.g., CMAKE_XCODE_ATTRIBUTE_CLANG_ENABLE_OBJC_ARC.
- tools.build:tools.apple:enable_visibility boolean value to enable/disable Visibility Apple Clang flags, e.g., CMAKE_XCODE_ATTRIBUTE_GCC_SYMBOLS_PRIVATE_EXTERN.
- tools.build:sysroot defines the value of CMAKE_SYSROOT.

Extending and customizing CMakeToolchain

Since Conan 1.36, CMakeToolchain implements a powerful capability for extending and customizing the resulting toolchain file.

The following predefined blocks are available, and added in this order:

- user_toolchain: Allows to include user toolchains from the conan_toolchain.cmake file. If the configuration tools.cmake.cmaketoolchain:user_toolchain=["xxxx", "yyyy"] is defined, its values will be include(xxx)\ninclude(yyyy) as the first lines in conan_toolchain.cmake.
- generic_system: Defines CMAKE_SYSTEM_NAME, CMAKE_SYSTEM_VERSION, CMAKE_SYSTEM_PROCESSOR, CMAKE_GENERATOR_PLATFORM, CMAKE_GENERATOR_TOOLSET, CMAKE_C_COMPILER, CMAKE_CXX_COMPILER

- `android_system`: Defines `ANDROID_PLATFORM`, `ANDROID_STL`, `ANDROID_ABI` and includes `ANDROID_NDK_PATH/build/cmake/android.toolchain.cmake` where `ANDROID_NDK_PATH` comes defined in `tools.android:ndk_path` configuration value.
- `apple_system`: Defines `CMAKE_OSX_ARCHITECTURES`, `CMAKE_OSX_SYSROOT` for Apple systems.
- `fpic`: Defines the `CMAKE_POSITION_INDEPENDENT_CODE` when there is a `options.fPIC`
- `arch_flags`: Defines C/C++ flags like `-m32, -m64` when necessary.
- `libcxx`: Defines `-stdlib=libc++` flag when necessary as well as `_GLIBCXX_USE_CXX11_ABI`.
- `vs_runtime`: Defines the `CMAKE_MSVC_RUNTIME_LIBRARY` variable, as a generator expression for multiple configurations.
- `cppstd`: defines `CMAKE_CXX_STANDARD`, `CMAKE_CXX_EXTENSIONS`
- `parallel`: defines `/MP` parallel build flag for Visual.
- `cmake_flags_init`: defines `CMAKE_XXX_FLAGS` variables based on previously defined Conan variables. The blocks above only define `CONAN_XXX` variables, and this block will define CMake ones like `set(CMAKE_CXX_FLAGS_INIT "${CONAN_CXX_FLAGS}" CACHE STRING "" FORCE)`.
- `try_compile`: Stop processing the toolchain, skipping the blocks below this one, if `IN_TRY_COMPILE` CMake property is defined.
- `find_paths`: Defines `CMAKE_FIND_PACKAGE_PREFER_CONFIG`, `CMAKE_MODULE_PATH`, `CMAKE_PREFIX_PATH` so the generated files from `CMakeDeps` are found.
- `rpath`: Defines `CMAKE_SKIP_RPATH`. By default it is disabled, and it is needed to define `self.blocks["rpath"].skip_rpath=True` if you want to activate `CMAKE_SKIP_RPATH`
- `shared`: defines `BUILD_SHARED_LIBS`.
- `output_dirs`: Define the `CMAKE_INSTALL_XXX` variables.
  - **CMAKE_INSTALL_PREFIX**: Is set with the `package_folder`, so if a "cmake install" operation is run, the artifacts go to that location.
  - **CMAKE_INSTALL_BINDIR**, **CMAKE_INSTALL_SBINDIR** and **CMAKE_INSTALL_LIBEXECDIR**: Set by default to `bin`.
  - **CMAKE_INSTALL_LIBDIR**: Set by default to `lib`.
  - **CMAKE_INSTALL_INCLUDEDIR** and **CMAKE_INSTALL_OLDINCLUDEDIR**: Set by default to `include`.
  - **CMAKE_INSTALL_DATAROOTDIR**: Set by default to `res`.

  If you want to change the default values, adjust the `cpp.package` object at the `layout()` method:

```
def layout(self):
    ...
    # For CMAKE_INSTALL_BINDIR, CMAKE_INSTALL_SBINDIR and CMAKE_
↪INSTALL_LIBEXECDIR, takes the first value:
    self.cpp.package.bindirs = ["mybin"]
    # For CMAKE_INSTALL_LIBDIR, takes the first value:
    self.cpp.package.libdirs = ["mylib"]
    # For CMAKE_INSTALL_INCLUDEDIR, CMAKE_INSTALL_OLDINCLUDEDIR,␣
↪takes the first value:
    self.cpp.package.includedirs = ["myinclude"]
    # For CMAKE_INSTALL_DATAROOTDIR, takes the first value:
    self.cpp.package.resdirs = ["myres"]
```

Note: It is **not valid** to change the self.cpp_info at the `package_info()` method.

Note: In Conan 1.45 the CMakeToolchain doesn't append the root package folder of the dependencies (declared in the cpp_info.builddirs) to the `CMAKE_PREFIX_PATH` variable. That interfered with the `find_file`, `find_path` and `find_program`, making, for example, impossible to locate only the executables from the build context. In Conan 2.0, the `cppinfo.builddirs` won't contain by default the `''` entry (root package).

Blocks can be customized in different ways:

```
# remove an existing block
def generate(self):
    tc = CMakeToolchain(self)
    tc.blocks.remove("generic_system")

# modify the template of an existing block
def generate(self):
    tc = CMakeToolchain(self)
    tmp = tc.blocks["generic_system"].template
    new_tmp = tmp.replace(...)  # replace, fully replace, append...
    tc.blocks["generic_system"].template = new_tmp

# modify one or more variables of the context
def generate(self):
    tc = CMakeToolchain(conanfile)
    # block.values is the context dictionary
    toolset = tc.blocks["generic_system"].values["toolset"]
    tc.blocks["generic_system"].values["toolset"] = "other_toolset"

# modify the whole context values
def generate(self):
    tc = CMakeToolchain(conanfile)
    tc.blocks["generic_system"].values = {"toolset": "other_toolset"}

# modify the context method of an existing block
import types

def generate(self):
    tc = CMakeToolchain(self)
    generic_block = toolchain.blocks["generic_system"]

    def context(self):
        assert self  # Your own custom logic here
        return {"toolset": "other_toolset"}
    generic_block.context = types.MethodType(context, generic_block)

# completely replace existing block
from conan.tools.cmake import CMakeToolchain

def generate(self):
    tc = CMakeToolchain(self)
```

(continues on next page)

(continued from previous page)

```
    # this could go to a python_requires
    class MyGenericBlock:
        template = "HelloWorld"

        def context(self):
            return {}

    tc.blocks["generic_system"] = MyGenericBlock

# add a completely new block
from conan.tools.cmake import CMakeToolchain
def generate(self):
    tc = CMakeToolchain(self)
    # this could go to a python_requires
    class MyBlock:
        template = "Hello {{myvar}}!!!"

        def context(self):
            return {"myvar": "World"}

    tc.blocks["mynewblock"] = MyBlock
```

Recall that this is a very **experimental** feature, and these interfaces might change in the following releases.

For more information about these blocks, please have a look at the source code.

Cross building

The `generic_system` block contains some basic cross-building capabilities. In the general case, the user would want to provide their own user toolchain defining all the specifics, which can be done with the configuration `tools.cmake.cmaketoolchain:user_toolchain`. If this conf value is defined, the `generic_system` block will include the provided file or files, but no further define any CMake variable for cross-building.

If `user_toolchain` is not defined and Conan detects it is cross-building, because the build and host profiles contain different OS or architecture, it will try to define the following variables:

- `CMAKE_SYSTEM_NAME`: `tools.cmake.cmaketoolchain:system_name` configuration if defined, otherwise, it will try to autodetect it. This block will consider cross-building if Android systems (that is managed by other blocks), and not 64bits to 32bits builds in x86_64, sparc and ppc systems.
- `CMAKE_SYSTEM_VERSION`: `tools.cmake.cmaketoolchain:system_version` conf if defined, otherwise `os.version` subsetting (host) when defined
- `CMAKE_SYSTEM_PROCESSOR`: `tools.cmake.cmaketoolchain:system_processor` conf if defined, otherwise `arch` setting (host) if defined

CMake

Warning: These tools are still **experimental** (so subject to breaking changes) but with very stable syntax. We encourage the usage of it to be prepared for Conan 2.0.

The `CMake` build helper is a wrapper around the command line invocation of cmake. It will abstract the calls like `cmake --build . --config Release` into Python method calls. It will also add the argument `-DCMAKE_TOOLCHAIN_FILE=conan_toolchain.cmake` to the `configure()` call, as well as other possible arguments like `-DCMAKE_BUILD_TYPE=<config>`. The arguments that will be used are obtained from a generated `CMakePresets.json` file.

The helper is intended to be used in the `build()` method, to call CMake commands automatically when a package is being built directly by Conan (create, install)

```
from conans import ConanFile
from conan.tools.cmake import CMake, CMakeToolchain, CMakeDeps

class App(ConanFile):
    settings = "os", "arch", "compiler", "build_type"
    requires = "hello/0.1"
    options = {"shared": [True, False], "fPIC": [True, False]}
    default_options = {"shared": False, "fPIC": True}

    def generate(self):
        tc = CMakeToolchain(self)
        tc.generate()
        deps = CMakeDeps(self)
        deps.generate()

    def build(self):
        cmake = CMake(self)
        cmake.configure()
        cmake.build()
```

Note: This helper includes the additional flag *-DCMAKE_SH="CMAKE_SH-NOTFOUND"* when using the *MinGW Makefiles* CMake's generator, to avoid the error of *sh* being in the PATH (CMake version < 3.17.0).

It supports the following methods:

constructor

```
def __init__(self, conanfile):
```

- `conanfile`: the current recipe object. Always use `self`.

configure()

```
def configure(self, variables=None, build_script_folder=None):
```

Reads the `CMakePresets.json` file generated by the *CMakeToolchain* to get:

- The generator, to append `-G="xxx"`.
- The path to the toolchain and append `-DCMAKE_TOOLCHAIN_FILE=/path/conan_toolchain.cmake`
- The declared `cache variables` and append `-Dxxx`.
- `build_script_folder`: Relative path to the folder containing the root *CMakeLists.txt*

and call `cmake`.

Important: If `CMakePresets.json` file is not there, Conan will raise an exception because it's a mandatory one even though it's empty.

- `variables`: should be a dictionary of CMake variables and values, that will be mapped to command line `-DVAR=VALUE` arguments. Recall that in the general case information to CMake should be passed in `CMakeToolchain` to be provided in the `conan_toolchain.cmake` file. This `variables` argument is intended for exceptional cases that wouldn't work in the toolchain approach.

build()

```
def build(self, build_type=None, target=None, cli_args=None, build_tool_args=None):
```

Calls the build system. Equivalent to **`cmake --build .`** in the build folder.

- `build_type`: Use it only to override the value defined in the `settings.build_type` for a multi-configuration generator (e.g. Visual Studio, XCode). This value will be ignored for single-configuration generators, they will use the one defined in the toolchain file during the install step.
- `target`: name of the build target to run.
- `cli_args`: A list of arguments `[arg1, arg2, ...]` that will be passed to the `cmake --build ... arg1 arg2` command directly.
- `build_tool_args`: A list of arguments `[barg1, barg2, ...]` for the underlying build system that will be passed to the command line after the `--` indicator: `cmake --build ... -- barg1 barg2`

install()

```
def install(self, build_type=None):
```

Equivalent to run `cmake --build . --target=install`

- `build_type`: Use it only to override the value defined in the `settings.build_type`. It can fail if the build is single configuration (e.g. Unix Makefiles), as in that case the build type must be specified at configure time, not build type.

test()

```
def test(self, build_type=None, target=None, cli_args=None, build_tool_args=None):
```

Equivalent to running **`cmake --build . --target=RUN_TESTS`**.

- `build_type`: Use it only to override the value defined in the `settings.build_type`. It can fail if the build is single configuration (e.g. Unix Makefiles), as in that case the build type must be specified at configure time, not build type.
- `target`: name of the build target to run, by default `RUN_TESTS` or `test`.
- `cli_args`: Same as above `build()`
- `build_tool_args`: Same as above `build()`

conf

- `tools.microsoft.msbuild:verbosity` will accept one of `"Quiet"`, `"Minimal"`, `"Normal"`, `"Detailed"`, `"Diagnostic"` to be passed to the `CMake.build()` command, when a Visual Studio generator (MSBuild build system) is being used for CMake. It is passed as an argument to the underlying build system via the call `cmake --build . --config Release -- /verbosity:Diagnostic`
- `tools.build:jobs` argument for the `--jobs` parameter when running Ninja generator.
- `tools.microsoft.msbuild:max_cpu_count` argument for the `/m` (`/maxCpuCount`) when running `MSBuild`

cmake_layout

> **Warning:** These tools are still **experimental** (so subject to breaking changes) but with very stable syntax. We encourage the usage of it to be prepared for Conan 2.0.

For example, this would implement the standard CMake project layout:

```
from conan.tools.cmake import cmake_layout

def layout(self):
    cmake_layout(self)
```

> **Note:** To try it you can use the `conan new hello/0.1 --template=cmake_lib` template.

The `cmake_layout()` sets the `folders` and `cpp` attributes described in the (*layout reference*).

The assigned values depend on the CMake generator that will be used. It can be defined with the `tools.cmake.cmaketoolchain:generator` [conf] entry or passing it in the recipe to the `cmake_layout(self, cmake_generator)` function. The assigned values are different if it is a multi-config generator (like Visual Studio or Xcode), or a single-config generator (like Unix Makefiles).

These are the values assigned by the `cmake_layout`:

- `conanfile.folders.source`: *src_folder* argument or `.` if not specified.
- **`conanfile.folders.build`:**
  - `build`: if the cmake generator is multi-configuration.
  - `cmake-build-debug` or `cmake-build-debug`: if the cmake generator is single-configuration, depending on the build_type.
- `conanfile.folders.generators`: `build/generators`
- `conanfile.cpp.source.includedirs`: `["include"]`
- **`conanfile.cpp.build.libdirs` and `conanfile.cpp.build.bindirs`:**
  - `["Release"]` or `["Debug"]` for a multi-configuration cmake generator.
  - `.` for a single-configuration cmake generator.

```
def layout(self):
    cmake_layout(self, src_folder="subfolder")
```

Multi-setting/option cmake_layout

The `folders.build` and `conanfile.folders.generators` can be customized to take into account the `settings` and `options` and not only the `build_type`. Use the `tools.cmake.cmake_layout:build_folder_vars` conf to declare a list of settings or options:

```
conan install . -c tools.cmake.cmake_layout:build_folder_vars='["settings.compiler",
↪"options.shared"]'
```

For the previous example, the values assigned by the `cmake_layout` (installing the Release/static default configuration) would be:

- **`conanfile.folders.build`:**
  - `build-apple-clang-shared_false`: if the cmake generator is multi-configuration.
  - `cmake-build-debug-apple-clang-shared_false`: if the cmake generator is single-configuration.
- `conanfile.folders.generators`: `build-apple-clang-shared_false/generators`

If we repeat the previous install with a different configuration:

```
conan install . -o shared=True -c tools.cmake.cmake_layout:build_folder_vars='["settings.
↪compiler", "options.shared"]'
```

The values assigned by the `cmake_layout` (installing the Release/shared configuration) would be:

- **`conanfile.folders.build`:**
  - `build-apple-clang-shared_true`: if the cmake generator is multi-configuration.

  - `cmake-build-debug-apple-clang-shared_true`: if the cmake generator is single-configuration.
- `conanfile.folders.generators`: `build-apple-clang-shared_true/generators`

So we can keep separated folders for any number of different configurations that we want to install.

The `CMakePresets.json` file generated at the *CMakeToolchain* generator, will also take this `tools.cmake.cmake_layout:build_folder_vars` config into account to generate different names for the presets, being very handy to install N configurations and building our project for any of them by selecting the chosen preset.

conan.tools.gnu

Warning: These tools are still **experimental** (so subject to breaking changes) but with very stable syntax. We encourage the usage of it to be prepared for Conan 2.0.

AutotoolsDeps

Warning: These tools are still **experimental** (so subject to breaking changes) but with very stable syntax. We encourage the usage of it to be prepared for Conan 2.0.

The `AutotoolsDeps` is the dependencies generator for Autotools. It will generate shell scripts containing environment variable definitions that the autotools build system can understand.

Important: This class will require very soon to define both the "host" and "build" profiles. It is very recommended to start defining both profiles immediately to avoid future breaking. Furthermore, some features, like trying to cross-compile might not work at all if the "build" profile is not provided.

The `AutotoolsDeps` generator can be used by name in conanfiles:

Listing 11: conanfile.py

```
class Pkg(ConanFile):
    generators = "AutotoolsDeps"
```

Listing 12: conanfile.txt

```
[generators]
AutotoolsDeps
```

And it can also be fully instantiated in the conanfile `generate()` method:

```
from conans import ConanFile
from conan.tools.gnu import AutotoolsDeps

class App(ConanFile):
    settings = "os", "arch", "compiler", "build_type"

    def generate(self):
        tc = AutotoolsDeps(self)
        tc.generate()
```

The `AutotoolsDeps` will generate after a `conan install` command the *conanautotoolsdeps.sh* or *conanautotoolsdeps.bat* files:

```
$ conan install conanfile.py # default is Release
$ source conanautotoolsdeps.sh
# or in Windows
$ conanautotoolsdeps.bat
```

This generator will define aggregated variables `CPPFLAGS`, `LIBS`, `LDFLAGS`, `CXXFLAGS`, `CFLAGS` that accumulate all dependencies information, including transitive dependencies, with flags like `-I<path>`, `-L<path>`, etc.

At this moment, only the `requires` information is generated, the `tool_requires` one is not managed by this generator yet.

Attributes

- **environment** : *Environment* object containing the computed variables. If you need to modify some of the computed values you can access to the `environment` object.

```
from conans import ConanFile
from conan.tools.gnu import AutotoolsDeps

class App(ConanFile):
    settings = "os", "arch", "compiler", "build_type"

    def generate(self):
        tc = AutotoolsDeps(self)
        tc.environment.remove("CPPFLAGS", "undesired_value")
        tc.environment.append("CPPFLAGS", "var")
        tc.environment.define("OTHER", "cat")
        tc.environment.unset("LDFLAGS")
        tc.generate()
```

AutotoolsToolchain

Warning: These tools are still **experimental** (so subject to breaking changes) but with very stable syntax. We encourage the usage of it to be prepared for Conan 2.0.

The `AutotoolsToolchain` is the toolchain generator for Autotools. It will generate shell scripts containing environment variable definitions that the autotools build system can understand.

Important: This class will require very soon to define both the "host" and "build" profiles. It is very recommended to start defining both profiles immediately to avoid future breaking. Furthermore, some features, like trying to cross-compile might not work at all if the "build" profile is not provided.

The `AutotoolsToolchain` generator can be used by name in conanfiles:

Listing 13: conanfile.py

```
class Pkg(ConanFile):
    generators = "AutotoolsToolchain"
```

Listing 14: conanfile.txt

```
[generators]
AutotoolsToolchain
```

And it can also be fully instantiated in the conanfile `generate()` method:

```
from conans import ConanFile
from conan.tools.gnu import AutotoolsToolchain

class App(ConanFile):
    settings = "os", "arch", "compiler", "build_type"

    def generate(self):
        tc = AutotoolsToolchain(self)
        tc.generate()
```

The `AutotoolsToolchain` will generate after a `conan install` command the *conanautotoolstoolchain.sh* or *conanautotoolstoolchain.bat* files:

```
$ conan install conanfile.py # default is Release
$ source conanautotoolstoolchain.sh
# or in Windows
$ conanautotoolstoolchain.bat
```

This generator will append information to the `CPPFLAGS`, `LDFLAGS`, `CXXFLAGS`, `CFLAGS` environment variables that translate the settings and options to the corresponding build flags like `-stdlib=libstdc++`, `-std=gnu14`, architecture flags, etc. It will also append the folder where the Conan generators are located to the `PKG_CONFIG_PATH` environment variable.

This generator will also generate a file called `conanbuild.conf` containing two keys:

- **configure_args**: Arguments to call the `configure` script.
- **make_args**: Arguments to call the `make` script.
- **autoreconf_args**: Arguments to call the `autoreconf` script.

The *Autotools build helper* will use that `conanbuild.conf` file to seamlessly call the configure and make script using these precalculated arguments.

It supports the following methods and attributes:

constructor

```
def __init__(self, conanfile, namespace=None):
```

- `conanfile`: the current recipe object. Always use `self`.
- `namespace`: this argument avoids collisions when you have multiple toolchain calls in the same recipe. By setting this argument, the *conanbuild.conf* file used to pass information to the build helper will be named as: *<namespace>_conanbuild.conf*. The default value is `None` meaning that the name of the generated file is *conanbuild.conf*. This namespace must be also set with the same value in the constructor of the *Autotools build helper* so that it reads the information from the proper file.

Attributes

You can change some attributes before calling the `generate()` method if you want to change some of the precalculated values:

```
from conans import ConanFile
from conan.tools.gnu import AutotoolsToolchain

class App(ConanFile):
    settings = "os", "arch", "compiler", "build_type"

    def generate(self):
        tc = AutotoolsToolchain(self)
        tc.configure_args.append("--my_argument")
        tc.generate()
```

- **configure_args: Additional arguments to be passed to the configure script.**
  - **By default the following arguments are passed:**
    - `--prefix`: With the self.package_folder value.
    - `--bindir=${prefix}/bin`
    - `--sbindir=${prefix}/bin`
    - `--libdir=${prefix}/lib`
    - `--includedir=${prefix}/include`
    - `--oldincludedir=${prefix}/include`
    - `--datarootdir=${prefix}/res`
  - **Also if the shared option exists it will add by default:**
    - `--enable-shared`, `--disable-static` if `shared==True`
    - `--disable-shared`, `--enable-static` if `shared==False`
- **make_args** (Defaulted to `[]`): Additional arguments to be passed to he make script.
- **autoreconf_args** (Defaulted to `["--force", "--install"]`): Additional arguments to be passed to he make script.
- **defines** (Defaulted to `[]`): Additional defines.
- **cxxflags** (Defaulted to `[]`): Additional cxxflags.

- **cflags** (Defaulted to `[]`): Additional cflags.
- **ldflags** (Defaulted to `[]`): Additional ldflags.
- **ndebug**: "NDEBUG" if the `settings.build_type` != *Debug*.
- **gcc_cxx11_abi**: "_GLIBCXX_USE_CXX11_ABI" if `gcc/libstdc++`.
- **libcxx**: Flag calculated from `settings.compiler.libcxx`.
- **fpic**: True/False from `options.fpic` if defined.
- **cppstd**: Flag from `settings.compiler.cppstd`
- **arch_flag**: Flag from `settings.arch`
- **build_type_flags**: Flags from `settings.build_type`
- **sysroot_flag**: To pass the `--sysroot` flag to the compiler.
- **apple_arch_flag**: Only when cross-building with Apple systems. Flags from `settings.arch`.
- **apple_isysroot_flag**: Only when cross-building with Apple systems. Path to the root sdk.
- **msvc_runtime_flag**: Flag from `settings.compiler.runtime_type` when compiler is `msvc` or `settings.compiler.runtime` when using the deprecated `Visual Studio`.

If you want to change the default values for `configure_args`, adjust the `cpp.package` object at the `layout()` method:

```
def layout(self):
    ...
    # For bindir and sbindir takes the first value:
    self.cpp.package.bindirs = ["mybin"]
    # For libdir takes the first value:
    self.cpp.package.libdirs = ["mylib"]
    # For includedir and oldincludedir takes the first value:
    self.cpp.package.includedirs = ["myinclude"]
    # For datarootdir takes the first value:
    self.cpp.package.resdirs = ["myres"]
```

Note: It is **not valid** to change the self.cpp_info at the `package_info()` method.

conf

`AutotoolsToolchain` is affected by these *[conf]* variables:

- `tools.build:cxxflags` list of extra C++ flags that will be used by `CXXFLAGS`.
- `tools.build:cflags` list of extra of pure C flags that will be used by `CFLAGS`.
- `tools.build:sharedlinkflags` list of extra linker flags that will be used by `LDFLAGS`.
- `tools.build:exelinkflags` list of extra linker flags that will be used by by `LDFLAGS`.
- `tools.build:defines` list of preprocessor definitions that will be used by `CPPFLAGS`.
- `tools.build:sysroot` defines the `--sysroot` flag to the compiler.

Customizing the environment

If your `Makefile` or `configure` scripts need some other environment variable rather than `CPPFLAGS`, `LDFLAGS`, `CXXFLAGS` or `CFLAGS`, you can customize it before calling the `generate()` method. Call the `environment()` method to calculate the mentioned variables and then add the variables that you need. The `environment()` method returns an *Environment* object:

```
from conans import ConanFile
from conan.tools.gnu import AutotoolsToolchain

class App(ConanFile):
    settings = "os", "arch", "compiler", "build_type"

    def generate(self):
        at = AutotoolsToolchain(self)
        env = at.environment()
        env.define("FOO", "BAR")
        at.generate(env)
```

Autotools

Warning: These tools are still **experimental** (so subject to breaking changes) but with very stable syntax. We encourage the usage of it to be prepared for Conan 2.0.

The `Autotools` build helper is a wrapper around the command line invocation of autotools. It will abstract the calls like `./configure` or `make` into Python method calls.

The `Autotools` helper can be used like:

```
from conans import conanfile
from conan.tools.gnu import Autotools

class App(ConanFile):
    settings = "os", "arch", "compiler", "build_type"

    def build(self):
        autotools = Autotools(self)
        autotools.configure()
        autotools.make()
```

It will read the `conanbuild.conf` file generated by the *AutotoolsToolchain* to know read the arguments for calling the configure and make scripts:

- **configure_args**: Arguments to call the `configure` script.
- **make_args**: Arguments to call the `make` script.

Methods

constructor

```
def __init__(self, conanfile, namespace=None):
```

- `conanfile`: the current recipe object. Always use `self`.
- `namespace`: this argument avoids collisions when you have multiple toolchain calls in the same recipe. By setting this argument, the *conanbuild.conf* file used to pass information to the toolchain will be named as: *<namespace>_conanbuild.conf*. The default value is `None` meaning that the name of the generated file is *conanbuild.conf*. This namespace must be also set with the same value in the constructor of the *AutotoolsToolchain* so that it reads the information from the proper file.

configure(self, build_script_folder=None, args=None):

```
def configure(self):
```

Call the configure script.

- **`build_script_folder` (Optional, Defaulted to `None`): Subfolder where the configure script is located.** If `None`, `conanfile.source_folder` will be used.
- **args (Optional, Defaulted to `None`): List of arguments to use for the** `configure` call.

autoreconf(self, args=None):

```
def autoreconf(self, target=None)
```

Call the `autoreconf` program.

Parameters:

- **target** (Optional, Defaulted to `None`): Choose which target to build. This allows building of e.g., docs, shared libraries or install for some AutoTools projects.
- **args (Optional, Defaulted to `None`): List of arguments to use for the** `autoreconf` call.

make(self, target=None, args=None):

```
def make(self, target=None)
```

Call the `make` program.

Parameters:

- **target** (Optional, Defaulted to `None`): Choose which target to build. This allows building of e.g., docs, shared libraries or install for some AutoTools projects.
- **args** (Optional, Defaulted to `None`): List of arguments to use for the `make` call. By default an argument `DESTDIR=self.package_folder` is added to the call if the passed value is `None`.

install(self, args=None):

```
def install(self)
```

This is just an "alias" of `self.make(target="install")`

A note about relocatable shared libraries in macOS built the Autotools build helper

When building a shared library with Autotools in macOS a section `LC_ID_DYLIB` and another `LC_LOAD_DYLIB` are added to the `.dylib`. These sections store `install_name` information, which is the location of the folder where the library or its dependencies are installed. You can check the install_name of your shared libraries using the otool command:

```
$ otool -l path/to/libMyLib.dylib
...
cmd LC_ID_DYLIB
    cmdsize 48
        name path/to/libMyLib.dylib (offset 24)
time stamp 1 Thu Jan  1 01:00:01 1970
    current version 0.0.0
compatibility version 0.0.0
...
Load command 11
        cmd LC_LOAD_DYLIB
    cmdsize 48
        name path/to/dependency.dylib (offset 24)
time stamp 2 Thu Jan  1 01:00:02 1970
    current version 1.0.0
compatibility version 1.0.0
...
```

Why is this a problem when using Conan?

When using Conan the library will be built in the local cache and this means that this location will point to Conan's local cache folder where the library was installed. This location is where the library tells any other binaries using it where to load it at runtime. This is a problem since you can build the shared library in one machine, then upload it to a server and install it in another machine to use it. In this case, as Autotools behaves by default, you would have a library storing an `install_name` pointing to a folder that does not exist in your current machine so you would get linker errors when building.

How to adress this problem in Conan

The only thing Conan can do to make these shared libraries relocatable is to patch the built binaries after installation. To do this, when using the `Autotools` build helper and after running the Makefile's `install()` step, you can use the *fix_apple_shared_install_name()* tool to search for the built `.dylib` files and patch them by running the `install_name_tool` macOS utility, like this:

```
from conan.tools.apple import fix_apple_shared_install_name
class HelloConan(ConanFile):
```

(continues on next page)

(continued from previous page)

```
    ...
    def package(self):
        autotools = Autotools(self)
        autotools.install()
        fix_apple_shared_install_name(self)
```

This will change the value of the `LC_ID_DYLIB` and `LC_LOAD_DYLIB` sections in the `.dylib` file to:

```
$ otool -l path/to/libMyLib.dylib
...
cmd LC_ID_DYLIB
    cmdsize 48
        name @rpath/libMyLib.dylib (offset 24)
time stamp 1 Thu Jan  1 01:00:01 1970
    current version 0.0.0
compatibility version 0.0.0
...
Load command 11
        cmd LC_LOAD_DYLIB
    cmdsize 48
        name @rpath/dependency.dylib (offset 24)
time stamp 2 Thu Jan  1 01:00:02 1970
    current version 1.0.0
compatibility version 1.0.0
...
```

The `@rpath` special keyword will tell the loader to search a list of paths to find the library. These paths can be defined by the consumer of that library by defining the `LC_RPATH` field. This is done by passing the `-Wl,-rpath -Wl,/path/to/libMyLib.dylib` linker flag when building the consumer of the library. Then if Conan builds an executable that consumes the `libMyLib.dylib` library, it will automatically add the `-Wl,-rpath -Wl,/path/to/libMyLib.dylib` flag so that the library is correctly found when building.

PkgConfigDeps

Warning: These tools are still **experimental** (so subject to breaking changes) but with very stable syntax. We encourage the usage of it to be prepared for Conan 2.0.

PkgConfigDeps

Available since: 1.38.0

The `PkgConfigDeps` is the dependencies generator for pkg-config. Generates pkg-config files named *<PKG-NAME>.pc* (where `<PKG-NAME` is the name declared by dependencies in `cpp_info.names["pkg_config"]` if specified), containing a valid pkg-config file syntax. Indeed, it can also be defined using `set_property` and the property `pkg_config_name` (available since Conan 1.36), for instance:

```
self.cpp_info.components["mycomponent"].set_property("pkg_config_name", "mypkg-config-
↪name")
```

Note: In Conan 2.0 that will be the default way of setting those properties and also passing custom properties to generators. Check the *cpp_info attributes reference* for more information.

The `prefix` variable is automatically adjusted to the `package_folder`.

The `PkgConfigDeps` generator can be used by name in conanfiles:

Listing 15: conanfile.py

```
class Pkg(ConanFile):
    generators = "PkgConfigDeps"
```

Listing 16: conanfile.txt

```
[generators]
PkgConfigDeps
```

And it can also be fully instantiated in the conanfile `generate()` method:

```
from conans import ConanFile
from conan.tools.gnu import PkgConfigDeps

class App(ConanFile):
    settings = "os", "arch", "compiler", "build_type"
    requires = "zlib/1.2.11"

    def generate(self):
        pc = PkgConfigDeps(self)
        pc.generate()
```

The `PkgConfigDeps` will generate the `*.pc` file after a `conan install` command:

```
$ conan install .
# Check the [PC_FILE_NAME].pc created in your current folder
```

Now, running this command using the previous `conanfile.py`, you can check the `zlib.pc` file created into your current folder:

```
prefix=/Users/YOUR_USER/.conan/data/zlib/1.2.11/_/_/package/
↪647afeb69d3b0a2d3d316e80b24d38c714cc6900
libdir=${prefix}/lib
includedir=${prefix}/include

Name: zlib
Description: Conan package: zlib
Version: 1.2.11
Libs: -L"${libdir}" -lz -F Frameworks
Cflags: -I"${includedir}"
```

Components

If a recipe uses *components*, the files generated will be *<[PKG-NAME]-[COMP-NAME]>.pc* with their corresponding flags and require relations.

Additionally, a *<PKG-NAME>.pc* is generated to maintain compatibility for consumers with recipes that start supporting components. This *<PKG-NAME>.pc* file will declare all the components of the package as requires while the rest of the fields will be empty, relying on the propagation of flags coming from the components *<[PKG-NAME]-[COMP-NAME]>.pc* files.

Properties

The following properties affect the `PkgConfigDeps` generator:

- **pkg_config_name** property will define the name of the generated `*.pc` file (`xxxxx.pc`)
- **pkg_config_aliases** property sets some aliases of any package/component name for *pkg_config* generator. This property only accepts list-like Python objects.
- **pkg_config_custom_content** property will add user defined content to the *.pc* files created by this generator.
- **component_version** property sets a custom version to be used in the `Version` field belonging to the created `*.pc` file for that component.

These properties can be defined at global `cpp_info` level or at component level.

Example:

```
def package_info(self):
    custom_content = "datadir=${prefix}/share"
    self.cpp_info.set_property("pkg_config_custom_content", custom_content)
    self.cpp_info.set_property("pkg_config_name", "myname")
    self.cpp_info.components["mycomponent"].set_property("pkg_config_name",
↪"componentname")
    self.cpp_info.components["mycomponent"].set_property("pkg_config_aliases", ["alias1",
↪ "alias2"])
    self.cpp_info.components["mycomponent"].set_property("component_version", "1.14.12")
```

Names and aliases

Aliases are available since: 1.43.0

By default, the `*.pc` files will be named following these rules:

- For packages, it uses the package name, e.g., package `zlib/1.2.11` -> `zlib.pc`.
- For components, the package name + hyphen + component name, e.g., `openssl/3.0.0` with `self.cpp_info.components["crytpo"]` -> `openssl-crypto.pc`.

You can change that default behavior with the `pkg_config_name` and `pkg_config_aliases` properties. For instance, `openssl/3.0.0`` recipe has these `pkg_config_name` properties already declared:

```
from conans import ConanFile

class OpenSSLConan(ConanFile):
    name = "openssl"
```

(continues on next page)

(continued from previous page)

```
    # any code here

    def package_info(self):
        self.cpp_info.set_property("pkg_config_name", "openssl")
        self.cpp_info.components["crypto"].set_property("pkg_config_name", "libcrypto")
        self.cpp_info.components["ssl"].set_property("pkg_config_name", "libssl")
```

Run **conan install openssl/3.0.0@ -g PkgConfigDeps** and check the *.pc files created:

- libcrypto.pc
- libssl.pc
- openssl.pc
- zlib.pc *(openssl requires zlib)*

Their pkg_config_name properties are used as the final *.pc file names:

Listing 17: openssl.pc

```
Name: openssl
Description: Conan package: openssl
Version: 3.0.0
Requires: libcrypto libssl
```

Listing 18: libcrypto.pc

```
prefix=/Users/conan_user/.conan/data/openssl/3.0.0/_/_/package/
↪88955cec2844f731470e07bd44ce5a3a24ec88b7
libdir1=${prefix}/lib
includedir1=${prefix}/include

Name: libcrypto
Description: Conan component: libcrypto
Version: 3.0.0
Libs: -L"${libdir1}" -lcrypto -F Frameworks
Cflags: -I"${includedir1}"
Requires: zlib
```

A special mention when a component shares the same *.pc file name as the root package one:

```
from conans import ConanFile

class OpenCLConan(ConanFile):

    # ...

    def package_info(self):
        self.cpp_info.set_property("pkg_config_name", "OpenCL")  # -> OpenCL.pc
        self.cpp_info.components["_opencl-headers"].set_property("pkg_config_name",
↪"OpenCL")  # -> OpenCL.pc
```

The only *.pc file created will be the one belonging to the component:

- OpenCL.pc (from component)

Now, let's see how `pkg_config_aliases` property works step by step.

Let's create our own `myopenssl/1.0.0` recipe and define several aliases like the following:

```
from conans import ConanFile

class MyOpenSSLConan(ConanFile):
    name = "myopenssl"
    version = "1.0.0"

    def package_info(self):
        # Aliases
        self.cpp_info.set_property("pkg_config_aliases", ["myopenssl_alias"])
        self.cpp_info.components["mycrypto"].set_property("pkg_config_aliases", [
↪"mycrypto", "crp"])
        self.cpp_info.components["myssl"].set_property("pkg_config_aliases", ["myssl"])
```

Then, after creating the package locally with **conan create .** and consuming it **conan install myopenssl/1.0.0@ -g PkgConfigDeps**, the files created will be:

- myopenssl-mycrypto.pc
- myopenssl-myssl.pc
- myopenssl.pc
- crp.pc *(alias of myopenssl-mycrypto)*
- mycrypto.pc *(alias of myopenssl-mycrypto)*
- myssl.pc *(alias of myopenssl-myssl)*
- myopenssl_alias.pc *(alias of myopenssl)*

Where any of those aliases files contains something like this:

Listing 19: mycrypto.pc

```
Name: mycrypto
Description: Alias mycrypto for myopenssl-mycrypto
Version: 1.0.0
Requires: myopenssl-mycrypto
```

It's also possible to use both properties together:

```
from conans import ConanFile

class MyOpenSSLConan(ConanFile):
    name = "myopenssl"
    version = "1.0.0"

    # any code here

    def package_info(self):
        self.cpp_info.set_property("pkg_config_name", "myopenssl")
        self.cpp_info.components["mycrypto"].set_property("pkg_config_name", "libmycrypto
↪")
```

(continues on next page)

(continued from previous page)

```
        self.cpp_info.components["myssl"].set_property("pkg_config_name", "libmyssl")
        # Aliases
        self.cpp_info.set_property("pkg_config_aliases", ["myopenssl_alias"])
        self.cpp_info.components["mycrypto"].set_property("pkg_config_aliases", [
↪"mycrypto", "crp"])
        self.cpp_info.components["myssl"].set_property("pkg_config_aliases", ["myssl"])
```

After executing the commands mentioned above, the files are:

- libmycrypto.pc
- libmyssl.pc
- myopenssl.pc
- crp.pc *(alias of libmycrypto)*
- mycrypto.pc *(alias of libmycrypto)*
- myssl.pc *(alias of libmyssl)*
- myopenssl_alias.pc *(alias of myopenssl)*

The only change is which name the alias is pointing to:

Listing 20: mycrypto.pc

```
Name: mycrypto
Description: Alias mycrypto for libmycrypto
Version: 1.0.0
Requires: libmycrypto
```

PkgConfig

Warning: These tools are still **experimental** (so subject to breaking changes) but with very stable syntax. We encourage the usage of it to be prepared for Conan 2.0.

Available since: 1.43.0

This tool can execute `pkg_config` executable to extract information from existing `.pc` files. This can be useful for example to create a "system" package recipe over some system installed library, as a way to automatically extract the `.pc` information from the system. Or if some proprietary package has a build system that only outputs `.pc` files.

The constructor is:

```
def __init__(self, conanfile, library, pkg_config_path=None):
```

- `conanfile`: The current `self` instance of the conanfile using the tool
- `library`: The library which `.pc` file is to be parsed. It must exist in the pkg_config path
- `pkg_config_path`: If defined it will be prepended to `PKG_CONFIG_PATH` environment variable, so the execution finds the required files.

It can be used as:

```
pkg_config = PkgConfig(conanfile, "libastral", pkg_config_path=<somedir>)

print(pkg_config.provides) # something like"libastral = 6.6.6"
print(pkg_config.version) # something like"6.6.6"
print(pkg_config.includedirs) # something like['/usr/local/include/libastral']
print(pkg_config.defines) # something like['_USE_LIBASTRAL']
print(pkg_config.libs) # something like['astral', 'm']
print(pkg_config.libdirs) # something like['/usr/local/lib/libastral']
print(pkg_config.linkflags) # something like['-Wl,--whole-archive']
print(pkg_config.variables['prefix']) # something like'/usr/local'
```

There is a convenience method `fill_cpp_info()`, that can be used in the `package_info()` method as:

```
def package_info(self):
    pkg_config = PkgConfig(conanfile, "libastral", pkg_config_path=tmp_dir)
    pkg_config.fill_cpp_info(self.cpp_info, is_system=False, system_libs=["m", "rt"])
```

Where:

- `cpp_info` first argument could be the global one or a component one.
- `is_system`: if `True`, all detected libraries will be assigned to `cpp_info.system_libs`, and none to `cpp_info.libs`.
- `system_libs`: If `is_system=False`, this argument allows defining some potential system libraries found that would be assigned to `cpp_info.system_libs`. The remaining detected libs will be assigned to `cpp_info.libs`.

conf

This helper will listen to `tools.gnu:pkg_config` *configuration* to define the `pkg_config` executable name or full path. It will by default it is `pkg-config`.

conan.tools.google

Warning: These tools are **experimental** and subject to breaking changes.

BazelDeps

Available since: 1.37.0

The `BazelDeps` helper will generate one **conandeps/xxxx/BUILD** file per dependency. This dependencies will be automatically added to the project if you add the following lines to the project's **WORKSPACE** file:

```
load("@//conandeps:dependencies.bzl", "load_conan_dependencies")
load_conan_dependencies()
```

The dependencies should be added to the **conanfile.py** file as usual:

```
class BazelExampleConan(ConanFile):
    name = "bazel-example"
```

(continues on next page)

(continued from previous page)

```
    ...
    generators = "BazelDeps", "BazelToolchain"
    requires = "boost/1.76.0"
```

BazelToolchain

The `BazelToolchain` is the toolchain generator for Bazel. It will generate a file called `conanbuild.conf` containing two keys:

- **bazelrc_path**: defining Bazel rc-path. Can be set using the conf `tools.google.bazel:bazelrc_path`.
- **bazel_configs**: defining the configs to be activated in the `bazelrc_path`. Can be set with the conf `tools.google.bazel:configs`.

Listing 21: **Example of a custom bazelrc file at '/path/to/mybazelrc':**

```
build:Release -c opt
build:RelWithDebInfo -c opt --copt=-O3 --copt=-DNDEBUG
build:MinSizeRel  -c opt --copt=-Os --copt=-DNDEBUG
build --color=yes
build:withTimeStamps --show_timestamps
```

Listing 22: **Example of a Release profile:**

```
[settings]
...
build_type=Release

[conf]
tools.google.bazel:bazelrc_path=/path/to/mybazelrc
tools.google.bazel:configs=["Release"]
```

The Bazel build helper will use that `conanbuild.conf` file to seamlessly call the configure and make script using these precalculated arguments. Note that the file can have a different name if you set the namespace argument in the constructor as explained bellow.

It supports the following methods and attributes:

constructor

```
def __init__(self, conanfile, namespace=None):
```

- `conanfile`: the current recipe object. Always use `self`.
- `namespace`: this argument avoids collisions when you have multiple toolchain calls in the same recipe. By setting this argument, the *conanbuild.conf* file used to pass information to the build helper will be named as: *<namespace>_conanbuild.conf*. The default value is `None` meaning that the name of the generated file is *conanbuild.conf*. This namespace must be also set with the same value in the constructor of the `Bazel` build helper so that it reads the information from the proper file.

Bazel

The `Bazel` build helper is a wrapper around the command line invocation of bazel. It will abstract the calls like `bazel build //main:hello-world` into Python method calls.

The helper is intended to be used in the `build()` method, to call Bazel commands automatically when a package is being built directly by Conan (create, install)

```
from conans import ConanFile
from conan.tools.google import Bazel

class App(ConanFile):
    settings = "os", "arch", "compiler", "build_type"

    def build(self):
        bazel = Bazel(self)
        bazel.configure()
        bazel.build(label="//main:hello-world")
```

It supports the following methods:

constructor

```
def __init__(self, conanfile, namespace=None):
```

- `conanfile`: the current recipe object. Always use `self`.
- `namespace`: this argument avoids collisions when you have multiple toolchain calls in the same recipe. By setting this argument, the *conanbuild.conf* file used to pass information to the toolchain will be named as: *<namespace>_conanbuild.conf*. The default value is `None` meaning that the name of the generated file is *conanbuild.conf*. This namespace must be also set with the same value in the constructor of `BazelToolchain` so that it reads the information from the proper file.

build()

```
def build(self, args=None, label=None):
```

Calls the build system. Equivalent to **`bazel build {label}`** in the build folder.

conan.tools.apple

> **Warning:** These tools are still **experimental** (so subject to breaking changes) but with very stable syntax. We encourage the usage of it to be prepared for Conan 2.0.

XcodeDeps

Available since: 1.42.0

The `XcodeDeps` tool is the dependency information generator for *Xcode*. It will generate multiple *.xcconfig* configuration files, the can be used by consumers using *xcodebuild* or *Xcode*. To use them just add the generated configuration files to the Xcode project or set the `-xcconfig` argument from the command line.

The `XcodeDeps` generator can be used by name in conanfiles:

Listing 23: conanfile.py

```
class Pkg(ConanFile):
    generators = "XcodeDeps"
```

Listing 24: conanfile.txt

```
[generators]
XcodeDeps
```

And it can also be fully instantiated in the conanfile `generate()` method:

Listing 25: conanfile.py

```
from conan import ConanFile
from conan.tools.apple import XcodeDeps

class Pkg(ConanFile):
    settings = "os", "compiler", "arch", "build_type"
    requires = "libpng/1.6.37@" # Note libpng has zlib as transitive dependency

    def generate(self):
        xcode = XcodeDeps(self)
        xcode.generate()
```

When the `XcodeDeps` generator is used, every invocation of `conan install` will generate several configuration files, per dependency and configuration. For the *conanfile.py* above, for example:

```
$ conan install conanfile.py # default is Release
$ conan install conanfile.py -s build_type=Debug
```

This generator is multi-configuration. It will generate different files for the different *Debug/Release* configurations for each requirement. It will also generate one single file (*conandeps.xcconfig*) aggregating all the files for the direct dependencies (just *libpng* in this case). The above commands generate the following files:

```
.
├── conan_libpng.xcconfig
├── conan_libpng_debug_x86_64.xcconfig
├── conan_libpng_release_x86_64.xcconfig
├── conan_libpng_vars_debug_x86_64.xcconfig
├── conan_libpng_vars_release_x86_64.xcconfig
├── conan_zlib.xcconfig
├── conan_zlib_debug_x86_64.xcconfig
├── conan_zlib_release_x86_64.xcconfig
├── conan_zlib_vars_debug_x86_64.xcconfig
├── conan_zlib_vars_release_x86_64.xcconfig
├── conandeps.xcconfig
└── conan_config.xcconfig
```

The first `conan install` with the default *Release* and *x86_64* configuration generates:

- *conan_libpng_vars_release_x86_64.xcconfig*: declares some intermediate variables that are included in *conan_libpng_release_x86_64.xcconfig*
- *conan_libpng_release_x86_64.xcconfig*: includes *conan_libpng_vars_release_x86_64.xcconfig* and declares variables with conditional logic to be considered only for the active configuration in *Xcode* or the one passed by command line to *xcodebuild*.
- *conan_libpng.xcconfig*: includes *conan_libpng_release_x86_64.xcconfig* and declares the following *Xcode* build settings: `HEADER_SEARCH_PATHS`, `GCC_PREPROCESSOR_DEFINITIONS`, `OTHER_CFLAGS`, `OTHER_CPLUSPLUSFLAGS`, `FRAMEWORK_SEARCH_PATHS`, `LIBRARY_SEARCH_PATHS`, `OTHER_LDFLAGS`. It also includes the generated *xcconfig* files for transitive dependencies (*conan_zlib.xcconfig* in this case).
- Same 3 files will be generated for each dependency in the graph. In this case, as *zlib* is a dependency of *libpng* it will generate: *conan_zlib_vars_release_x86_64.xcconfig*, *conan_zlib_release_x86_64.xcconfig* and *conan_zlib.xcconfig*.
- *conandeps.xcconfig*: configuration files including all direct dependencies, in this case, it just includes `conan_libpng.xcconfig`.
- The main *conan_config.xcconfig* file, to be added to the project. Includes both the files from this generator and the generated by the *XcodeToolchain* in case it was also set.

The second `conan install -s build_type=Debug` generates:

- *conan_libpng_vars_debug_x86_64.xcconfig*: same variables as the one below for *Debug* configuration.
- *conan_libpng_debug_x86_64.xcconfig*: same variables as the one below for *Debug* configuration.
- *conan_libpng.xcconfig*: this file has been already creted by the previous command, now it's modified to add the include for *conan_libpng_debug_x86_64.xcconfig*.
- Like in the previous command the same 3 files will be generated for each dependency in the graph. In this case, as *zlib* is a dependency of *libpng* it will generate: *conan_zlib_vars_debug_x86_64.xcconfig*, *conan_zlib_debug_x86_64.xcconfig* and *conan_zlib.xcconfig*.
- *conandeps.xcconfig*: configuration files including all direct dependencies, in this case, it just includes `conan_libpng.xcconfig`.
- The main *conan_config.xcconfig* file, to be added to the project. Includes both the files from this generator and the generated by the *XcodeToolchain* in case it was also set.

If you want to add this dependencies to you Xcode project, you just have to add the *conan_config.xcconfig* configuration file for all of the configurations you want to use (usually *Debug* and *Release*).

Components support

Since Conan version 1.49.0, this generator supports packages with components. That means that:

- If a **dependency** `package_info()` declares `cpp_info.requires` on some components, the generated *.xcconfig* files will contain includes to only those components.
- The current package `requires` will be fully dependent on and all components. Recall that the `package_info()` only applies for consumers, but not to the current package.

Custom configurations

If your Xcode project defines custom configurations, like `ReleaseShared`, or `MyCustomConfig`, it is possible to define it into the `XcodeDeps` generator, so different project configurations can use different set of dependencies. Let's say that our current project can be built as a shared library, with the custom configuration `ReleaseShared`, and the package also controls this with the `shared` option:

```
from conan import ConanFile
from conan.tools.apple import XcodeDeps

class Pkg(ConanFile):
    settings = "os", "compiler", "arch", "build_type"
    options = {"shared": [True, False]}
    default_options = {"shared": False}
    requires = "zlib/1.2.11"

    def generate(self):
        xcode = XcodeDeps(self)
        # We assume that -o *:shared=True is used to install all shared deps too
        if self.options.shared:
            xcode.configuration = str(self.settings.build_type) + "Shared"
        xcode.generate()
```

This will manage to generate new *.xcconfig* files for this custom configuration, and when you switch to this configuration in the IDE, the build system will take the correct values depending wether we want to link with shared or static libraries.

XcodeToolchain

Available since: 1.46.0

The `XcodeToolchain` is the toolchain generator for Xcode. It will generate *.xcconfig* configuration files that can be added to Xcode projects. This generator translates the current package configuration, settings, and options, into Xcode *.xcconfig* files syntax.

The `XcodeToolchain` generator can be used by name in conanfiles:

Listing 26: conanfile.py

```
class Pkg(ConanFile):
    generators = "XcodeToolchain"
```

Listing 27: conanfile.txt

```
[generators]
XcodeToolchain
```

And it can also be fully instantiated in the conanfile `generate()` method:

```
from conan import ConanFile
from conan.tools.apple import XcodeToolchain

class App(ConanFile):
    settings = "os", "arch", "compiler", "build_type"
```

(continues on next page)

(continued from previous page)

```
    def generate(self):
        tc = XcodeToolchain(self)
        tc.generate()
```

The `XcodeToolchain` will generate three files after a `conan install` command. As explained above for the XcodeDeps generator, each different configuration will create a set of files with different names. For example, running `conan install` for *Release* first and then *Debug* configuration:

```
$ conan install conanfile.py # default is Release
$ conan install conanfile.py -s build_type=Debug
```

Will create these files:

```
.
├── conan_config.xcconfig
├── conantoolchain_release_x86_64.xcconfig
├── conantoolchain_debug_x86_64.xcconfig
├── conantoolchain.xcconfig
└── conan_global_flags.xcconfig
```

Those files are:

- The main *conan_config.xcconfig* file, to be added to the project. Includes both the files from this generator and the generated by the *XcodeDeps* in case it was also set.
- *conantoolchain_<debug/release>_x86_64.xcconfig*: declares `CLANG_CXX_LIBRARY`, `CLANG_CXX_LANGUAGE_STANDARD` and `MACOSX_DEPLOYMENT_TARGET` variables with conditional logic depending on the build configuration, architecture and sdk set.
- *conantoolchain.xcconfig*: aggregates all the *conantoolchain_<config>_<arch>.xcconfig* files for the different installed configurations.
- *conan_global_flags.xcconfig*: this file will only be generated in case of any configuration variables related to compiler or linker flags are set. Check *the configuration section* below for more details.

Every invocation to `conan install` with different configuration will create a new *conantoolchain_<config>_<arch>.xcconfig* file that is aggregated in the *conantoolchain.xcconfig*, so you can have different configurations included in your Xcode project.

The XcodeToolchain files can declare the following Xcode build settings based on Conan settings values:

- `MACOSX_DEPLOYMENT_TARGET` is based on the value of the `os.version` setting and will make the build system to pass the flag `-mmacosx-version-min` with that value (if set). It defines the operating system version the binary should run into.
- `CLANG_CXX_LANGUAGE_STANDARD` is based on the value of the `compiler.cppstd` setting that sets the C++ language standard.
- `CLANG_CXX_LIBRARY` is based on the value of the compiler.libcxx setting and sets the version of the C++ standard library to use.

One of the advantages of using toolchains is that they can help to achieve the exact same build with local development flows, than when the package is created in the cache.

conf

This toolchain is also affected by these *[conf]* variables:

- `tools.build:cxxflags` list of C++ flags.
- `tools.build:cflags` list of pure C flags.
- `tools.build:sharedlinkflags` list of flags that will be used by the linker when creating a shared library.
- `tools.build:exelinkflags` list of flags that will be used by the linker when creating an executable.
- `tools.build:defines` list of preprocessor definitions.

If you set any of these variables, the toolchain will use them to generate the `conan_global_flags.xcconfig` file that will be included from the `conan_config.xcconfig` file.

XcodeBuild

Available since: 1.46.0

The `Xcode` build helper is a wrapper around the command line invocation of Xcode. It will abstract the calls like `xcodebuild -project app.xcodeproj -configuration <config> -arch <arch> ...`

The `Xcode` helper can be used like:

```
from conan import conanfile
from conan.tools.apple import XcodeBuild

class App(ConanFile):
    settings = "os", "arch", "compiler", "build_type"

    def build(self):
        xcodebuild = XcodeBuild(self)
        xcodebuild.build("app.xcodeproj")
```

Xcode.build() method

```
def build(self, xcodeproj, target=None):
```

- `xcodeproj`: the *xcodeproj* file to build.
- `target`: the target to build, in case this argument is passed to the `build()` method it will add the `-target` argument to the build system call. If not passed, it will build all the targets passing the `-alltargets` argument instead.

The `Xcode.build()` method internally implements a call to `xcodebuild` like:

```
$ xcodebuild -project app.xcodeproj -configuration <configuration> -arch <architecture>
↪<sdk> <verbosity> -target <target>/-alltargets
```

Where:

- `configuration` is the configuration, typically *Release* or *Debug*, which will be obtained from `settings.build_type`.

- `architecture` is the build architecture, a mapping from the `settings.arch` to the common architectures defined by Apple 'i386', 'x86_64', 'armv7', 'arm64', etc.
- `sdk` is set based on the values of the `os.sdk` and `os.sdk_version` defining the `SDKROOT` Xcode build setting according to them. For example, setting `os.sdk=iOS` and *os.sdk_version=8.3* ` will pass `SDKROOT=iOS8.3` to the build system. In case you defined the `tools.apple:sdk_path` in your *[conf]* this value will take preference and will directly pass `SDKROOT=<tools.apple:sdk_path>` so **take into account** that for this case the skd located in that path should set your `os.sdk` and `os.sdk_version` settings values.
- `verbosity` is the verbosity level for the build and can take value 'verbose' or 'quiet' if set by `tools.apple.xcodebuild:verbosity` in your *[conf]*

conf

- `tools.apple.xcodebuild:verbosity` verbosity value for the build, can be 'verbose' or 'quiet'
- `tools.apple:sdk_path` path for the sdk location, will set the `SDKROOT` value with preference over composing the value from the `os.sdk` and `os.sdk_version` settings.

conan.tools.apple.fix_apple_shared_install_name()

```
def fix_apple_shared_install_name(conanfile):
```

Parameters:

- **conanfile**: Conanfile instance.

This tool will search for all the *dylib* files in the conanfile's *package_folder* and fix both the `LC_ID_DYLIB` and `LC_LOAD_DYLIB` fields on those files using the *install_name_tool* utility available in macOS.

- For `LC_ID_DYLIB` which is the field containing the install name of the library, it will change the install name to one that uses the `@rpath`. For example, if the install name is `/path/to/lib/libname.dylib`, the new install name will be `@rpath/libname.dylib`. This is done by executing internally something like:

```
install_name_tool /path/to/lib/libname.dylib -id @rpath/libname.dylib
```

- For `LC_LOAD_DYLIB` which is the field containing the path to the library dependencies, it will change the path of the dependencies to one that uses the `@rpath`. For example, if the path is `/path/to/lib/dependency.dylib`, the new path will be `@rpath/dependency.dylib`. This is done by executing internally something like:

```
install_name_tool /path/to/lib/libname.dylib -change /path/to/lib/dependency.dylib␣
↪@rpath/dependency.dylib
```

This tool is typically needed by recipes that use Autotools as the build system and in the case that the correct install names are not fixed in the library being packaged. Use this tool, if needed, in the conanfile's `package()` method like:

```
from conan.tools.apple import fix_apple_shared_install_name

class HelloConan(ConanFile):

  ...

  def package(self):
      autotools = Autotools(self)
```

(continues on next page)

(continued from previous page)

```
        autotools.install()
        fix_apple_shared_install_name(self)
```

conan.tools.meson

Warning: These tools are **experimental** and subject to breaking changes.

You can use `conan new hello/0.1 --template=meson_lib` and `conan new hello/0.1 --template=meson_exe` templates to try this Meson integration.

MesonToolchain

Warning: This is an **experimental** feature subject to breaking changes in future releases.

Available since: 1.33.0

The `MesonToolchain` can be used in the `generate()` method:

```
from conan import ConanFile
from conan.tools.meson import MesonToolchain

class App(ConanFile):
    settings = "os", "arch", "compiler", "build_type"
    requires = "hello/0.1"
    options = {"shared": [True, False]}
    default_options = {"shared": False}

    def generate(self):
        tc = MesonToolchain(self)
        tc.preprocessor_definitions["MYDEFINE"] = "MYDEF_VALUE"
        tc.generate()
```

The `MesonToolchain` will generate a file: - *conan_meson_native.ini*: if doing a native build. - *conan_meson_cross.ini*: if doing a cross-build (*tools.cross_building()*).

Important: This class will require very soon to define both the "host" and "build" profiles. It is very recommended to start defining both profiles immediately to avoid future breaking. Furthermore, some features, like trying to cross-compile might not work at all if the "build" profile is not provided.

`conan_meson_native.ini` will contain the definitions of all the Meson properties related to the Conan options and settings for the current package, platform, etc. This includes but is not limited to the following:

- Detection of `default_library` from Conan settings
  - Based on existance/value of a option named `shared`
- Detection of `buildtype` from Conan settings
- Definition of the C++ standard as necessary

- The Visual Studio runtime (`b_vscrt`), obtained from Conan input settings

conan_meson_cross.ini contains the same information as *conan_meson_native.ini*, but with additional information to describe host, target, and build machines (such as the processor architecture).

Check out the meson documentation for more details on native and cross files:

- Machine files
- Native environments
- Cross compilation

constructor

```
def __init__(self, conanfile, backend=None):
```

Most of the arguments are optional and will be deduced from the current `settings`, and not necessary to define them.

- `conanfile`: the current recipe object. Always use `self`.
- `backend`: the meson backend to use. By default, `ninja` is used. Possible values: ninja, vs, vs2010, vs2015, vs2017, vs2019, xcode.

project_options

This attribute allows defining Meson project options:

```
def generate(self):
    tc = MesonToolchain(self)
    tc.project_options["MYVAR"] = "MyValue"
    tc.generate()
```

- One project options definition for `MYVAR` in `conan_meson_native.init` or `conan_meson_cross.ini` file.

preprocessor_definitions

This attribute allows defining compiler preprocessor definitions, for multiple configurations (Debug, Release, etc).

```
def generate(self):
    tc = MesonToolchain(self)
    tc.preprocessor_definitions["MYDEF"] = "MyValue"
    tc.generate()
```

This will be translated to:

- One preprocessor definition for `MYDEF` in `conan_meson_native.ini` or `conan_meson_cross.ini` file.

Generators

The `MesonToolchain` only works with the `PkgConfigDeps` generator. Please, do not use other generators, as they can have overlapping definitions that can conflict.

Using the toolchain in developer flow

One of the advantages of using Conan toolchains is that they can help to achieve the exact same build with local development flows, than when the package is created in the cache.

With the `MesonToolchain` it is possible to do:

```
# Lets start in the folder containing the conanfile.py
$ mkdir build && cd build
# Install both debug and release deps and create the toolchain
$ conan install ..
# the build type Release is encoded in the toolchain already.
# This conan_meson_native.iniis specific for release
$ meson setup --native-file conan_meson_native.ini build .
$ meson compile -C build
```

conf

`MesonToolchain` is affected by these *[conf]* variables:

- `tools.meson.mesontoolchain:backend`. the meson backend to use. Possible values: `ninja`, `vs`, `vs2010`, `vs2015`, `vs2017`, `vs2019`, `xcode`.
- `tools.apple:sdk_path` argument for SDK path in case of Apple cross-compilation. It will be used as value of the flag `-isysroot`.
- `tools.android:ndk_path` argument for NDK path in case of Android cross-compilation. It will be used to get some binaries like `c`, `cpp` and `ar` used in `[binaries]` section from *conan_meson_cross.ini*.

Apart from that, since Conan 1.47, you can inject extra flags thanks to these ones:

- `tools.build:cxxflags` list of extra C++ flags that will be used by `cpp_args`.
- `tools.build:cflags` list of extra of pure C flags that will be used by `c_args`.
- `tools.build:sharedlinkflags` list of extra linker flags that will be used by `c_link_args` and `cpp_link_args`.
- `tools.build:exelinkflags` list of extra linker flags that will be used by `c_link_args` and `cpp_link_args`.

Cross-building for Apple and Android

It deserves a special mention because `MesonToolchain` is automatically adding all the flags needed to cross-compile for Apple (MacOS M1, iOS, etc.) and Android.

Apple

It'll add link flags like `-arch XXX`, `-isysroot [SDK_PATH]` and the minimum deployment target flag, e.g., `-mios-version-min=8.0` into Meson `c_args`, `c_link_args`, `cpp_args` and `cpp_link_args` built-in options.

Android

It'll initialize the `c`, `cpp` and `ar` variables which are needed to cross-compile for Android. For instance:

- `c == $TOOLCHAIN/bin/llvm-ar`
- `cpp == $TOOLCHAIN/bin/$TARGET$API-clang`
- `ar == $TOOLCHAIN/bin/$TARGET$API-clang++`

Where:

- `$TOOLCHAIN`: `[NDK_PATH]/toolchains/llvm/prebuilt/[OS_BUILD]-x86_64/bin`.
- `$TARGET`: target triple, e.g., for `armv8` will be `aarch64-linux-android`.
- `$API`: Android API version.

Besides that, you'll always be able to change any of these variables before being applied thanks to the `MesonToolchain` class interface. For instance:

```
from conan import ConanFile
from conan.tools.meson import MesonToolchain

class App(ConanFile):
    settings = "os", "arch", "compiler", "build_type"
    requires = "hello/0.1"
    options = {"shared": [True, False]}
    default_options = {"shared": False}

    def generate(self):
        tc = MesonToolchain(self)
        tc.cpp = "/path/to/other/compiler"
        tc.generate()
```

Meson

Warning: This is an **experimental** feature subject to breaking changes in future releases.

The `Meson()` build helper that works with the `MesonToolchain` is also experimental, and subject to breaking change in the future. It will evolve to adapt and complement the toolchain functionality.

The helper is intended to be used in the `build()` method, to call Meson commands automatically when a package is being built directly by Conan (create, install)

```
from conan.tools.meson import Meson

def build(self):
    meson = Meson(self)
    meson.configure()
    meson.build()
```

It supports the following methods:

constructor

```
def __init__(self, conanfile):
```

- `conanfile`: the current recipe object. Always use `self`.

configure()

```
def configure(self):
```

Calls **`meson`**, with the given generator and passing either **`--native-file conan_meson_native.ini`** (native builds) or **`--cross-file conan_meson_cross.ini`** (cross builds).

build()

```
def build(self, target=None):
```

Calls the build system. Equivalent to **`meson compile -C .`** in the build folder.

Parameters:

- **target** (Optional, Defaulted to `None`): Specifies the target to execute. The default *all* target will be built if `None` is specified.

install()

```
def install(self):
```

Installs development files (headers, libraries, etc.). Equivalent to run **`meson install -C .`** in the build folder.

test()

```
def test(self):
```

Runs project's tests. Equivalent to running **`meson test -v -C .`** in the build folder..

conf

- `tools.build:jobs=10` argument for the `--jobs` parameter when running Ninja.

conan.tools.intel

IntelCC

Available since: 1.41.0

This tool helps you to manage the new Intel oneAPI DPC++/C++ and Classic ecosystem in Conan.

Warning: This generator is **experimental** and subject to breaking changes.

Warning: macOS is not supported for the Intel oneAPI DPC++/C++ (icx/icpx or dpcpp) compilers. For macOS or Xcode support, you'll have to use the Intel C++ Classic Compiler.

Note: Remember, you need to have installed previously the Intel oneAPI software.

Note: If you are using `CMakeToolchain` or `MSBuildToolchain`, you don't need to use this generator. See *intel-cc compiler section* for more information.

This generator creates a `conanintelsetvars.sh|bat` wrapping the Intel script `setvars.sh|bat` that set the Intel oneAPI environment variables needed. That script is the first step to start using the Intel compilers because it's setting some important variables in your local environment.

In summary, the `IntelCC` generator:

1. Reads your profile `[settings]` and `[conf]`.
2. Uses that information to generate a `conanintelsetvars.sh|bat` script with the command to load the Intel `setvars.sh|bat` script.
3. Then, you or the chosen generator will be able to run that script and use any Intel compiler to compile the project.

Note: You can launch the `conanintelsetvars.sh|bat` before calling your intel compiler to build a project. Also, Conan will automatically call it in the conanfile `build(self)` method when running any command with `self.run`.

At first, ensure you are using a *profile* like this one:

Listing 28: intelprofile

```
[settings]
...
compiler=intel-cc
compiler.mode=dpcpp
compiler.version=2021.3
compiler.libcxx=libstdc++
build_type=Release
[options]

[tool_requires]
```

(continues on next page)

(continued from previous page)

```
[env]
CC=dpcpp
CXX=dpcpp

[conf]
tools.intel:installation_path=/opt/intel/oneapi
```

The `IntelCC` generator can be used by name in conanfiles:

Listing 29: conanfile.py

```
class Pkg(ConanFile):
    generators = "IntelCC"
```

Listing 30: conanfile.txt

```
[generators]
IntelCC
```

And it can also be fully instantiated in the conanfile `generate()` method:

Listing 31: conanfile.py

```
from conans import ConanFile
from conan.tools.intel import IntelCC

class App(ConanFile):
    settings = "os", "arch", "compiler", "build_type"

    def generate(self):
        intelcc = IntelCC(self)
        intelcc.generate()
```

Now, running the command **`conan install . -pr intelprofile`** will generate the `conanintelsetvars.sh|bat` script which will run the Intel *setvars* script and load all the variables into your local environment.

Custom configurations

You can apply different installation paths and command arguments simply by changing the `[conf]` entries. For instance:

Listing 32: intelprofile

```
[settings]
...
compiler=intel-cc
compiler.mode=dpcpp
compiler.version=2021.3
compiler.libcxx=libstdc++
build_type=Release
[options]
```

(continues on next page)

(continued from previous page)

```
[tool_requires]
[env]
CC=dpcpp
CXX=dpcpp

[conf]
tools.intel:installation_path=/opt/intel/oneapi
tools.intel:setvars_args=--config="full/path/to/your/config.txt" --force
```

If we run again a **conan install . -pr intelprofile** then the conanintelsetvars.sh script (if we are using Linux OS) will contain something like:

Listing 33: conanintelsetvars.sh

```
. "/opt/intel/oneapi/setvars.sh" --config="full/path/to/your/config.txt" --force
```

conf

These are the two different entries for IntelCC:

- tools.intel:installation_path: **(required)** argument to tells Conan the installation path, if it's not defined, Conan will try to find it out automatically.
- tools.intel:setvars_args: **(optional)** it is used to pass whatever we want as arguments to our *setvars.sh|bat* file. You can check out all the possible ones from the Intel official documentation.

conan.tools.microsoft

These tools allow a native integration for Microsoft Visual Studio, natively (without using CMake, but using directly Visual Studio solutions, projects and property files).

> **Warning:** These tools are still **experimental** (so subject to breaking changes) but with very stable syntax. We encourage the usage of it to be prepared for Conan 2.0.

MSBuildDeps

The MSBuildDeps is the dependency information generator for Microsoft MSBuild build system. It will generate multiple *xxxx.props* properties files one per dependency of a package, to be used by consumers using MSBuild or Visual Studio, just adding the generated properties files to the solution and projects.

Important: This class will require very soon to define both the "host" and "build" profiles. It is very recommended to start defining both profiles immediately to avoid future breaking. Furthermore, some features, like trying to cross-compile might not work at all if the "build" profile is not provided.

It is important to highlight that this one is a **dependencies generator** and it is focused on the **dependencies** of a conanfile, not the current build.

The MSBuildDeps generator can be used by name in conanfiles:

Listing 34: conanfile.py

```
class Pkg(ConanFile):
    generators = "MSBuildDeps"
```

Listing 35: conanfile.txt

```
[generators]
MSBuildDeps
```

And it can also be fully instantiated in the conanfile `generate()` method:

Listing 36: conanfile.py

```
from conans import ConanFile
from conan.tools.microsoft import MSBuildDeps

class Pkg(ConanFile):
    settings = "os", "compiler", "arch", "build_type"
    requires = "zlib/1.2.11", "bzip2/1.0.8"

    def generate(self):
        ms = MSBuildDeps(self)
        ms.generate()
```

When the `MSBuildDeps` generator is used, every invocation of `conan install` will generate properties files, one per dependency and per configuration. For the last *conanfile.py* above:

```
$ conan install conanfile.py # default is Release
$ conan install conanfile.py -s build_type=Debug
```

This is a multi-configuration generator, and will generate different files for the different Debug/Release configuration. The above commands the following files will be generated:

- *conan_zlib_vars_release_x64.props*: `Conanzlibxxxx` variables definitions for the `zlib` dependency, Release config, like `ConanzlibIncludeDirs`, `ConanzlibLibs`, etc.
- *conan_zlib_vars_debug_x64.props*: Same `Conanzlib``variables for ``zlib` dependency, Debug config
- *conan_zlib_release_x64.props*: Activation of `Conanzlibxxxx` variables in the current build as standard C/C++ build configuration, Release config. This file contains also the transitive dependencies definitions.
- *conan_zlib_debug_x64.props*: Same activation of `Conanzlibxxxx` variables, Debug config, also inclusion of transitive dependencies.
- *conan_zlib.props*: Properties file for `zlib`. It conditionally includes, depending on the configuration, one of the two immediately above Release/Debug properties files.
- Same 5 files will be generated for every dependency in the graph, in this case `conan_bzip.props` too, which will conditionally include the Release/Debug bzip properties files.
- *conandeps.props*: Properties files including all direct dependencies, in this case, it includes `conan_zlib.props` and `conan_bzip2.props`

You will be adding the *conandeps.props* to your solution project files if you want to depend on all the declared dependencies. For single project solutions, this is probably the way to go. For multi-project solutions, you might be more efficient and add properties files per project. You could add *conan_zlib.props* properties to "project1" in the solution and *conan_bzip2.props* to "project2" in the solution for example.

Custom configurations

If your Visual Studio project defines custom configurations, like `ReleaseShared`, or `MyCustomConfig`, it is possible to define it into the `MSBuildDeps` generator, so different project configurations can use different set of dependencies. Let's say that our current project can be built as a shared library, with the custom configuration `ReleaseShared`, and the package also controls this with the `shared` option:

```
from conans import ConanFile
from conan.tools.microsoft import MSBuildDeps

class Pkg(ConanFile):
    settings = "os", "compiler", "arch", "build_type"
    options = {"shared": [True, False]}
    default_options = {"shared": False}
    requires = "zlib/1.2.11"

    def generate(self):
        ms = MSBuildDeps(self)
        # We assume that -o *:shared=True is used to install all shared deps too
        if self.options.shared:
            ms.configuration = str(self.settings.build_type) + "Shared"
        ms.generate()
```

This will manage to generate new properties files for this custom configuration, and switching it in the IDE allows to be switching dependencies configuration like Debug/Release, it could be also switching dependencies from static to shared libraries.

Included dependencies

`MSBuildDeps` uses the new experimental `self.dependencies` access to dependencies. The following dependencies will be translated to properties files:

- All direct dependencies, that is, the ones declared by the current `conanfile`, that lives in the host context: all regular `requires`, plus the `tool_requires` that are in the host context, for example test frameworks as `gtest` or `catch`.
- All transitive `requires` of those direct dependencies (all in the host context)
- Tool requires, in the build context, that is, application and executables that run in the build machine irrespective of the destination platform, are added exclusively to the <ExecutablePath> property, taking the value from `$(Conan{{name}}BinaryDirectories)` defined properties. This allows to define custom build commands, invoke code generation tools, with the <CustomBuild> and <Command> elements.

MSBuildToolchain

The `MSBuildToolchain` is the toolchain generator for MSBuild. It will generate MSBuild properties files that can be added to the Visual Studio solution projects. This generator translates the current package configuration, settings, and options, into MSBuild properties files syntax.

Important: This class will require very soon to define both the "host" and "build" profiles. It is very recommended to start defining both profiles immediately to avoid future breaking. Furthermore, some features, like trying to cross-compile might not work at all if the "build" profile is not provided.

The `MSBuildToolchain` generator can be used by name in conanfiles:

Listing 37: conanfile.py

```
class Pkg(ConanFile):
    generators = "MSBuildToolchain"
```

Listing 38: conanfile.txt

```
[generators]
MSBuildToolchain
```

And it can also be fully instantiated in the conanfile `generate()` method:

```
from conans import ConanFile
from conan.tools.microsoft import MSBuildToolchain

class App(ConanFile):
    settings = "os", "arch", "compiler", "build_type"

    def generate(self):
        tc = MSBuildToolchain(self)
        tc.generate()
```

The `MSBuildToolchain` will generate three files after a `conan install` command:

```
$ conan install conanfile.py # default is Release
$ conan install conanfile.py -s build_type=Debug
```

- The main *conantoolchain.props* file, to be added to the project.
- A *conantoolchain_<config>.props* file, that will be conditionally included from the previous *conantoolchain.props* file based on the configuration and platform, e.g.: *conantoolchain_release_x86.props*
- A *conanvcvars.bat* file with the necessary `vcvars` invocation to define the build environment if necessary to build from the command line or from automated tools (might not be necessary if opening the IDE). This file will be automatically called by the `tools.microsoft.MSBuild` helper `build()` method.

Every invocation to `conan install` with different configuration will create a new properties `.props` file, that will also be conditionally included. This allows to install different configurations, then switch among them directly from the Visual Studio IDE.

The MSBuildToolchain files can configure:

- The Visual Studio runtime (MT/MD/MTd/MDd), obtained from Conan input settings
- The C++ standard, obtained from Conan input settings

One of the advantages of using toolchains is that they can help to achieve the exact same build with local development flows, than when the package is created in the cache.

conf

`MSBuildToolchain` is affected by these *[conf]* variables:

- `tools.microsoft.msbuildtoolchain:compile_options` dict-like object of extra compile options to be added to <ClCompile> section. The dict will be translated as follows: <[KEY]>[VALUE]</[KEY]>.
- `tools.build:cxxflags` list of extra C++ flags that will be appended to <AdditionalOptions> section from <ClCompile> and <ResourceCompile> one.
- `tools.build:cflags` list of extra of pure C flags that will be appended to <AdditionalOptions> section from <ClCompile> and <ResourceCompile> one.
- `tools.build:sharedlinkflags` list of extra linker flags that will be appended to <AdditionalOptions> section from <Link> one.
- `tools.build:exelinkflags` list of extra linker flags that will be appended to <AdditionalOptions> section from <Link> one.
- `tools.build:defines` list of preprocessor definitions that will be appended to <PreprocessorDefinitions> section from <ResourceCompile> one.

MSBuild

The `MSBuild` build helper is a wrapper around the command line invocation of MSBuild. It will abstract the calls like `msbuild "MyProject.sln" /p:Configuration=<conf> /p:Platform=<platform>` into Python method calls.

The `MSBuild` helper can be used like:

```
from conans import conanfile
from conan.tools.microsoft import MSBuild

class App(ConanFile):
    settings = "os", "arch", "compiler", "build_type"

    def build(self):
        msbuild = MSBuild(self)
        msbuild.build("MyProject.sln")
```

The `MSBuild.build()` method internally implements a call to `msbuild` like:

```
$ <vcvars-cmd> && msbuild "MyProject.sln" /p:Configuration=<conf> /p:Platform=<platform>
```

Where:

- `vcvars-cmd` is calling the Visual Studio prompt that matches the current recipe `settings`
- `conf` is the configuration, typically Release, Debug, which will be obtained from `settings.build_type` but this will be configurable. Please open a Github issue if you want to define custom configurations.
- `platform` is the architecture, a mapping from the `settings.arch` to the common 'x86', 'x64', 'ARM', 'ARM64'. If your platform is unsupported, please report in Github issues as well.

conf

`MSBuild` is affected by these *[conf]* variables:

- `tools.microsoft.msbuild:verbosity` will accept one of `"Quiet"`, `"Minimal"`, `"Normal"`, `"Detailed"`, `"Diagnostic"` to be passed to the `MSBuild.build()` call as `msbuild .... /verbosity:XXX`

VCVars

Generates a file called `conanvcvars.bat` that activate the Visual Studio developer command prompt according to the current settings by wrapping the vcvarsall Microsoft bash script.

The `VCVars` generator can be used by name in conanfiles:

Listing 39: conanfile.py

```
class Pkg(ConanFile):
    generators = "VCVars"
```

Listing 40: conanfile.txt

```
[generators]
VCVars
```

And it can also be fully instantiated in the conanfile `generate()` method:

Listing 41: conanfile.py

```
from conans import ConanFile
from conan.tools.microsoft import VCVars

class Pkg(ConanFile):
    settings = "os", "compiler", "arch", "build_type"
    requires = "zlib/1.2.11", "bzip2/1.0.8"

    def generate(self):
        ms = VCVars(self)
        ms.generate()
```

Constructor

```
def __init__(self, conanfile):
```

- `conanfile`: the current recipe object. Always use `self`.

generate()

```
def generate(self, scope="build"):
```

Parameters:

- **scope** (Defaulted to "build"): Add the launcher automatically to the conanbuild launcher. Read more in the *Environment documentation*.

conan.tools.microsoft.is_msvc()

```
def is_msvc(conanfile):
```

Validate self.settings.compiler for which compiler is being used. It returns True when the host compiler is Visual Studio or msvc, otherwise, returns False. When the compiler is empty, it returns False.

Parameters:

- **conanfile**: ConanFile instance.

```
from conan.tools.microsoft import is_msvc

def validate(self):
    if not is_msvc(self):
        raise ConanInvalidConfiguration("Only supported by Visual Studio and msvc.")
```

conan.tools.microsoft.is_msvc_static_runtime()

```
def is_msvc_static_runtime(conanfile):
```

Validate self.settings.compiler.runtime for which compiler is being used. It returns True when the host compiler is Visual Studio or msvc, and its runtime is MT, MTd or static. When the compiler is empty, it returns False.

Parameters:

- **conanfile**: ConanFile instance.

```
from conan.tools.microsoft import is_msvc_static_runtime

def validate(self):
    if is_msvc_static_runtime(self) and self.options.shared(self):
        raise ConanInvalidConfiguration("This project does not support shared and static␣
↪runtime together.")
```

conan.tools.microsoft.msvc_runtime_flag()

```
def msvc_runtime_flag(conanfile):
```

If the current compiler is `Visual Studio`, `msvc`, `clang ``` or ```intel-cc`, then detects the runtime type and returns between `MD`, `MT`, `MDd` or `MTd`, otherwise, returns `""` (empty string). When the runtime type is `static`, it returns `MT`, otherwise, `MD`. The suffix `d` is added when running on Debug mode.

Parameters:

- **conanfile**: Conanfile instance.

```
from conan.tools.microsoft import msvc_runtime_flag

def validate(self):
     if "MT" in msvc_runtime_flag(self):
        self.output.warning("Runtime MT/MTd is not well tested.")
```

conan.tools.microsoft.unix_path()

```
def unix_path(conanfile, path):
```

Transforms the specified path into the correct one according to the subsystem. To determine the subsystem:

- The `settings_build.os` is checked to verify that we are running on "Windows" otherwise, the path is returned without changes.
- If `settings_build.os.subsystem` is specified (meaning we are running Conan under that subsystem) it will be returned.
- If `conanfile.win_bash==True` (meaning we have to run the commands inside the subsystem), the conf `tools.microsoft.bash:subsystem` has to be declared or it will raise an Exception.
- Otherwise the path is returned without changes.

Parameters:

- **conanfile**: ConanFile instance.

```
from conan.tools.microsoft import unix_path



def build(self):
    adjusted_path = unix_path(self, "C:\\path\\to\\stuff")
```

In the example above, `adjusted_path` will be:

- `/c/path/to/stuff` if msys2 or msys
- `/cygdrive/c/path/to/stuff` if cygwin
- `/mnt/c/path/to/stuff` if wsl
- `/dev/fs/C/path/to/stuff` if sfu

check_min_vs()

Helper method to allow the migration to 2.0 more easily. It will handle internally both `Visual Studio` and `msvc` compiler settings, raising a `ConanInvalidConfiguration` error if the minimum version is not satisfied

```
def check_min_vs(conanfile, version):
```

- `conanfile`: Always use `self`, the current recipe
- `version`: Minimum version that will be accepted. Use a version number following the MSVC compiler version (or `msvc` setting), that is, `191`, `192`, etc (updates like `193.1` are also acceptable)

Example:

```
def validate(self):
    check_min_vs(self, "192")
```

conan.tools.qbs

QbsProfile

Warning: This is an **experimental** feature subject to breaking changes in future releases.

Available since: 1.33.0

The `QbsProfile` can be used in the `generate()` method:

```
from conans import ConanFile
from conan.tools.qbs import QbsProfile

class App(ConanFile):
    settings = "os", "arch", "compiler", "build_type"
    requires = "hello/0.1"
    options = {"shared": [True, False]}
    default_options = {"shared": False}

    def generate(self):
        tc = QbsProfile(self)
        tc.generate()
```

The `QbsProfile` will generate the following file during **`conan install`** command (or before calling the `build()` method when the package is being built in the cache): *conan_toolchain_profile.qbs*. This file will contain a qbs profile named *conan_toolchain_profile*.

conan_toolchain_profile.qbs will contain the definitions of all the Qbs properties related to the Conan options and settings for the current package, platform, etc. This includes the following:

- Detection of compiler.
- Based on the compiler set in environment variable `CC`.
- Uses detected system compiler based on Conan setting `compiler` if environment variable `CC` is not set.
- Detection of compiler flags from environment (as defined at https://www.gnu.org/software/make/manual/html_node/Implicit-Variables.html):

  - ASFLAGS
  - CFLAGS
  - CPPFLAGS
  - CXXFLAGS
  - LDFLAGS
- Detection of sysroot from environment.
- Detection of `build_type` from Conan settings.
- Detection of `arch` from Conan settings.
- Detection of `compiler.cxxstd` from Conan settings.
- Detection of `fPIC` based on the existence of such option in the recipe.

Qbs

If you are using **Qbs** as your build system, you can use the **Qbs** build helper.

```
from conans import ConanFile
from conan.tools.qbs import Qbs

class ConanFileToolsTest(ConanFile):
    ...

    def build(self):
        qbs = Qbs(self)
        qbs.build()
```

Constructor

```
class Qbs(object):

    def __init__(self, conanfile, project_file=None)
```

Parameters:

- **conanfile** (Required): Use `self` inside a `conanfile.py`.
- **project_file** (Optional, Defaulted to `None`): Path to the root project file.

Attributes

profile

Defaulted to: `conan_toolchain_profile`

Specifies the qbs profile to build the project for.

Methods

add_configuration()

```
def add_configuration(self, name, values)
```

Add a build configuration to use.

Parameters:

- **name** (Required): Specifies build configuration name.
- **values** (Required): A dict of properties set for this build configuration.

build()

```
def build(self, products=None)
```

Build Qbs project.

Parameters:

- **products** (Optional, Defaulted to `None`): Specifies a list of products to build. If `None` build all products which have the qbs property `buildByDefault` set to `true`.

build_all()

```
def build_all(self)
```

Build all products of Qbs project, even products which set the qbs property `buildByDefault` set to `false`

install()

```
def install(self)
```

Install products.

Example

A typical usage of the Qbs build helper, if you want to be able to both execute **`conan create`** and also build your package for a library locally (in your user folder, not in the local cache), could be:

```
from conans import ConanFile
from conan.tools.qbs import Qbs


class HelloConan(ConanFile):
    name = "hello"
    version = "0.1"
    settings = "os", "compiler", "build_type", "arch"
```

(continues on next page)

(continued from previous page)

```
    generators = "qbs"
    exports_sources = "src/*", "*.qbs"
    no_copy_source = True
    requires = "zlib/1.2.11"

    def build(self):
        qbs = Qbs(self)
        qbs.add_configuration("default", {
            "project.conanBuildInfo", self.build_folder + "/conanbuildinfo.qbs"
        })
        qbs.build()

    def package(self):
        self.copy("*.h", dst="include", src="src")
        self.copy("*.lib", dst="lib", keep_path=False)
        self.copy("*.dll", dst="bin", keep_path=False)
        self.copy("*.dylib*", dst="lib", keep_path=False)
        self.copy("*.so", dst="lib", keep_path=False)
        self.copy("*.a", dst="lib", keep_path=False)

    def package_info(self):
        self.cpp_info.libs = ["hello"]
```

Note the `qbs` generator, which generates the *conanbuildinfo.qbs* file, to process dependencies information. Setting `no_copy_source = True` helps qbs to pick the right project file and not get confused by the generated files.

The *hello.qbs* could be as simple as:

```
Project {
    readonly property path conanBuildInfo

    references: conanBuildInfo

    DynamicLibrary {
        name: "hello"
        version: "0.1.0"
        files: "src/hello.cpp"
        cpp.cxxLanguageVersion: "c++11"

        Depends { name: "cpp" }
        Depends { name: "zlib" }
    }
}
```

conan.tools.env

Warning: These tools are still **experimental** (so subject to breaking changes) but with very stable syntax. We encourage the usage of it to be prepared for Conan 2.0.

Environment

Warning: These tools are still **experimental** (so subject to breaking changes) but with very stable syntax. We encourage the usage of it to be prepared for Conan 2.0.

`Environment` is a generic class that helps defining modifications to the environment variables. This class is used by other tools like the *conan.tools.gnu* autotools helpers and the *VirtualBuildEnv* and *VirtualRunEnv* generator. It is important to highlight that this is a generic class, to be able to use it, a specialization for the current context (shell script, bat file, path separators, etc), a `EnvVars` object needs to be obtained from it.

Variable declaration

```
from conan.tools.env import Environment

def generate(self):
    env = Environment()
    env.define("MYVAR1", "MyValue1")  # Overwrite previously existing MYVAR1 with new␣
↪value
    env.append("MYVAR2", "MyValue2")  # Append to existing MYVAR2 the new value
    env.prepend("MYVAR3", "MyValue3") # Prepend to existing MYVAR3 the new value
    env.remove("MYVAR3", "MyValue3")  # Remove the MyValue3 from MYVAR3
    env.unset("MYVAR4")               # Remove MYVAR4 definition from environment

    # And the equivalent with paths
    env.define_path("MYPATH1", "path/one")  # Overwrite previously existing MYPATH1 with␣
↪new value
    env.append_path("MYPATH2", "path/two")  # Append to existing MYPATH2 the new value
    env.prepend_path("MYPATH3", "path/three") # Prepend to existing MYPATH3 the new value
```

The "normal" variables (the ones declared with `define`, `append` and `prepend`) will be appended with a space, by default, but the `separator` argument can be provided to define a custom one.

The "path" variables (the ones declared with `define_path`, `append_path` and `prepend_path`) will be appended with the default system path separator, either `:` or `;`, but it also allows defining which one.

Composition

Environments can be composed:

```
from conan.tools.env import Environment

env1 = Environment()
env1.define(...)
env2 = Environment()
env2.append(...)

env1.compose_env(env2) # env1 has priority, and its modifications will prevail
```

Obtaining environment variables

You can obtain an `EnvVars` object with the `vars()` method like this:

```
from conan.tools.env import Environment

def generate(self):
    env = Environment()
    env.define("MYVAR1", "MyValue1")
    envvars = env.vars(self, scope="build")
    # use the envvars object
```

The default `scope` is equal `"build"`, which means that if this `envvars` generate a script to activate the variables, such script will be automatically added to the `conanbuild.sh|bat` one, for users and recipes convenience. Conan generators use `build` and `run` scope, but it might be possible to manage other scopes too.

Environment definition

There are some other places where `Environment` can be defined and used:

- In recipes `package_info()` method, in new `self.buildenv_info` and `self.runenv_info`, this environment will be propagated via `VirtualBuildEnv` and `VirtualRunEnv` respectively to packages depending on this recipe.
- In generators like `AutootoolsDeps`, `AutotoolsToolchain`, that need to define environment for the current recipe.
- In profiles new `[buildenv]` section.

The definition in `package_info()` is as follow, taking into account that both `self.buildenv_info` and `self.runenv_info` are objects of `Environment()` class.

```
from conans import ConanFile

class App(ConanFile):
    name = "mypkg"
    version = "1.0"
    settings = "os", "arch", "compiler", "build_type"

    def package_info(self):
```

(continues on next page)

(continued from previous page)

```
        # This is information needed by consumers to build using this package
        self.buildenv_info.append("MYVAR", "MyValue")
        self.buildenv_info.prepend_path("MYPATH", "some/path/folder")

        # This is information needed by consumers to run apps that depends on this␣
↪package
        # at runtime
        self.runenv_info.define("MYPKG_DATA_DIR", os.path.join(self.package_folder,
                                                               "datadir"))
```

EnvVars

Warning: These tools are still **experimental** (so subject to breaking changes) but with very stable syntax. We encourage the usage of it to be prepared for Conan 2.0.

`EnvVars` is a class that represents an instance of environment variables for a given system. It is obtained from the generic `Environment` class.

This class is used by other tools like the *conan.tools.gnu* autotools helpers and the *VirtualBuildEnv* and *VirtualRunEnv* generator.

Creating environment files

`EnvVars` object can generate environment (shell, bat or powershell scripts) files:

```
def generate(self):
    env1 = Environment()
    env1.define("foo", "var")
    envvars = env1.vars(self)
    envvars.save_script("my_env_file")
```

Although it potentially could be used in other methods, this functionality is intended to work in the `generate()` method.

It will generate automatically a `my_env_file.bat` for Windows systems or `my_env_file.sh` otherwise.

In Windows, it is possible to opt-in to generate Powershell `.ps1` scripts instead of `.bat` ones, using the conf `tools.env.virtualenv:powershell=True`.

Also, by default, Conan will automatically append that launcher file path to a list that will be used to create a `conanbuild.bat|sh|ps1` file aggregating all the launchers in order. The `conanbuild.sh|bat|ps1` launcher will be created after the execution of the `generate()` method.

The `scope` argument (`"build"` by default) can be used to define different scope of environment files, to aggregate them separately. For example, using a `scope="run"`, like the `VirtualRunEnv` generator does, will aggregate and create a `conanrun.bat|sh|ps1` script:

```
def generate(self):
    env1 = Environment(self)
    env1.define("foo", "var")
    envvars = env1.vars(self, scope="run")
```

(continues on next page)

(continued from previous page)

```
    # Will append "my_env_file" to "conanrun.bat|sh|ps1"
    envvars.save_script("my_env_file")
```

You can also use `scope=None` argument to avoid appending the script to the aggregated `conanbuild.bat|sh|ps1`:

```
env1 = Environment(self)
env1.define("foo", "var")
# Will not append "my_env_file" to "conanbuild.bat|sh|ps1"
envvars = env1.vars(self, scope=None)
envvars.save_script("my_env_file")
```

Running with environment files

The `conanbuild.bat|sh|ps1` launcher will be executed by default before calling every `self.run()` command. This would be typically done in the `build()` method.

You can change the default launcher with the `env` argument of `self.run()`:

```
...
def build(self):
    # This will automatically wrap the "foo" command with the correct environment:
    # source my_env_file.sh && foo
    # my_env_file.bat && foo
    # powershell my_env_file.ps1 ; cmd c/ foo
    self.run("foo", env=["my_env_file"])
```

Applying the environment variables

As an alternative to running a command, environments can be applied in the python environment:

```
from conan.tools.env import Environment

env1 = Environment(self)
env1.define("foo", "var")
envvars = env1.vars(self)
with envvars.apply():
   # Here os.getenv("foo") == "var"
   ...
```

Iterating the variables

You can iterate the environment variables of an `EnvVars` object like this:

```
env1 = Environment()
env1.append("foo", "var")
env1.append("foo", "var2")
envvars = env1.vars(self)
for name, value in envvars.items():
```

(continues on next page)

(continued from previous page)

```
    assert name == "foo":
    assert value == "var var2"
```

Warning: In Windows, there is a limit to the size of environment variables, a total of 32K for the whole environment, but spcifically the PATH variable has a limit of 2048 characters. That means that the above utils could hit that limit, for example for large dependency graphs where all packages contribute to the PATH env-var.

This can be mitigated by:

- Putting the Conan cache closer to C:/ for shorter paths
- Better definition of what dependencies can contribute to the PATH env-var
- Other mechanisms for things like running with many shared libraries dependencies with too many .dlls, like `imports`

VirtualBuildEnv

Warning: These tools are still **experimental** (so subject to breaking changes) but with very stable syntax. We encourage the usage of it to be prepared for Conan 2.0.

The `VirtualBuildEnv` generator can be used by name in conanfiles:

Listing 42: conanfile.py

```
class Pkg(ConanFile):
    generators = "VirtualBuildEnv"
```

Listing 43: conanfile.txt

```
[generators]
VirtualBuildEnv
```

And it can also be fully instantiated in the conanfile `generate()` method:

Listing 44: conanfile.py

```
from conans import ConanFile
from conan.tools.env import VirtualBuildEnv

class Pkg(ConanFile):
    settings = "os", "compiler", "arch", "build_type"
    requires = "zlib/1.2.11", "bzip2/1.0.8"

    def generate(self):
        ms = VirtualBuildEnv(self)
        ms.generate()
```

When the `VirtualBuildEnv` generator is used, calling **`conan install`** will generate a *conanbuildenv* .bat or .sh script containing environment variables of the build time environment.

That information is collected from the direct `tool_requires` in "build" context recipes from the `self.buildenv_info` definition plus the `self.runenv_info` of the transitive dependencies of those `tool_requires`.

This generator (for example the invocation of `conan install cmake/3.20.0@ -g VirtualBuildEnv --build-require`) will create the following files:

- conanbuildenv-release-x86_64.(bat|sh): This file contains the actual definition of environment variables like PATH, LD_LIBRARY_PATH, etc, and any other variable defined in the dependencies `buildenv_info` corresponding to the `build` context, and to the current installed configuration. If a repeated call is done with other settings, a different file will be created. After the execution or sourcing of this file, a new deactivation script will be generated, capturing the current environment, so the environment can be restored when desired. The file will be named also following the current active configuration, like `deactivate_conanbuildenv-release-x86_64.bat`.
- conanbuild.(bat|sh): Accumulates the calls to one or more other scripts, in case there are multiple tools in the generate process that create files, to give one single convenient file for all. This only calls the latest specific configuration one, that is, if `conan install` is called first for Release build type, and then for Debug, `conanbuild.(bat|sh)` script will call the Debug one.
- deactivate_conanbuild.(bat|sh): Accumulates the deactivation calls defined in the above `conanbuild.(bat|sh)`. This file should only be called after the accumulated activate has been called first.

Constructor

```
def __init__(self, conanfile):
```

- `conanfile`: the current recipe object. Always use `self`.

generate()

```
def generate(self, scope="build"):
```

Parameters:

- **scope** (Defaulted to `"build"`): Add the launcher automatically to the `conanbuild` launcher. Read more in the *Environment documentation*.

VirtualRunEnv

Warning: These tools are still **experimental** (so subject to breaking changes) but with very stable syntax. We encourage the usage of it to be prepared for Conan 2.0.

The `VirtualRunEnv` generator can be used by name in conanfiles:

Listing 45: conanfile.py

```
class Pkg(ConanFile):
    generators = "VirtualRunEnv"
```

Listing 46: conanfile.txt

```
[generators]
VirtualRunEnv
```

And it can also be fully instantiated in the conanfile `generate()` method:

Listing 47: conanfile.py

```
from conans import ConanFile
from conan.tools.env import VirtualRunEnv

class Pkg(ConanFile):
    settings = "os", "compiler", "arch", "build_type"
    requires = "zlib/1.2.11", "bzip2/1.0.8"

    def generate(self):
        ms = VirtualRunEnv(self)
        ms.generate()
```

When the `VirtualRunEnv` generator is used, calling **`conan install`** will generate a *conanrunenv* .bat or .sh script containing environment variables of the run time environment.

The launcher contains the runtime environment information, anything that is necessary in the environment to actually run the compiled executables and applications. The information is obtained from the `self.runenv_info` and also automatically deduced from the `self.cpp_info` definition of the package, to define `PATH`, `LD_LIBRARY_PATH`, `DYLD_LIBRARY_PATH` and `DYLD_FRAMEWORK_PATH` environment variables.

This generator will create the following files:

- conanrunenv-release-x86_64.(bat|sh): This file contains the actual definition of environment variables like PATH, LD_LIBRARY_PATH, etc, and `runenv_info` of dependencies corresponding to the `host` context, and to the current installed configuration. If a repeated call is done with other settings, a different file will be created.
- conanrun.(bat|sh): Accumulates the calls to one or more other scripts to give one single convenient file for all. This only calls the latest specific configuration one, that is, if `conan install` is called first for Release build type, and then for Debug, `conanrun.(bat|sh)` script will call the Debug one.

After the execution of one of those files, a new deactivation script will be generated, capturing the current environment, so the environment can be restored when desired. The file will be named also following the current active configuration, like `deactivate_conanrunenv-release-x86_64.bat`.

Let's see an example on how to add an environment variable to the `runenv_info` and get its value later in the consumer side using the `conanrun` launcher:

Listing 48: conanfile.py

```
from conan import ConanFile

class HelloConan(ConanFile):

    def package_info(self):
        self.runenv_info.define("MYVAR", "My value!")
```

Listing 49: test_package/conanfile.py

```
from conan import ConanFile

class HelloTestConan(ConanFile):
    # VirtualBuildEnv and VirtualRunEnv can be avoided if "tools.env.virtualenv:auto_use
↪" is defined
    # (it will be defined in Conan 2.0)
    generators = "VirtualRunEnv"

    def test(self):
        self.run("echo $MYVAR", env="conanrun")  # Unix-style
```

As we already said above, the `conanrun` launcher contains the runtime environment information, so let's run a **`conan create . hello/1.0@`** and check the console output that should show something like this:

```
....
Configuring environment variables
My value!
```

Constructor

```
def __init__(self, conanfile):
```

- `conanfile`: the current recipe object. Always use `self`.

generate()

```
def generate(self, scope="run"):
```

Parameters:

- **scope** (Defaulted to `run`): Add the launcher automatically to the `conanrun` launcher. Read more in the *Environment documentation*.

conan.tools.system

> **Warning:** These tools are **experimental** and subject to breaking changes.

conan.tools.system.package_manager

> **Warning:** These tools are **experimental** and subject to breaking changes.

The tools under *conan.tools.system.package_manager* are wrappers around some of the most popular system package managers for different platforms. You can use them to invoke system package managers in recipes and perform the

most typical operations, like installing a package, updating the package manager database or checking if a package is installed.

You can use these tools inside the *system_requirements()* method of your recipe, like:

Listing 50: conanfile.py

```
from conan.tools.system.package_manager import Apt, Yum, PacMan, Zypper

def system_requirements(self):
    # depending on the platform or the tools.system.package_manager:tool configuration
    # only one of these will be executed
    Apt(self).install(["libgl-dev"])
    Yum(self).install(["libglvnd-devel"])
    PacMan(self).install(["libglvnd"])
    Zypper(self).install(["Mesa-libGL-devel"])
```

Conan will automatically choose which package manager to use by looking at the Operating System name. In the example above, if we are running on Ubuntu Linux, Conan will ignore all the calls except for the `Apt()` one and will only try to install the packages using the `apt-get` tool. Conan uses the following mapping by default:

- *Apt* for **Linux** with distribution names: *ubuntu*, *debian*
- *Yum* for **Linux** with distribution names: *pidora*, *scientific*, *xenserver*, *amazon*, *oracle*, *amzn*, *almalinux*
- *Dnf* for **Linux** with distribution names: *fedora*, *rhel*, *centos*, *mageia*
- *Brew* for **macOS**
- *PacMan* for **Linux** with distribution names: *arch*, *manjaro* and when using **Windows** with *msys2*
- *Chocolatey* for **Windows**
- *Zypper* for **Linux** with distribution names: *opensuse*, *sles*
- *Pkg* for **Linux** with distribution names: *freebsd*
- *PkgUtil* for **Solaris**

You can override this default mapping and set the package manager tool you want to use by default setting the configuration property *tools.system.package_manager:tool*.

Methods available for system package manager tools

All these wrappers share three methods that represent the most common operations with a system package manager. They take the same form for all of the package managers except for *Apt* that also accepts the *recommends* argument for the *install method*.

- `install(self, packages, update=False, check=False)`: try to install the list of packages passed as a parameter. If the parameter `check` is `True` it will check if those packages are already installed before installing them. If the parameter `update` is `True` it will try to update the package manager database before checking and installing. Its behaviour is affected by the value of `tools.system.package_manager:mode` *configuration*.
- `install_substitutes(packages_substitutes, update=False, check=True)`: try to install the list of lists of substitutes packages passed as a parameter, e.g., `[["pkg1", "pkg2"], ["pkg3"]]`. It succeeds if one of the substitutes list is completely installed, so it's intended to be used when you have different packages for different distros. Internally, it's calling the previous `install(packages, update=update, check=check)` method, so `update` and `check` have the same purpose as above.

- `update()` update the system package manager database. Its behaviour is affected by the value of `tools.system.package_manager:mode` *configuration*.
- `check(packages)` check if the list of packages passed as parameter are already installed.

Configuration properties that affect how system package managers are invoked

As explained above there are several *configuration properties* that affect how these tools are invoked:

- `tools.system.package_manager:tool`: to choose which package manager tool you want to use by default: `"apt-get"`, `"yum"`, `"dnf"`, `"brew"`, `"pacman"`, `"choco"`, `"zypper"`, `"pkg"` or `"pkgutil"`
- `tools.system.package_manager:mode`: mode to use when invoking the package manager tool. There are two possible values:
  - `"check"`: will not try to update the package manager database or install any packages in any case. This is the default value.
  - `"install"`: it will allow Conan to perform update or install operations.
- `tools.system.package_manager:sudo`: Use *sudo* when invoking the package manager tools in Linux (`False` by default)
- `tools.system.package_manager:sudo_askpass`: Use the `-A` argument if using sudo in Linux to invoke the system package manager (`False` by default)

There are some specific arguments for each of these tools. Here is the complete reference:

conan.tools.system.package_manager.Apt

Will invoke the *apt-get* command. Enabled by default for **Linux** with distribution names: *ubuntu* and *debian*.

Constructor

```
def __init__(self, conanfile, arch_names=None):
```

- **conanfile**: the current recipe object. Always use `self`.
- **arch_names**: this argument maps the Conan architecture setting with the package manager tool architecture names. It is `None` by default, which means that it will use a default mapping for the most common architectures. For example, if you are using `x86_64` Conan architecture setting, it will map this value to `amd64` for *Apt* and try to install the `<package_name>:amd64` package. You can pass this argument to override the default Conan mapping, like:

Listing 51: conanfile.py

```
...
def system_requirements(self):
    apt = Apt(self, arch_names={"<conan_arch_setting>": "apt_arch_setting"})
    apt.install(["libgl-dev"])
```

The default mapping Conan uses for *APT* packages architecture is:

```
self._arch_names = {"x86_64": "x86_64",
                    "x86": "i?86",
                    "ppc32": "powerpc",
                    "ppc64le": "ppc64le",
                    "armv7": "armv7",
                    "armv7hf": "armv7hl",
                    "armv8": "aarch64",
                    "s390x": "s390x"} if arch_names is None else arch_names
```

Methods

- `install(self, packages, update=False, check=False, recommends=False)`:: will try to install the list of packages passed as a parameter. If the parameter `check` is `True` it will check if those packages are already installed before installing them. If the parameter `update` is `True` it will try to update the package manager database before checking and installing. If the parameter `recommends` is `False` it will add the `'--no-install-recommends'` argument to the *apt-get* command call. Its behaviour is affected by the value of `tools.system.package_manager:mode` *configuration*.
- `update()` same behaviour as the one explained in the *section* above.
- `check(packages)` same behaviour as the one explained in the *section* above.

conan.tools.system.package_manager.Yum

Will invoke the *yum* command. Enabled by default for **Linux** with distribution names: *pidora*, *scientific*, *xenserver*, *amazon*, *oracle*, *amzn* and *almalinux*.

Constructor

```
def __init__(self, conanfile, arch_names=None):
```

- **conanfile**: the current recipe object. Always use `self`.
- **arch_names**: this argument maps the Conan architecture setting with the package manager tool architecture names. It is `None` by default, which means that it will use a default mapping for the most common architectures. For example, if you are using `x86` Conan architecture setting, it will map this value to `i?86` for *Yum* and try to install the `<package_name>.i?86` package.

The default mapping Conan uses for *Yum* packages architecture is:

```
self._arch_names = {"x86_64": "x86_64",
                    "x86": "i?86",
                    "ppc32": "powerpc",
                    "ppc64le": "ppc64le",
                    "armv7": "armv7",
                    "armv7hf": "armv7hl",
                    "armv8": "aarch64",
                    "s390x": "s390x"} if arch_names is None else arch_names
```

conan.tools.system.package_manager.Dnf

Will invoke the *dnf* command. Enabled by default for **Linux** with distribution names: *fedora*, *rhel*, *centos* and *mageia*. This tool has exactly the same default values, constructor and methods than the *Yum* tool.

conan.tools.system.package_manager.PacMan

Will invoke the *pacman* command. Enabled by default for **Linux** with distribution names: *arch*, *manjaro* and when using **Windows** with *msys2*

Constructor

```
def __init__(self, conanfile, arch_names=None):
```

- **conanfile**: the current recipe object. Always use `self`.
- **arch_names**: this argument maps the Conan architecture setting with the package manager tool architecture names. It is `None` by default, which means that it will use a default mapping for the most common architectures. If you are using `x86` Conan architecture setting, it will map this value to `lib32` for *PacMan* and try to install the `<package_name>-lib32` package.

The default mapping Conan uses for *PacMan* packages architecture is:

```
self._arch_names = {"x86": "lib32"} if arch_names is None else arch_names
```

conan.tools.system.package_manager.Zypper

Will invoke the *zypper* command. Enabled by default for **Linux** with distribution names: *opensuse*, *sles*.

Constructor

```
def __init__(self, conanfile, arch_names=None):
```

- **conanfile**: the current recipe object. Always use `self`.

conan.tools.system.package_manager.Brew

Will invoke the *brew* command. Enabled by default for **macOS**.

Constructor

```
def __init__(self, conanfile, arch_names=None):
```

- **conanfile**: the current recipe object. Always use `self`.

conan.tools.system.package_manager.Pkg

Will invoke the *pkg* command. Enabled by default for **Linux** with distribution names: *freebsd*.

Constructor

```
def __init__(self, conanfile, arch_names=None):
```

- **conanfile**: the current recipe object. Always use `self`.

conan.tools.system.package_manager.PkgUtil

Will invoke the *pkgutil* command. Enabled by default for **Solaris**.

Constructor

```
def __init__(self, conanfile, arch_names=None):
```

- **conanfile**: the current recipe object. Always use `self`.

conan.tools.system.package_manager.Chocolatey

Will invoke the *choco* command. Enabled by default for **Windows**.

Constructor

```
def __init__(self, conanfile, arch_names=None):
```

- **conanfile**: the current recipe object. Always use `self`

conan.tools.files

Warning: These tools are still **experimental** (so subject to breaking changes) but with very stable syntax. We encourage the usage of it to be prepared for Conan 2.0.

conan.tools.files basic operations

Warning: These tools are **experimental** and subject to breaking changes.

conan.tools.files.copy()

```
def copy(conanfile, pattern, src, dst, keep_path=True, excludes=None, ignore_case=True)
```

Copy the files matching the `pattern` (fnmatch) at the `src` folder to a `dst` folder.

Parameters:

- **conanfile**: Conanfile object.
- **pattern** (Required): An fnmatch file pattern of the files that should be copied. It must not start with `..` relative path or an exception will be raised.
- **src** (Required): Source folder in which those files will be searched. This folder will be stripped from the dst parameter. E.g., *lib/Debug/x86*.
- **dst** (Required): Destination local folder. It must be different from `src` value or an exception will be raised.
- **keep_path** (Optional, defaulted to `True`): Means if you want to keep the relative path when you copy the files from the **src** folder to the **dst** one.
- **excludes** (Optional, defaulted to `None`): A tuple/list of fnmatch patterns or even a single one to be excluded from the copy.
- **ignore_case** (Optional, defaulted to `True`): If enabled, it will do a case-insensitive pattern matching.

Note: The files that are **symlinks to files** or **symlinks to folders** with be treated like any other file, so they will only be copied if the specified pattern matches with the file.

At the destination folder, the symlinks will be created pointing to the exact same file or folder, absolute or relative, being the responsibility of the user to manipulate the symlink to, for example, transform the symlink into a relative path before copying it so it points to the destination folder.

Check *here* the reference of tools to manage symlinks.

---

conan.tools.files.load()

```
def load(conanfile, path, encoding="utf-8")
```

Utility function to load files in one line. It will manage the open and close of the file, and load binary encodings. Returns the content of the file.

```
from conan.tools.files import load

content = load(self, "myfile.txt")
```

Parameters:

- **conanfile**: Conanfile object.
- **path** (Required): Path to the file.

- **encoding** (Optional, Defaulted to `utf-8`): Specifies the input file text encoding.

conan.tools.files.save()

```
def save(conanfile, path, content, append=False, encoding="utf-8"):
```

Utility function to save files in one line. It will manage the open and close of the file and creating directories if necessary.

```
from conan.tools.files import save

save(self, "path/to/otherfile.txt", "contents of the file")
```

Parameters:

- **conanfile**: Conanfile object.
- **path** (Required): Path to the file.
- **content** (Required): Content that should be saved into the file.
- **append** (Optional, Defaulted to `False`): If `True`, it will append the content.
- **encoding** (Optional, Defaulted to `utf-8`): Specifies the output file text encoding.

conan.tools.files.rename()

```
def rename(conanfile, src, dst)
```

Utility function to rename a file or folder *src* to *dst*. On Windows, it is very common that `os.rename()` raises an "Access is denied" exception, so this tool uses:command:*robocopy* if available. If that is not the case, or the rename is done in a non-Windows machine, it falls back to the `os.rename()` implementation.

```
from conan.tools.files import rename

def source(self):
    rename(self, "lib-sources-abe2h9fe", "sources")  # renaming a folder
```

Parameters:

- **conanfile**: Conanfile object.
- **src** (Required): Path to be renamed.
- **dst** (Required): Path to be renamed to.

conan.tools.files.replace_in_file()

```
def replace_in_file(conanfile, file_path, search, replace, strict=True, encoding="utf-8")
```

Replace a string `search` in the contents of the file `file_path` with the string `replace`.

```
from conan.tools.files import replace_in_file

replace_in_file(self, os.path.join(self.source_folder, "folder", "file.txt"), "foo", "bar
↪")
```

Parameters:

- **conanfile**: Conanfile object.
- **file_path** (Required): File path of the file to perform the replace in.
- **search** (Required): String you want to be replaced.
- **replace** (Required): String to replace the searched string.
- **strict** (Optional, Defaulted to `True`): If `True`, it raises an error if the searched string is not found, so nothing is actually replaced.
- **encoding** (Optional, Defaulted to `utf-8`): Specifies the input and output files text encoding.

conan.tools.files.mkdir()

```
def mkdir(path)
```

Utility functions to create a directory. The existence of the specified directory is checked, so `mkdir()` will do nothing if the directory already exists.

```
from conan.tools.files import mkdir

mkdir(self, "mydir") # Creates mydir if it does not already exist
mkdir(self, "mydir") # Does nothing
```

Parameters:

- **conanfile**: Conanfile object.
- **path** (Required): Path to the directory.

conan.tools.files.rmdir()

```
def rmdir(path)
```

Utility functions to remove a directory. The existence of the specified directory is checked, so `rmdir()` will do nothing if the directory doesn't exists.

```
from conan.tools.files import rmdir

rmdir(self, "mydir") # Remove mydir if it exist
rmdir(self, "mydir") # Does nothing
```

Parameters:

- **conanfile**: Conanfile object.
- **path** (Required): Path to the directory.

conan.tools.files.chdir()

```
def chdir(conanfile, newdir):
```

This is a context manager that allows to temporary change the current directory in your conanfile:

```
from conan.tools.files import chdir

def build(self):
    with chdir(self, "./subdir"):
        do_something()
```

Parameters:

- **conanfile**: Conanfile object.
- **newdir** (Required): Directory path name to change the current directory.

conan.tools.files.unzip()

```
def unzip(conanfile, filename, destination=".", keep_permissions=False, pattern=None,
          strip_root=False):
```

This function extract different compressed formats (`.tar.gz`, `.tar`, `.tzb2`, `.tar.bz2`, `.tgz`, `.txz`, `tar.xz`, and `.zip`) into the given destination folder.

It also accepts gzipped files, with extension `.gz` (not matching any of the above), and it will unzip them into a file with the same name but without the extension, or to a filename defined by the `destination` argument.

```
from conan.tools.files import unzip

tools.unzip("myfile.zip")
# or to extract in "myfolder" sub-folder
tools.unzip("myfile.zip", "myfolder")
```

You can keep the permissions of the files using the `keep_permissions=True` parameter.

```
from conan.tools.files import unzip

unzip(self, "myfile.zip", "myfolder", keep_permissions=True)
```

Use the `pattern` argument if you want to filter specific files and paths to decompress from the archive.

```
from conan.tools.files import unzip

# Extract only files inside relative folder "small"
unzip(self, "bigfile.zip", pattern="small/*")
# Extract only txt files
unzip(self, "bigfile.zip", pattern="*.txt")
```

Parameters:

- **conanfile**: Conanfile object.
- **filename** (Required): File to be unzipped.

- **destination** (Optional, Defaulted to "."): Destination folder for unzipped files.
- **keep_permissions** (Optional, Defaulted to `False`): Keep permissions of files. **WARNING:** Can be dangerous if the zip was not created in a NIX system, the bits could produce undefined permission schema. Use only this option if you are sure that the zip was created correctly.
- **pattern** (Optional, Defaulted to `None`): Extract from the archive only paths matching the pattern. This should be a Unix shell-style wildcard. See fnmatch documentation for more details.
- **strip_root** (Optional, Defaulted to `False`): When `True` and the ZIP file contains one folder containing all the contents, it will strip the root folder moving all its contents to the root. E.g: *mylib-1.2.8/main.c* will be extracted as *main.c*. If the compressed file contains more than one folder or only a file it will raise a `ConanException`.

conan.tools.files.update_conandata()

```
def update_conandata(conanfile, data)
```

Parameters:

- **conanfile**: Conanfile object.
- **data** (Required): A dictionary (can be nested), of values to update

This function reads the `conandata.yml` inside the exported folder in the conan cache, if it exists. If the `conandata.yml` does not exist, it will create it. Then, it updates the conandata dictionary with the provided `data` one, which is updated recursively, prioritizing the `data` values, but keeping other existing ones. Finally the `conandata.yml` is saved in the same place.

This helper can only be used within the `export()` method, it can raise otherwise. One application is to capture in the `conandata.yml` the scm coordinates (like Git remote url and commit), to be able to recover it later in the `source()` method and have reproducible recipes that can build from sources without actually storing the sources in the recipe.

Example:

```
from conan import ConanFile
from conan.tools.files import update_conandata

class Pkg(ConanFile):
    name = "pkg"
    version = "0.1"

    def export(self):
        # This is an example, doesn't make sense to have static data, instead you
        # could put the data directly in a conandata.yml file.
        # This would be useful for storing dynamic data, obtained at export() time from␣
↪elsewhere
        update_conandata(self, {"mydata": {"value": {"nested1": 123, "nested2": "some-
↪string"}}})

    def source(self):
        data = self.conan_data["sources"]["mydata"]
```

conan.tools.files downloads

Warning: These tools are **experimental** and subject to breaking changes.

conan.tools.files.get()

```
def get(conanfile, url, md5='', sha1='', sha256='', destination=".", filename="",
        keep_permissions=False, pattern=None, verify=True, retry=None, retry_wait=None,
        auth=None, headers=None, strip_root=False)
```

High level download and decompressing of a tgz, zip or other compressed format file. Just a high level wrapper for download, unzip, and remove the temporary zip file once unzipped. You can pass hash checking parameters: `md5`, `sha1`, `sha256`. All the specified algorithms will be checked. If any of them doesn't match, it will raise a `ConanException`.

Parameters:

- **url**, **filename**, **md5**, **sha1**, **sha256**, **verify**, **retry**, **retry_wait**, **auth**, **headers**: forwarded to `download()`
- **keep_permissions**, **pattern**, **strip_root**: forwarded to `tools.unzip()` (legacy, will be updated).

Examples:

```
from conan.tools.files import get

def source(self):
    get(self, "http://url/file", md5='d2da0cd0756cd9da6560b9a56016a0cb')
    # also, specify a destination folder
    get(self, "http://url/file", destination="subfolder")
```

conan.tools.files.ftp_download()

```
def ftp_download(conanfile, ip, filename, login='', password='')
```

Ftp download of a file. Retrieves a file from an FTP server. This doesn't support SSL, but you might implement it yourself using the standard Python FTP library.

Parameters:

- **conanfile**: Conanfile object, use always `self`
- **ip** (Required): The IP or address of the ftp server.
- **filename** (Required): The filename, including the path/folder where it is located.
- **login** (Optional, Defaulted to ""): Login credentials for the ftp server.
- **password** (Optional, Defaulted to ""): Password credentials for the ftp server.

Examples:

```
from conan.tools.files import ftp_download

def source(self):
    ftp_download(self, 'ftp.debian.org', "debian/README")
    self.output.info(load("README"))
```

conan.tools.files.download()

Download a file

```
def download(conanfile, url, filename, verify=True, retry=None, retry_wait=None,
             auth=None, headers=None, md5='', sha1='', sha256='')
```

Retrieves a file from a given URL into a file with a given filename. It uses certificates from a list of known verifiers for https downloads, but this can be optionally disabled.

You can pass hash checking parameters: `md5`, `sha1`, `sha256`. All the specified algorithms will be checked. If any of them doesn't match, the downloaded file will be removed and it will raise a `ConanException`.

Parameters:

- **conanfile** (Required): Conanfile object, use `self` always
- **url** (Required): URL to download. It can be a list, which only the first one will be downloaded, and the follow URLs will be used as mirror in case of download error.
- **filename** (Required): Name of the file to be created in the local storage
- **verify** (Optional, Defaulted to `True`): When False, disables https certificate validation.
- **retry** (Optional, Defaulted to `1`): Number of retries in case of failure.
- **retry_wait** (Optional, Defaulted to `5`): Seconds to wait between download attempts.
- **auth** (Optional, Defaulted to `None`): A tuple of user and password to use HTTPBasic authentication. This is used directly in the `requests` Python library. Check other uses here: https://requests.readthedocs.io/en/master/user/authentication/#basic-authentication
- **headers** (Optional, Defaulted to `None`): A dictionary with additional headers.
- **md5** (Optional, Defaulted to `""`): MD5 hash code to check the downloaded file.
- **sha1** (Optional, Defaulted to `""`): SHA-1 hash code to check the downloaded file.
- **sha256** (Optional, Defaulted to `""`): SHA-256 hash code to check the downloaded file.

Configuration:

- `tools.files.download:retry`: number of retries in case some error occurs.
- `tools.files.download:retry_wait`: seconds to wait between retries.

Examples:

```
download(self, "http://someurl/somefile.zip", "myfilename.zip")

# to disable verification:
download(self, "http://someurl/somefile.zip", "myfilename.zip", verify=False)

# to retry the download 2 times waiting 5 seconds between them
download(self, "http://someurl/somefile.zip", "myfilename.zip", retry=2, retry_wait=5)

# Use https basic authentication
download(self, "http://someurl/somefile.zip", "myfilename.zip", auth=("user", "password
↪"))

# Pass some header
download(self, "http://someurl/somefile.zip", "myfilename.zip", headers={"Myheader": "My␣
```

(continues on next page)

(continued from previous page)

```
↪value"})

# Download and check file checksum
download(self, "http://someurl/somefile.zip", "myfilename.zip", md5=
↪"e5d695597e9fa520209d1b41edad2a27")

# to add mirrors
download(self, ["https://ftp.gnu.org/gnu/gcc/gcc-9.3.0/gcc-9.3.0.tar.gz",
                "http://mirror.linux-ia64.org/gnu/gcc/releases/gcc-9.3.0/gcc-9.3.0.tar.gz
↪"],
                "gcc-9.3.0.tar.gz",
               sha256="5258a9b6afe9463c2e56b9e8355b1a4bee125ca828b8078f910303bc2ef91fa6")
```

Available since: 1.42.0

conan.tools.files patches

conan.tools.files.patch()

```
def patch(conanfile, base_path=None, patch_file=None, patch_string=None, strip=0,␣
↪fuzz=False, **kwargs):
```

Applies a diff from file (*patch_file*) or string (*patch_string*) in the `conanfile.source_folder` directory. The folder containing the sources can be customized with the `self.folders` attribute in the *layout(self) method*.

Parameters:

- **patch_file**: Patch file that should be applied. The path is relative to the **conanfile.source_folder** unless an absolute path is provided.
- **base_path**: The path is a relative path to **conanfile.export_sources_folder** unless an absolute path is provided.
- **patch_string**: Patch string that should be applied.
- **strip**: Number of folders to be stripped from the path.
- **output**: Stream object.
- **fuzz**: Should accept fuzzy patches.
- **kwargs**: Extra parameters that can be added and will contribute to output information.

```
from conan.tools.files import patch

def build(self):
    for it in self.conan_data.get("patches", {}).get(self.version, []):
        patch(self, **it)
```

conan.tools.files.apply_conandata_patches()

```
def apply_conandata_patches(conanfile):
```

Applies patches stored in `conanfile.conan_data` (read from `conandata.yml` file). It will apply all the patches under `patches` entry that matches the given `conanfile.version`. If versions are not defined in `conandata.yml` it will apply all the patches directly under `patches` keyword.

The key entries will be passed as kwargs to the *patch* function.

Example of `conandata.yml` without versions defined:

```
from conan.tools.files import apply_conandata_patches

def build(self):
    apply_conandata_patches(self)
```

```
patches:
- patch_file: "patches/0001-buildflatbuffers-cmake.patch"
- patch_file: "patches/0002-implicit-copy-constructor.patch"
  base_path: "subfolder"
  patch_type: backport
  patch_source: https://github.com/google/flatbuffers/pull/5650
  patch_description: Needed to build with modern clang compilers.
```

Example of `conandata.yml` with different patches for different versions:

```
patches:
  "1.11.0":
    - patch_file: "patches/0001-buildflatbuffers-cmake.patch"
    - patch_file: "patches/0002-implicit-copy-constructor.patch"
      base_path: "subfolder"
      patch_type: backport
      patch_source: https://github.com/google/flatbuffers/pull/5650
      patch_description: Needed to build with modern clang compilers.
  "1.12.0":
    - patch_file: "patches/0001-buildflatbuffers-cmake.patch"
    - patch_string: |
        --- a/tests/misc-test.c
        +++ b/tests/misc-test.c
        @@ -1232,6 +1292,8 @@ main (int argc, char **argv)
              g_test_add_func ("/misc/pause-cancel", do_pause_cancel_test);
              g_test_add_data_func ("/misc/stealing/async", GINT_TO_POINTER (FALSE), do_
↪stealing_test);
              g_test_add_data_func ("/misc/stealing/sync", GINT_TO_POINTER (TRUE), do_
↪stealing_test);
        +     g_test_add_func ("/misc/response/informational/content-length", do_
↪response_informational_content_length_test);
        +

        ret = g_test_run ();
    - patch_file: "patches/0003-fix-content-length-calculation.patch"
```

conan.tools.files checksums

Warning: These tools are **experimental** and subject to breaking changes.

conan.tools.files.check_md5()

```
def check_md5(conanfile, file_path, signature)
```

Check that the specified md5sum of the `file_path` matches with `signature`. If doesn't match it will raise a `ConanException`.

Parameters:

- **conanfile**: Conanfile object.
- **file_path** (Required): Path of the file to check.
- **signature** (Required): Expected md5sum.

conan.tools.files.check_sha1()

```
def check_sha1(conanfile, file_path, signature)
```

Check that the specified sha1 of the `file_path` matches with `signature`. If doesn't match it will raise a `ConanException`.

Parameters:

- **conanfile**: Conanfile object.
- **file_path** (Required): Path of the file to check.
- **signature** (Required): Expected sha1sum.

conan.tools.files.check_sha256()

```
def check_sha256(conanfile, file_path, signature)
```

Check that the specified sha256 of the `file_path` matches with `signature`. If doesn't match it will raise a `ConanException`.

Parameters:

- **conanfile**: Conanfile object.
- **file_path** (Required): Path of the file to check.
- **signature** (Required): Expected sha256sum.

conan.tools.files.symlinks

conan.tools.files.symlinks.absolute_to_relative_symlinks()

```
def absolute_to_relative_symlinks(conanfile, base_folder):
```

Convert the symlinks with absolute paths into relative ones if they are pointing to a file or directory inside the 'base_folder'. Any absolute symlink pointing outside the 'base_folder' will be ignored.

Parameters:

- **base_folder**: Folder to be scanned.

conan.tools.files.symlinks.remove_external_symlinks()

```
def remove_external_symlinks(conanfile, base_folder):
```

Remove the symlinks to files that point outside the 'base_folder', no matter if relative or absolute.

Parameters:

- **base_folder**: Folder to be scanned.

conan.tools.files.symlinks.remove_broken_symlinks()

```
def remove_broken_symlinks(conanfile, base_folder):
```

Remove the broken symlinks, no matter if relative or absolute.

Parameters:

- **base_folder**: Folder to be scanned.

conan.tools.files.AutoPackager

The `AutoPackager` together with the *package layouts* feature, allow to automatically package the files following the declared information in the `layout()` method:

It will copy:

- Files from `self.cpp.local.includedirs` to `self.cpp.package.includedirs`
- Files from `self.cpp.local.libdirs` to `self.cpp.package.libdirs`
- Files from `self.cpp.local.bindirs` to `self.cpp.package.bindirs`
- Files from `self.cpp.local.srcdirs` to `self.cpp.package.srcdirs`
- Files from `self.cpp.local.builddirs` to `self.cpp.package.builddirs`
- Files from `self.cpp.local.resdirs` to `self.cpp.package.resdirs`
- Files from `self.cpp.local.frameworkdirs` to `self.cpp.package.frameworkdirs`

The patterns of the files to be copied can be defined with the *.patterns* property of the `AutoPackager` instance. The default patterns are:

```
packager = AutoPackager(self)
packager.patterns.include == ["*.h", "*.hpp", "*.hxx"]
packager.patterns.lib == ["*.so", "*.so.*", "*.a", "*.lib", "*.dylib"]
packager.patterns.bin == ["*.exe", "*.dll"]
packager.patterns.src == []
packager.patterns.build == []
packager.patterns.res == []
packager.patterns.framework == []
```

Usage:

```
from conans import ConanFile
from conan.tools.files import AutoPackager

class Pkg(ConanFile):

    def layout(self):
        ...

    def package(self):
        packager = AutoPackager(self)
        packager.patterns.include = ["*.hpp", "*.h", "include3.h"]
        packager.patterns.lib = ["*.a"]
        packager.patterns.bin = ["*.exe"]
        packager.patterns.src = ["*.cpp"]
        packager.patterns.framework = ["sframe*", "bframe*"]
        packager.run()
```

conan.tools.layout

Warning: These tools are still **experimental** (so subject to breaking changes) but with very stable syntax. We encourage the usage of it to be prepared for Conan 2.0. The `layout()` feature will be fully functional only in the new build system integrations (*in the conan.tools space*). If you are using other integrations, they might not fully support this feature.

Predefined layouts

There are some pre-defined common *layouts*, ready to be simply used in recipes:

- `cmake_layout()`: *a layout for a typical CMake project*
- `vs_layout()`: a layout for a typical Visual Studio project
- `basic_layout()`: *a very basic layout for a generic project*

The pre-defined layouts define the Conanfile `.folders` and `.cpp` attributes with typical values. To check which are the values set by these pre-defined layouts please check the reference for the *layout()* method. For example in the `cmake_layout()` the source folder is called `"."`, meaning that Conan will expect the sources in the same directory where the conanfile is (most likely the project root, where a `CMakeLists.txt` file will be typically found). If you have a different folder where the `CMakeLists.txt` is located, you can use the `src_folder` argument:

```
from conan.tools.cmake import cmake_layout

def layout(self):
    cmake_layout(self, src_folder="mysrcfolder")
```

Even if this pre-defined layout doesn't suit your specific projects layout, checking how they implement their logic shows how you could implement your own logic (and probably put it in a common `python_require` if you are going to use it in multiple packages).

To learn more about the layouts and how to use them while developing packages, please check the *package layout* section.

basic_layout

Usage:

```
from conan.tools.layout import basic_layout

def layout(self):
    basic_layout(self)
```

The current layout implementation is very simple, basically sets a different build folder for different build_types and set the generators output folder inside the build folder. This way we avoid to clutter our project while working locally.

```
def basic_layout(conanfile, src_folder="."):
    conanfile.folders.build = "build"
    if conanfile.settings.get_safe("build_type"):
        conanfile.folders.build += "-{}".format(str(conanfile.settings.build_type).
↪lower())
    conanfile.folders.generators = os.path.join(conanfile.folders.build, "conan")
    conanfile.cpp.build.bindirs = ["."]
    conanfile.cpp.build.libdirs = ["."]
    conanfile.folders.source = src_folder
```

conan.tools.scm

Git

Warning: This tool is **experimental** and subject to breaking changes.

constructor

```
def __init__(self, conanfile, folder=".")
```

Construct a `Git` object, specifying the current directory, by default `"."`, the current working directory.

get_commit()

```
def get_commit(self)
```

Returns the current commit, with `git rev-list HEAD -n 1 -- <folder>`. The latest commit is returned, irrespective of local not commmited changes.

get_remote_url()

```
def get_remote_url(self, remote="origin")
```

Obtains the URL of the `remote` git remote repository, with `git remote -v`

commit_in_remote()

```
def commit_in_remote(self, commit, remote="origin")
```

Checks that the given commit exists in the remote, with `branch -r --contains <commit>` and checking an occurence of a branch in that remote exists.

is_dirty()

```
def is_dirty(self)
```

Returns if the current folder is dirty, running `git status -s`

get_repo_root()

```
def get_repo_root(self)
```

Returns the current repository top folder with `git rev-parse --show-toplevel`

clone()

```
def clone(self, url, target="")
```

Does a `git clone <url> <target>`

checkout()

```
def checkout(self, commit)
```

Checkouts the given commit

get_url_and_commit()

```
def get_url_and_commit(self, remote="origin")
```

This is an advanced method, that returns both the current commit, and the remote repository url. This method is intended to capture the current remote coordinates for a package creation, so that can be used later to build again from sources from the same commit. This is the behavior:

- If the repository is dirty, it will raise an exception. Doesn't make sense to capture coordinates of something dirty, as it will not be reproducible. If there are local changes, and the user wants to test a local `conan create`, should commit the changes first (locally, not push the changes).
- If the repository is not dirty, but the commit doesn't exist in the given remote, the method will return that commit and the URL of the local user checkout. This way, a package can be `conan create` created locally, testing everything works, before pushing some changes to the remote.
- If the repository is not dirty, and the commit exists in the specified remote, it will return that commit and the url of the remote.

Example: Implementing the `scm` feature

This example is the new way to implement the `scm` feature (to be removed in Conan 2.0), using this new `Git` capabilities.

Assume we have this project with this layout, in a git repository:

```
├── conanfile.py
├── CMakeLists.txt
├── src
    └── hello.cpp
```

And the conanfile.py is:

```
import os
from conan import ConanFile
from conan.tools.scm import Git
from conan.tools.files import load, update_conandata


class Pkg(ConanFile):
    name = "pkg"
    version = "0.1"

    def export(self):
        git = Git(self, self.recipe_folder)
        scm_url, scm_commit = git.get_url_and_commit()
        # we store the current url and commit in conandata.yml
        update_conandata(self, {"sources": {"commit": scm_commit, "url": scm_url}})
```

(continues on next page)

(continued from previous page)

```
    def layout(self):
        self.folders.source = "."

    def source(self):
        # we recover the saved url and commit from conandata.yml and use them to get␣
↪sources
        git = Git(self)
        sources = self.conan_data["sources"]
        git.clone(url=sources["url"], target=".")
        git.checkout(commit=sources["commit"])

    def build(self):
        # build() will have access to the sources, obtained with the clone in source()
        cmake = os.path.join(self.source_folder, "CMakeLists.txt")
        hello = os.path.join(self.source_folder, "src/hello.cpp")
        self.output.info("MYCMAKE-BUILD: {}".format(load(self, cmake)))
        self.output.info("MYFILE-BUILD: {}".format(load(self, hello)))
```

This conanfile does:

- In the `export()` method, it captures the url and commit, according to the rules of `get_url_and_commit()` above
- The url and commit are saved in the `conandata.yml`
- These two first steps happen in the `conan export` or first part of `conan create` when the recipe is exported to the cache.
- When the recipe is building from sources in the cache, it will call the `source()` method which will clone and checkout from the user folder if the commit is only local or from the git remote if the commit is remote too.

This `conan create` flow is not recommended for continuous usage. To develop and test, users should use the local flow (`conan install` + build system). Only in the last stage, to check that things are looking good to push, the user can do a local commit, and before pushing, do a `conan create` to check locally.

Version

> **Warning:** This tool is **experimental** and subject to breaking changes.

constructor

```
def __init__(self, value: str):
```

Construct a `Version` object from a string. It supports basic version comparison between objects, like for example:

```
compiler_lower_than_12 = Version(str(self.settings.compiler.version)) < "12.0"
```

Please note this is not an implementation of semver, as users may use any pattern in their versions. It is just a helper to parse "." or "-" and compare taking into account integers when possible.

conan.tools.build

conan.tools.build.cross_building()

```
def cross_building(conanfile=None, skip_x64_x86=False):
```

Check it we are cross building comparing the *build* and *host* settings. Returns `True` in the case that we are cross-building.

Parameters:

- **conanfile**: Conanfile object, use always `self`.
- **skip_x64_x86**: Will not consider the as cross-building the case of building in 64 bit architecture for 32 bit architecture, like `build_arch=x86_64` and `host_arch=x86` for example.

conan.tools.build.can_run()

```
def can_run(conanfile):
```

Validates whether is possible to run a non-native app on the same architecture. It returns the configuration value for `tools.build.cross_building:can_run` if exists, otherwise, it returns `False` if we are cross-building, else, `True`.

It's an useful feature for the case your architecture can run more than one target. For instance, Mac M1 machines can run both `armv8` and `x86_64`.

Parameters:

- **conanfile**: Conanfile object, use always `self`.

18.3.4 Dependencies

Introduced in Conan 1.38.

Warning: These tools are **experimental** and subject to breaking changes.

Note: This is an advanced feature. Most users will not need to use it, it is intended for developing new build system integrations and similar purposes. For defining dependencies between packages, check the `requires`, `tool_requires` and other attributes

Conan recipes provide access to their dependencies via the `self.dependencies` attribute. This attribute is extensively used by generators like `CMakeDeps` or `MSBuildDeps` to generate the necessary files for the build.

This section documents the `self.dependencies` attribute, as it might be used by users both directly in recipe or indirectly to create custom build integrations and generators.

Dependencies interface

It is possible to access each one of the individual dependencies of the current recipe, with the following syntax:

```
class Pkg(ConanFile):
    requires = "openssl/0.1"

    def generate(self):
        openssl = self.dependencies["openssl"]
        # access to members
        openssl.ref.version
        openssl.ref.revision # recipe revision
        openssl.options
        openssl.settings
```

Some **important** points:

- All the information is **read only**. Any attempt to modify dependencies information is an error and can raise at any time, even if it doesn't raise yet.
- It is not possible either to call any methods or any attempt to reuse code from the dependencies via this mechanism.
- This information does not exist in some recipe methods, only in those methods that evaluate after the full dependency graph has been computed. It will not exist in `configure()`, `config_options`, `export()`, `export_source()`, `set_name()`, `set_version()`, `requirements()`, `build_requirements()`, `system_requirements()`, `source()`, `init()`, `layout()`. Any attempt to use it in these methods can raise an error at any time.
- At the moment, this information should only be used in `generate()` and `validate()` methods. Any other use, please submit a Github issue.

Not all fields of the dependency conanfile are exposed, the current fields are:

- package_folder: The folder location of the dependency package binary
- ref: an object that contains `name`, `version`, `user`, `channel` and `revision` (recipe revision)
- pref: an object that contains `ref`, `package_id` and `revision` (package revision)
- buildenv_info: `Environment` object with the information of the environment necessary to build
- runenv_info: `Environment` object with the information of the environment necessary to run the app
- cpp_info: includedirs, libdirs, etc for the dependency.
- settings: The actual settings values of this dependency
- settings_build: The actual build settings values of this dependency
- options: The actual options values of this dependency
- context: The context (build, host) of this dependency
- conf_info: Configuration information of this dependency, intended to be applied to consumers.
- dependencies: The transitive dependencies of this dependency
- is_build_context: Return `True` if `context == "build"`.

Iterating dependencies

It is possible to iterate in a dict-like fashion all dependencies of a recipe. Take into account that `self.dependencies` contains all the current dependencies, both direct and transitive. Every upstream dependency of the current one that has some effect on it, will have an entry in this `self.dependencies`.

Iterating the dependencies can be done as:

```
requires = "zlib/1.2.11", "poco/1.9.4"

def generate(self):
    for require, dependency in self.dependencies.items():
        self.output.info("Dependency is direct={}: {}".format(require.direct, dependency.
↪ref))
```

will output:

```
conanfile.py (hello/0.1): Dependency is direct=True: zlib/1.2.11
conanfile.py (hello/0.1): Dependency is direct=True: poco/1.9.4
conanfile.py (hello/0.1): Dependency is direct=False: pcre/8.44
conanfile.py (hello/0.1): Dependency is direct=False: expat/2.4.1
conanfile.py (hello/0.1): Dependency is direct=False: sqlite3/3.35.5
conanfile.py (hello/0.1): Dependency is direct=False: openssl/1.1.1k
conanfile.py (hello/0.1): Dependency is direct=False: bzip2/1.0.8
```

Where the `require` dictionary key is a "requirement", and can contain specifiers of the relation between the current recipe and the dependency. At the moment they can be:

- `require.direct`: boolean, `True` if it is direct dependency or `False` if it is a transitive one.
- `require.build`: boolean, `True` if it is a `build_require` in the build context, as `cmake`.
- `require.test`: boolean, `True` if its a `build_require` in the host context (defined with `self.test_requires()`), as `gtest`.

The `dependency` dictionary value is the read-only object described above that access the dependency attributes.

The `self.dependencies` contains some helpers to filter based on some criteria:

- `self.dependencies.host`: Will filter out requires with `build=True`, leaving regular dependencies like `zlib` or `poco`.
- `self.dependencies.direct_host`: Will filter out requires with `build=True` or `direct=False`
- `self.dependencies.build`: Will filter out requires with `build=False`, leaving only `tool_requires` in the build context, as `cmake`.
- `self.dependencies.direct_build`: Will filter out requires with `build=False` or `direct=False`
- `self.dependencies.test`: Will filter out requires with `build=True` or with `test=False`, leaving only test requirements as `gtest` in the host context.

They can be used in the same way:

```
requires = "zlib/1.2.11", "poco/1.9.4"

def generate(self):
    cmake = self.dependencies.direct_build["cmake"]
    for require, dependency in self.dependencies.build.items():
        # do something, only build deps here
```

Dependencies cpp_info interface

The cpp_info interface is heavily used by build systems to access the data. This object defines global and per-component attributes to access information like the include folders:

```
def generate(self):
    cpp_info = self.dependencies["mydep"].cpp_info
    cpp_info.includedirs
    cpp_info.libdirs

    cpp_info.components["mycomp"].includedirs
    cpp_info.components["mycomp"].libdirs
```

These are the defined attributes in cpp_info. All the paths are typically relative paths to the root of the package folder that contains the dependency artifacts:

```
# ###### DIRECTORIES
self.includedirs = None  # Ordered list of include paths
self.srcdirs = None  # Ordered list of source paths
self.libdirs = None  # Directories to find libraries
self.resdirs = None  # Directories to find resources, data, etc
self.bindirs = None  # Directories to find executables and shared libs
self.builddirs = None
self.frameworkdirs = None

# ##### FIELDS
self.system_libs = None  # Ordered list of system libraries
self.frameworks = None  # Macos .framework
self.libs = None  # The libs to link against
self.defines = None  # preprocessor definitions
self.cflags = None  # pure C flags
self.cxxflags = None  # C++ compilation flags
self.sharedlinkflags = None  # linker flags
self.exelinkflags = None  # linker flags
self.objects = None  # objects to link
```

18.3.5 Python requires

It is possible to reuse python code existing in other *conanfile.py* recipes with the python_requires functionality, doing something like:

```
from conans import ConanFile

class Pkg(ConanFile):
    python_requires = "pyreq/0.1@user/channel"

    def build(self):
        v = self.python_requires["pyreq"].module.myvar # myvar defined in pyreq's␣
↪conanfile.py
        f = self.python_requires["pyreq"].module.myfunct() # myfunct defined in pyreq's␣
↪conanfile.py
        self.output.info("%s,%s" % (v, f))
```

See this section: *Python requires: reusing python code in recipes*

18.3.6 Output and Running

Output contents

Use the *self.output* to print contents to the output.

```
self.output.success("This is a good, should be green")
self.output.info("This is a neutral, should be white")
self.output.warn("This is a warning, should be yellow")
self.output.error("Error, should be red")
self.output.rewrite_line("for progress bars, issues a cr")
```

Check the source code. You might be able to produce different outputs with different colors.

Running commands

```
run(self, command, output=True, cwd=None, win_bash=False, subsystem=None, msys_
↪mingw=True,
    ignore_errors=False, run_environment=False, with_login=True, env="conanbuild"):
```

`self.run()` is a helper to run system commands and throw exceptions when errors occur, so that command errors do not pass unnoticed. It is just a wrapper for `subprocess.call()`.

When the environment variable `CONAN_PRINT_RUN_COMMANDS` is set to true (or its equivalent `print_run_commands` *conan.conf* configuration variable, under `[general]`) then all the invocations of `self.run()` will print to output the command to be executed.

The `command` can be specified as a string which is passed to the system shell. Alternatively it can be specified as a sequence of strings, the first of which is interpreted as the name of the program to be executed and the remaining ones are passed as arguments. Unless you are relying on shell-specific features such as redirection or command substitution, providing a sequence of strings is generally preferred as it allows Conan to take care of any required escaping and quoting of arguments (e.g. to permit spaces in file names).

Optional parameters:

- **output (Optional, Defaulted to True) When True it will write in stdout.**
  You can pass any stream that accepts a `write` method like a `six.StringIO()`:

  ```
  from six import StringIO  # Python 2 and 3 compatible
  mybuf = StringIO()
  self.run("mycommand", output=mybuf)
  self.output.warn(mybuf.getvalue())
  ```

- **cwd** (Optional, Defaulted to `.` current directory): Current directory to run the command.
- **win_bash** (Optional, Defaulted to `False`): When True, it will run the configure/make commands inside a bash.
- **subsystem** (Optional, Defaulted to `None` will autodetect the subsystem): Used to escape the command according to the specified subsystem.
- **msys_mingw** (Optional, Defaulted to `True`) If the specified subsystem is MSYS2, will start it in MinGW mode (native windows development).

- **ignore_errors** (Optional, Defaulted to `False`). This method raises an exception if the command fails. If `ignore_errors=True`, it will not raise an exception. Instead, the user can use the return code to check for errors.
- **run_environment** (Optional, Defaulted to `False`). Applies a `RunEnvironment`, so the environment variables PATH, LD_LIBRARY_PATH and DYLIB_LIBRARY_PATH are defined in the command execution adding the values of the "lib" and "bin" folders of the dependencies. Allows executables to be easily run using shared libraries from its dependencies.
- **with_login** (Optional, Defaulted to `True`): Pass the `--login` flag to **bash** command when using `win_bash` parameter. This might come handy when you don't want to create a fresh user session for running the command.
- **env** (Optional, Defaulted to `conanbuild`): the environment or list of environment activations scripts to pre-pend to the given `command`. If `None`, no environment will be pre-pended.

Requiring a Conan version for the recipe

Available since: 1.28.0

A required Conan version can be declared in the *conanfile.py* using `required_conan_version` to throw an error when the Conan version installed does not meet the criteria established by the variable. To add `required_conan_version` to a *conanfile.py* just declare it before the recipe class definition:

```
from conans import ConanFile

required_conan_version = ">=1.27.1"

class Pkg(ConanFile):
    settings = "os", "compiler", "arch", "build_type"
    ...
```

It is also possible to declare `required_conan_version` at configuration level for the whole client adding it to the *conan.conf* file.

18.4 Generators

Generators are specific components that provide the information of dependencies calculated by Conan in a suitable format for a build system. They normally provide Conan users with a *conanbuildinfo.XXX* file that can be included or injected to the specific build system. The file generated contains information of dependencies in form of different variables and sometimes function helpers too.

You can specify a generator in:

- The `[generators]` section from *conanfile.txt*.
- The `generators` attribute in *conanfile.py*.
- The command line when installing dependencies **`conan install --generator`**.

Available generators:

18.4.1 cmake

This is the reference page for `cmake` generator. Go to *Integrations/CMake* if you want to learn how to integrate your project or recipes with CMake.

It generates a file named *conanbuildinfo.cmake* and declares some variables and methods.

Variables in *conanbuildinfo.cmake*

- **Package declared variables**:

  For each requirement *conanbuildinfo.cmake* file declares the following variables. Where `<PKG-NAME>` is the placeholder for the name of the require in uppercase (`ZLIB` for `zlib/1.2.8@lasote/stable`) in `cpp_info.names["cmake_find_package"]` or `cpp_info.names["cmake_find_package_multi"]` if specified:

NAME	VALUE
CONAN_<PKG-NAME>_ROOT	Abs path to root package folder.
CONAN_INCLUDE_DIRS_<PKG-NAME>	Header's folders
CONAN_LIB_DIRS_<PKG-NAME>	Library folders (default {CONAN_<PKG-NAME>_ROOT}/lib)
CONAN_BIN_DIRS_<PKG-NAME>	Binary folders (default {CONAN_<PKG-NAME>_ROOT}/bin)
CONAN_SRC_DIRS_<PKG-NAME>	Sources folders
CONAN_LIBS_<PKG-NAME>	Library names to link (package libs, system libs and frameworks)
CONAN_PKG_LIBS_<PKG-NAME>	Package library names to link
CONAN_SYSTEM_LIBS_<PKG-NAME>	System library names to link
CONAN_DEFINES_<PKG-NAME>	Library defines
CONAN_COMPILE_DEFINITIONS_<PKG-NAME>	Compile definitions
CONAN_CXX_FLAGS_<PKG-NAME>	CXX flags
CONAN_SHARED_LINK_FLAGS_<PKG-NAME>	Shared link flags
CONAN_C_FLAGS_<PKG-NAME>	C flags
CONAN_FRAMEWORKS_<PKG-NAME>	Frameworks names to use them in *find_library()*
CONAN_FRAMEWORKS_FOUND_<PKG-NAME>	Frameworks found after using CONAN_FRAMEWORKS in *find_library()*
CONAN_FRAMEWORK_PATHS_<PKG-NAME>	Framework folders to locate the frameworks (OSX)

- **Global declared variables**:

  This generator also declares some global variables with the aggregated values of all our requirements. The values are ordered in the right order according to the dependency tree.

NAME	VALUE
CONAN_INCLUDE_DIRS	Aggregated header's folders
CONAN_LIB_DIRS	Aggregated library folders
CONAN_BIN_DIRS	Aggregated binary folders
CONAN_SRC_DIRS	Aggregated sources folders
CONAN_LIBS	Aggregated library names to link (with system libs and frameworks)
CONAN_SYSTEM_LIBS	Aggregated system libraries names to link
CONAN_DEFINES	Aggregated library defines
CONAN_COMPILE_DEFINITIONS	Aggregated compile definitions
CONAN_CXX_FLAGS	Aggregated CXX flags
CONAN_SHARED_LINK_FLAGS	Aggregated Shared link flags
CONAN_C_FLAGS	Aggregated C flags
CONAN_FRAMEWORKS	Aggregated frameworks to be found with *find_library()* (OSX)
CONAN_FRAMEWORKS_FOUND	Aggregated found frameworks after *find_library()* call (OSX)
CONAN_FRAMEWORK_PATHS	Aggregated framework folders (OSX)
CONAN_BUILD_MODULES	Aggregated paths for build module files (like *.cmake*)

- **User information declared variables**:

  If any of the requirements is filling the *user_info* object in the *package_info* method a set of variables will be declared following this naming:

NAME	VALUE
CONAN_USER_<PKG-NAME>_<VAR-NAME>	User declared value

  Where `<PKG-NAME>` means the name of the requirement in uppercase and `<VAR-NAME>` the variable name. For example, if this recipe declares:

```
class MyLibConan(ConanFile):
    name = "mylib"
    version = "1.6.0"

    # ...

    def package_info(self):
        self.user_info.var1 = 2
```

  Other library requiring `mylib` and using this generator will get:

Listing 52: *conanbuildinfo.cmake*

```
# ...
set(CONAN_USER_MYLIB_var1 "2")
```

Macros available in *conanbuildinfo.cmake*

conan_basic_setup()

This is a helper and general purpose macro that uses all the macros below to set all the CMake variables according to the Conan generated variables. See the macros below for detailed information.

```
macro(conan_basic_setup)
    set(options TARGETS NO_OUTPUT_DIRS SKIP_RPATH KEEP_RPATHS SKIP_STD SKIP_FPIC)
```

Parameters:

- `TARGETS` (Optional): Setup all the CMake variables by target (only CMake > 3.1.2). Activates the call to the macro `conan_target_link_libraries()`.
- `NO_OUTPUT_DIRS` (Optional): Do not adjust the build output directories. Deactivates the call to the macro [`conan_output_dirs_setup()`](#conan_output_dirs_setup).
- `SKIP_RPATH` (Optional): **[DEPRECATED]** Use `KEEP_RPATHS` instead. Activate `CMAKE_SKIP_RPATH` variable in OSX.
- `KEEP_RPATHS` (Optional): Do not adjust the `CMAKE_SKIP_RPATH` variable in OSX. Activates the call to the macro `conan_set_rpath()`
- `SKIP_STD` (Optional): Do not adjust the C++ standard flag in `CMAKE_CXX_FLAGS`. Deactivates the call to the macro `conan_set_std()`.
- `SKIP_FPIC` (Optional): Do not adjust the `CMAKE_POSITION_INDEPENDENT_CODE` flag. Deactivates the call to the macro `conan_set_fpic()`.

Note: You can also call each of the following macros individually instead of using the `conan_basic_setup()`.

conan_target_link_libraries()

Helper to link all libraries to a specified target.

These targets are:

- A `CONAN_PKG::<PKG-NAME>` target per package in the dependency graph. This is an `IMPORTED INTERFACE` target. `IMPORTED` because it is external, a pre-compiled library. `INTERFACE`, because it doesn't necessarily match a library, it could be a header-only library, or the package could even contain several libraries. It contains all the properties (include paths, compile flags, etc.) that are defined in the consumer. It contains all the properties (include paths, compile flags, etc.) that are defined in the `package_info()` method of the recipe.
- Inside each package a `CONAN_LIB::<PKG-NAME>_<LIB-NAME>` target will be generated for each library. Its type is `IMPORTED UNKNOWN` and its main purpose is to provide a correct link order. Their only properties are the location and the dependencies.
- A `CONAN_PKG` depends on every `CONAN_LIB` that belongs to it, and to its direct public dependencies (e.g. other `CONAN_PKG` targets from its requirements).

- Each `CONAN_LIB` depends on the direct public dependencies `CONAN_PKG` targets of its container package. This guarantees correct link order.

conan_check_compiler()

Checks that your compiler matches the one declared in settings.

This method can be disabled setting the *CONAN_DISABLE_CHECK_COMPILER* variable.

conan_output_dirs_setup()

Adjusts each *CMAKE_RUNTIME_OUTPUT_DIRECTORY* variable to be `${CMAKE_CURRENT_BINARY_DIR}/bin` and each `CMAKE_ARCHIVE_OUTPUT_DIRECTORY` and `CMAKE_LIBRARY_OUTPUT_DIRECTORY` variable to be `${CMAKE_CURRENT_BINARY_DIR}/lib`.

Calling this method makes writing the `package()` method for recipies easier. All artifacts will always be found in the same location. Otherwise, they may be found in different locations depending on your build environment (eg Linux vs Windows).

conan_set_find_library_paths()

Sets `CMAKE_INCLUDE_PATH` and `CMAKE_LIBRARY_PATH`.

conan_global_flags()

Sets the corresponding variables to CMake's `include_directories()` and `link_directories()`. You can enable the variable *CONAN_SYSTEM_INCLUDES* in order to get directories included with the *SYSTEM* option.

conan_define_targets()

Defines the targets for each dependency (target flags instead of global flags).

conan_set_rpath()

Sets `CMAKE_SKIP_RPATH=1` in the case of working in OSX.

conan_set_vs_runtime()

Adjusts the runtime flags `/MD`, `/MDd`, `/MT` or `/MTd` for Visual Studio.

conan_set_std()

Sets `CMAKE_CXX_STANDARD` and `CMAKE_CXX_EXTENSIONS` to the appropriate values.

conan_set_libcxx()

Adjusts the standard library flags (`libc++`, `libstdc++`, `libstdc++11`) in `CMAKE_CXX_FLAGS`.

conan_set_find_paths()

Adjusts `CMAKE_MODULE_PATH` and `CMAKE_PREFIX_PATH` to the values of `deps_cpp_info.build_paths`.

conan_include_build_modules()

Includes CMake files declared in `CONAN_BUILD_MODULES` using the `include(...)` directive. This loads the functions or macros that packages may export and makes them available for usage in the consumers *CMakeLists.txt*.

conan_find_apple_frameworks(FRAMEWORKS_FOUND FRAMEWORKS)

Find framework library names provided in *${FRAMEWORKS}* using *find_library()* and return the found values in *FRAMEWORKS_FOUND*.

Input variables for *conanbuildinfo.cmake*

CONAN_CMAKE_SILENT_OUTPUT

Default to: `FALSE`

Activate it to silence the Conan message output.

CONAN_DISABLE_CHECK_COMPILER

Default to: `FALSE`

Deactivates the check of the compiler done with the method *conan_check_compiler()*.

18.4.2 cmake_multi

This is the reference page for `cmake_multi` generator. Go to *Integrations/CMake* if you want to learn how to integrate your project or recipes with CMake.

This generator will create 3 files with the general information and specific Debug/Release ones:

- *conanbuildinfo_release.cmake*: Variables adjusted only for build type Release
- *conanbuildinfo_debug.cmake*: Variables adjusted only for build type Debug
- *conanbuildinfo_multi.cmake*: Which includes the other two and enables its use and has more generic variables and macros.

Variables in *conanbuildinfo_release.cmake*

Same as *conanbuildinfo.cmake* with suffix `_RELEASE`

Variables in *conanbuildinfo_debug.cmake*

Same as *conanbuildinfo.cmake* with suffix `_DEBUG`

Macros available in *conanbuildinfo_multi.cmake*

conan_basic_setup()

This is a helper and general purpose macro that uses all the macros below to set all the CMake variables according to the Conan generated variables. See the macros below for detailed information.

```
macro(conan_basic_setup)
    set(options TARGETS NO_OUTPUT_DIRS SKIP_RPATH KEEP_RPATHS SKIP_STD SKIP_FPIC)
```

Parameters:

- `TARGETS` (Optional): Setup all the CMake variables by target (only CMake > 3.1.2). Activates the call to the macro `conan_target_link_libraries()`.
- `NO_OUTPUT_DIRS` (Optional): This variable has no effect and it works as if it was activated by default (does not se fixed output directories and uses the default ones designated by CMake).
- `SKIP_RPATH` (Optional): **[DEPRECATED]** Use `KEEP_RPATHS` instead. Activate `CMAKE_SKIP_RPATH` variable in OSX.
- `KEEP_RPATHS` (Optional): Do not adjust the `CMAKE_SKIP_RPATH` variable in OSX. Activates the call to the macro `conan_set_rpath()`
- `SKIP_STD` (Optional): Do not adjust the C++ standard flag in `CMAKE_CXX_FLAGS`. Deactivates the call to the macro `conan_set_std()`.
- `SKIP_FPIC` (Optional): Do not adjust the `CMAKE_POSITION_INDEPENDENT_CODE` flag. Deactivates the call to the macro `conan_set_fpic()`.

Note: You can also call each of the following macros individually instead of using the `conan_basic_setup()`.

conan_target_link_libraries()

Helper to link all libraries to a specified target.

These targets are:

- A `CONAN_PKG::<PKG-NAME>` target per package in the dependency graph. This is an `IMPORTED INTERFACE` target. `IMPORTED` because it is external, external, a pre-compiled library. `INTERFACE`, because it doesn't necessarily match a library, it could be a header-only library, or the package could even contain several libraries. It contains all the properties (include paths, compile flags, etc.) that are defined in the consumer. It contains all the properties (include paths, compile flags, etc.) that are defined in the `package_info()` method of the recipe.

- Inside each package a `CONAN_LIB::<PKG-NAME>_<LIB-NAME>` target will be generated for each library. Its type is `IMPORTED UNKNOWN` and its main purpose is to provide a correct link order. Their only properties are the location and the dependencies.
- A `CONAN_PKG` depends on every `CONAN_LIB` that belongs to it, and to its direct public dependencies (e.g. other `CONAN_PKG` targets from its requirements).
- Each `CONAN_LIB` depends on the direct public dependencies `CONAN_PKG` targets of its container package. This guarantees correct link order.

conan_check_compiler()

Checks that your compiler matches the one declared in settings.

conan_output_dirs_setup()

Adjust the *bin/* and *lib/* output directories.

conan_global_flags()

Set the corresponding variables to CMake's `include_directories()` and `link_directories()`. You can enable the variable *CONAN_SYSTEM_INCLUDES* in order to get directories included with the *SYSTEM* option.

conan_define_targets()

Define the targets for each dependency (target flags instead of global flags).

conan_set_rpath()

Set `CMAKE_SKIP_RPATH=1` in the case of working in OSX.

conan_set_vs_runtime()

Adjust the runtime flags `/MD`, `/MDd`, `/MT` or `/MTd` for Visual Studio.

conan_set_std()

Set `CMAKE_CXX_STANDARD` and `CMAKE_CXX_EXTENSIONS` to the appropriate values.

conan_set_libcxx()

Adjust the standard library flags (`libc++`, `libstdc++`, `libstdc++11`) in `CMAKE_CXX_FLAGS`.

conan_set_find_paths()

Adjust `CMAKE_MODULE_PATH` and `CMAKE_PREFIX_PATH` to the values of `deps_cpp_info.build_paths`.

conan_include_build_modules()

Includes CMake files declared in `CONAN_BUILD_MODULES` using the `include(...)` directive. This loads the functions or macros that packages may export and makes them available for usage in the consumers *CMakeLists.txt*.

conan_find_apple_frameworks(FRAMEWORKS_FOUND FRAMEWORKS)

Find framework library names provided in *${FRAMEWORKS}* using *find_library()* and return the found values in *FRAMEWORKS_FOUND*.

Input variables for *conanbuildinfo_multi.cmake*

CONAN_CMAKE_SILENT_OUTPUT

Default to: `FALSE`

Activate it to silence the Conan message output.

18.4.3 cmake_paths

This is the reference page for `cmake_paths` generator. Go to *Integrations/CMake* if you want to learn how to integrate your project or recipes with CMake.

It generates a file named `conan_paths.cmake` and declares these variables:

Variables in *conan_paths.cmake*

NAME	VALUE
CMAKE_MODULE_P/	Containing all requires root folders, any declared *self.cpp_info.builddirs* and the current directory of this file
CMAKE_PREFIX_PAT	Containing all requires root folders, any declared *self.cpp_info.builddirs* and the current directory of this file
CONAN_<PKG-NAME>_ROOT	For each dep, the root folder, being XXX the dep name uppercase. Useful when a *.cmake* is patched with *cmake.patch_config_paths()*

Where `<PKG-NAME>` is the placeholder for the name of the require in uppercase (`ZLIB` for `zlib/1.2.11`) or the one declared in `cpp_info.names["cmake_paths"]` if specified.

18.4.4 cmake_find_package

This is the reference page for `cmake_find_package` generator. Go to *Integrations/CMake* if you want to learn how to integrate your project or recipes with CMake.

The `cmake_find_package` generator creates a file for each requirement specified in the conanfile.

The name of the files follow the pattern `Find<PKG-NAME>.cmake`. So for the `asio/1.14.0` package, a `Findasio.cmake` file will be generated.

Variables in Find<PKG-NAME>.cmake

Being `<PKG-NAME>` the package name used in the reference (by default) or the one declared in `cpp_info.names["cmake_find_package"]` if specified:

NAME	VALUE
<PKG-NAME>_FOUND	Set to 1
<PKG-NAME>_VERSION	Package version
<PKG-NAME>_INCLUDE_DIRS	Containing all the include directories of the package
<PKG-NAME>_INCLUDES	Same as the XXX_INCLUDE_DIRS
<PKG-NAME>_DEFINITIONS	Definitions of the library
<PKG-NAME>_LIBS	Library paths to link
<PKG-NAME>_LIBRARIES	Same as <PKG-NAME>_LIBS
<PKG-NAME>_BUILD_MODULES	List of CMake module files with functionalities for consumers
<PKG-NAME>_SYSTEM_LIBS	System libraries to link
<PKG-NAME>_FRAMEWORKS	Framework names to do a *find_library()*
<PKG-NAME>_FRAMEWORKS_FOUND	Found frameworks to link with after *find_library()*
<PKG-NAME>_FRAMEWORK_DIRS	Framework directories to perform the *find_library()* of <PKG-NAME>_FRAMEWORKS

This file uses *<PKG-NAME>_BUILD_MODULES* values to include the files using the *include(...)* CMake directive after the targets are created. This makes functions or utilities exported by the package available for consumers just by setting *find_package(<PKG-NAME>)* in the *CMakeLists.txt*.

Moreover, this also adjusts *CMAKE_MODULE_PATH* and *CMAKE_PREFIX_PATH* to the values declared by the package in `cpp_info.buildirs`, so modules in those directories can be found.

Targets in Find<PKG-NAME>.cmake

A target named `<PKG-NAME>::<PKG-NAME>` target is generated with the following properties adjusted:

- `INTERFACE_INCLUDE_DIRECTORIES`: Containing all the include directories of the package.
- `INTERFACE_LINK_LIBRARIES`: Library paths to link.
- `INTERFACE_COMPILE_DEFINITIONS`: Definitions of the library.

The targets are transitive. So, if your project depends on a packages `A` and `B`, and at the same time `A` depends on `C`, the `A` target will contain automatically the properties of the `C` dependency, so in your *CMakeLists.txt* file you only need to `find_package(A)` and `find_package(B)`.

Components

If a recipe uses components, the targets generated will be <PKG-NAME>::<COMP-NAME> with the following properties adjusted:

- INTERFACE_INCLUDE_DIRECTORIES: Containing all the include directories of the component.
- INTERFACE_LINK_DIRECTORIES: Containing all the lib directories of the component.
- INTERFACE_LINK_LIBRARIES: Containing the targets to link the component to (includes component's libraries and dependencies).
- INTERFACE_COMPILE_DEFINITIONS: Containing the definitions of the component.
- INTERFACE_COMPILE_OPTIONS: Containing the compile options of the component.

Moreover, a global target <PKG-NAME>::<PKG-NAME> will be declared with the following properties adjusted:

- INTERFACE_LINK_LIBRARIES: Containing all the component targets to link the global target to (includes package's components only).

Important: Name conflicts: If the name of the global target is the same for different packages, Conan will aggregate into this global target all the components from all those different packages. This means that this global target will contain information coming from different packages. For the components themselves, a name conflict will result in one of them being inaccessible without further notice.

18.4.5 cmake_find_package_multi

> **Warning:** This is an **experimental** feature subject to breaking changes in future releases.

This is the reference page for `cmake_find_package_multi` generator. Go to *Integrations/CMake* if you want to learn how to integrate your project or recipes with CMake.

Generated files

For each conan package in your graph, it will generate 2 files and 1 more per different `build_type`. Being `<PKG-NAME>` the package name used in the reference (by default) or the one declared in `cpp_info.names["cmake_find_package_multi"]` if specified:

NAME	CONTENTS
<PKG-NAME>Config.cmake / <PKG-NAME>-config.cmake	It includes the <PKG-NAME>Targets.cmake and call find_dependency for each dep
<PKG-NAME>ConfigVersion.cmake / <PKG-NAME>-config-version.cmake	Package version file for each dep
<PKG-NAME>Targets.cmake	It includes the files *<PKG-NAME>Targets-<BUILD-TYPE>.cmake*
<PKG-NAME>Targets-debug.cmake	Specific information for the Debug configuration
<PKG-NAME>Targets-release.cmake	Specific information for the Release configuration
<PKG-NAME>Targets-relwithdebinfo.cmake	Specific information for the RelWithDebInfo configuration
<PKG-NAME>Targets-minsizerel.cmake	Specific information for the MinSizeRel configuration

Targets

A target named `<PKG-NAME>::<PKG-NAME>` target is generated with the following properties adjusted:

- `INTERFACE_INCLUDE_DIRECTORIES`: Containing all the include directories of the package.
- `INTERFACE_LINK_LIBRARIES`: Library paths to link.
- `INTERFACE_COMPILE_DEFINITIONS`: Definitions of the library.
- `INTERFACE_COMPILE_OPTIONS`: Compile options of the library.

The targets contains multi-configuration properties, for example, the compile options property is declared like this:

```
set_property(TARGET <PKG-NAME>::<PKG-NAME>
         PROPERTY INTERFACE_COMPILE_OPTIONS
             $<$<CONFIG:Release>:${<PKG-NAME>_COMPILE_OPTIONS_RELEASE_LIST}>
             $<$<CONFIG:RelWithDebInfo>:${<PKG-NAME>_COMPILE_OPTIONS_RELWITHDEBINFO_
```

(continues on next page)

(continued from previous page)

```
↪LIST}}>
            $<$<CONFIG:MinSizeRel>:${{<PKG-NAME>_COMPILE_OPTIONS_MINSIZEREL_LIST}}>
            $<$<CONFIG:Debug>:${{<PKG-NAME>_COMPILE_OPTIONS_DEBUG_LIST}}>)
```

The targets are also transitive. So, if your project depends on a packages `A` and `B`, and at the same time `A` depends on `C`, the `A` target will contain automatically the properties of the `C` dependency, so in your *CMakeLists.txt* file you only need to `find_package(A CONFIG)` and `find_package(B CONFIG)`.

Important: Add the `CONFIG` option to `find_package` so that *module mode* is explicitly skipped by CMake. This helps to solve issues when there is for example a `Find<PKG-NAME>.cmake` file in CMake's default modules directory that could be loaded instead of the `<PKG-NAME>Config.cmake` generated by Conan.

You also need to adjust CMAKE_PREFIX_PATH and CMAKE_MODULE_PATH so CMake can locate all the `<PKG-NAME>Config.cmake/<PKG-NAME>-config.cmake` files: The `CMAKE_PREFIX_PATH` is used by the `find_package` and the `CMAKE_MODULE_PATH` is used by the `find_dependency` calls that locates the transitive dependencies.

The *<PKG-NAME>Targets-.cmake* files use *<PKG-NAME>_BUILD_MODULES_<BUILD-TYPE>* values to include the files using the *include(...)* CMake directive after the targets are created. This makes functions or utilities exported by the package available for consumers just by setting *find_package(<PKG-NAME>)* in the *CMakeLists.txt*.

Moreover, this also adjusts *CMAKE_MODULE_PATH* and *CMAKE_PREFIX_PATH* to the values declared by the package in `cpp_info.buildirs`, so modules in those directories can be found.

Components

If a recipe uses *components*, the targets generated will be `<PKG-NAME>::<COMP-NAME>` with the following properties adjusted. Being `<COMP-NAME>` the dictionary key used to declare the component or the one declared in `cpp_info.names["cmake_find_package_multi"]` or the alternative name declared in `cpp_info.components["comp_name"].names["cmake_find_package_multi"]` if specified:

- `INTERFACE_INCLUDE_DIRECTORIES`: Containing all the include directories of the component.
- `INTERFACE_LINK_LIBRARIES`: Containing the targets to link the component to (includes component's libraries and dependencies).
- `INTERFACE_COMPILE_DEFINITIONS`: Containing the definitions of the component.
- `INTERFACE_COMPILE_OPTIONS`: Containing the compile options of the component.

Moreover, a global target `<PKG-NAME>::<PKG-NAME>` will be declared with the following properties adjusted:

- `INTERFACE_LINK_LIBRARIES`: Containing the list of targets for all the components in the package.

Important: **Name conflicts**: If the name of the global target is the same for different packages, Conan will aggregate into this global target all the components from all those different packages. This means that this global target will contain information coming from different packages. For the components themselves, a name conflict will result in one of them being inaccessible without further notice.

18.4.6 msbuild

Introduced in Conan 1.26. This generator is aimed to supersede the existing `visualstudio` and `visualstudiomulti` generators.

> **Warning:** This generator is experimental and subject to breaking changes.

This is a generator to be used for Visual Studio projects (*.sln* solutions and *.vcxproject* files), natively, without using CMake at all. The generator will create Visual Studio properties files that can be added to the projects and solutions in the IDE, under the "properties" tab.

If a conanfile declares two requirements `"zlib/1.2.11"`, `"poco/1.9.4"`, then running the **`conan install -g=msbuild`** will create the following files:

- One properties file for each dependency and transitive dependency, like *conan_zlib.props*, *conan_openssl.props* and *conan_poco.props*. These files will transitively import other files, in this case as the `poco` package depends on `openssl`, the *conan_poco.props* will import *conan_openssl.props* file.
- One file for each dependency for each configuration, like *conan_zlib_release_x64_v141.props*, containing the corresponding variables (include folders, library folders, library name, etc.) for that configuration, like the <ConanzlibIncludeDirectories> variable. These files are conditionally included per configuration by the base dependency file (*conan_zlib.props*).
- One *conandeps.props* Visual Studio properties file, importing all the direct dependencies, in this example both *conan_zlib.props* and *conan_poco.props*.

The per-configuration files are created after installing that specific configurations.

```
$ conan install . -g msbuild -s build_type=Release -s arch=x86_64
# This will generate the conan_xxx_release_x64 properties files
$ conan install . -g msbuild -s build_type=Debug -s arch=x86
# This will generate the conan_xxx_debug_x86 properties files
```

This is a multi-configuration generator, after installing different configurations it is possible to switch the configuration directly in the Visual Studio IDE.

If a Visual Studio solutions consists of multiple subprojects, it is possible to add individual property files to specific subprojects, making it available that dependency and its transitive dependencies to that subproject only.

18.4.7 visual_studio

This is the reference page for `visual_studio` generator. Go to *Integrations/Visual Studio* if you want to learn how to integrate your project or recipes with Visual Studio.

Generates a file named *conanbuildinfo.props* containing an XML that can be imported to your Visual Studio project.

Generated xml structure:

```
<?xml version="1.0" encoding="utf-8"?>
<Project ToolsVersion="4.0" xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
  <ImportGroup Label="PropertySheets" />
  <PropertyGroup Label="UserMacros" />
  <PropertyGroup Label="Conan-RootDirs">
    <Conan-Lib1-Root>{PACKAGE LIB1 FOLDER}</Conan-Poco-Root>
    <Conan-Lib2-Root>{PACKAGE LIB2 FOLDER}</Conan-Poco-Root>
```

(continues on next page)

(continued from previous page)

```
    ...
  </PropertyGroup>
  <PropertyGroup Label="ConanVariables">
    <ConanCompilerFlags>{compiler_flags}</ConanCompilerFlags>
    <ConanLinkerFlags>{linker_flags}</ConanLinkerFlags>
    <ConanPreprocessorDefinitions>{definitions}</ConanPreprocessorDefinitions>
    <ConanIncludeDirectories>{include_dirs}</ConanIncludeDirectories>
    <ConanResourceDirectories>{res_dirs}</ConanResourceDirectories>
    <ConanLibraryDirectories>{lib_dirs}</ConanLibraryDirectories>
    <ConanBinaryDirectories>{bin_dirs}</ConanBinaryDirectories>
    <ConanLibraries>{libs}</ConanLibraries>
    <ConanSystemDeps>{system_libs}</ConanSystemDeps>
  </PropertyGroup>
  <PropertyGroup>
    <LocalDebuggerEnvironment>PATH=%PATH%;{CONAN BINARY DIRECTORIES LIST}</
↪LocalDebuggerEnvironment>
    <DebuggerFlavor>WindowsLocalDebugger</DebuggerFlavor>
  </PropertyGroup>
  <ItemDefinitionGroup>
    <ClCompile>
      <AdditionalIncludeDirectories>$(ConanIncludeDirectories)
↪%(AdditionalIncludeDirectories)</AdditionalIncludeDirectories>
      <PreprocessorDefinitions>$(ConanPreprocessorDefinitions)%(PreprocessorDefinitions)
↪</PreprocessorDefinitions>
      <AdditionalOptions>$(ConanCompilerFlags) %(AdditionalOptions)</AdditionalOptions>
    </ClCompile>
    <Link>
      <AdditionalLibraryDirectories>$(ConanLibraryDirectories)
↪%(AdditionalLibraryDirectories)</AdditionalLibraryDirectories>
      <AdditionalDependencies>$(ConanLibraries)%(AdditionalDependencies)</
↪AdditionalDependencies>
      <AdditionalDependencies>$(ConanSystemDeps)%(AdditionalDependencies)</
↪AdditionalDependencies>
      <AdditionalOptions>$(ConanLinkerFlags) %(AdditionalOptions)</AdditionalOptions>
    </Link>
    <Midl>
      <AdditionalIncludeDirectories>$(ConanIncludeDirectories)
↪%(AdditionalIncludeDirectories)</AdditionalIncludeDirectories>
    </Midl>
    <ResourceCompile>
      <AdditionalIncludeDirectories>$(ConanIncludeDirectories)
↪%(AdditionalIncludeDirectories)</AdditionalIncludeDirectories>
      <PreprocessorDefinitions>$(ConanPreprocessorDefinitions)%(PreprocessorDefinitions)
↪</PreprocessorDefinitions>
      <AdditionalOptions>$(ConanCompilerFlags) %(AdditionalOptions)</AdditionalOptions>
    </ResourceCompile>
  </ItemDefinitionGroup>
  <ItemGroup />
</Project>
```

There are `ConanVariables` containing the information of the dependencies. Those variables are used later in the file, like in the <Link> task.

Note that for single-configuration packages, which is the most typical, Conan installs Debug/Release, 32/64bits, packages separately. So a different property sheet will be generated for each configuration. The process could be:

Given for example a `conanfile.txt` like:

Listing 53: *conanfile.txt*

```
[requires]
pkg/0.1@user/channel

[generators]
visual_studio
```

And assuming that binary packages exist for `pkg/0.1@user/channel`, we could do:

```
$ mkdir debug32 && cd debug32
$ conan install .. -s compiler="Visual Studio" -s compiler.version=15 -s arch=x86 -s␣
↪build_type=Debug
$ cd ..
$ mkdir debug64 && cd debug64
$ conan install .. -s compiler="Visual Studio" -s compiler.version=15 -s arch=x86_64 -s␣
↪build_type=Debug
$ cd ..
$ mkdir release32 && cd release32
$ conan install .. -s compiler="Visual Studio" -s compiler.version=15 -s arch=x86 -s␣
↪build_type=Release
$ cd ..
$ mkdir release64 && cd release64
$ conan install .. -s compiler="Visual Studio" -s compiler.version=15 -s arch=x86_64 -s␣
↪build_type=Release
...
# Now go to VS 2017 Property Manager, load the respective sheet into each configuration
```

The above process can be simplified using profiles (assuming you have created the respective profiles), and you can also specify the generators in the command line:

```
$ conan install .. -pr=vs15release64 -g visual_studio
...
```

18.4.8 visual_studio_multi

This is the reference page for `visual_studio_multi` generator. Go to *Integrations/Visual Studio* if you want to learn how to integrate your project or recipes with Visual Studio.

Usage

```
$ conan install . -g visual_studio_multi -s arch=x86 -s build_type=Debug
$ conan install . -g visual_studio_multi -s arch=x86_64 -s build_type=Debug
$ conan install . -g visual_studio_multi -s arch=x86 -s build_type=Release
$ conan install . -g visual_studio_multi -s arch=x86_64 -s build_type=Release
```

These commands will generate 5 files for each compiler version:

- *conanbuildinfo_multi.props*: All properties
- *conanbuildinfo_release_x64_v141.props.props*: Variables for release/64bits/VS2015 (toolset v141).
- *conanbuildinfo_debug_x64_v141.props.props*: Variables for debug/64bits/VS2015 (toolset v141).
- *conanbuildinfo_release_win32_v141.props.props*: Variables for release/32bits/VS2015 (toolset v141).
- *conanbuildinfo_debug_win32_v141.props.props*: Variables for debug/32bits/VS2015 (toolset v141).

You can now load *conanbuildinfo_multi.props* in your Visual Studio IDE property manager, and all configurations will be loaded at once.

Each one of the configurations will have the format and information defined in *the visual_studio generator*.

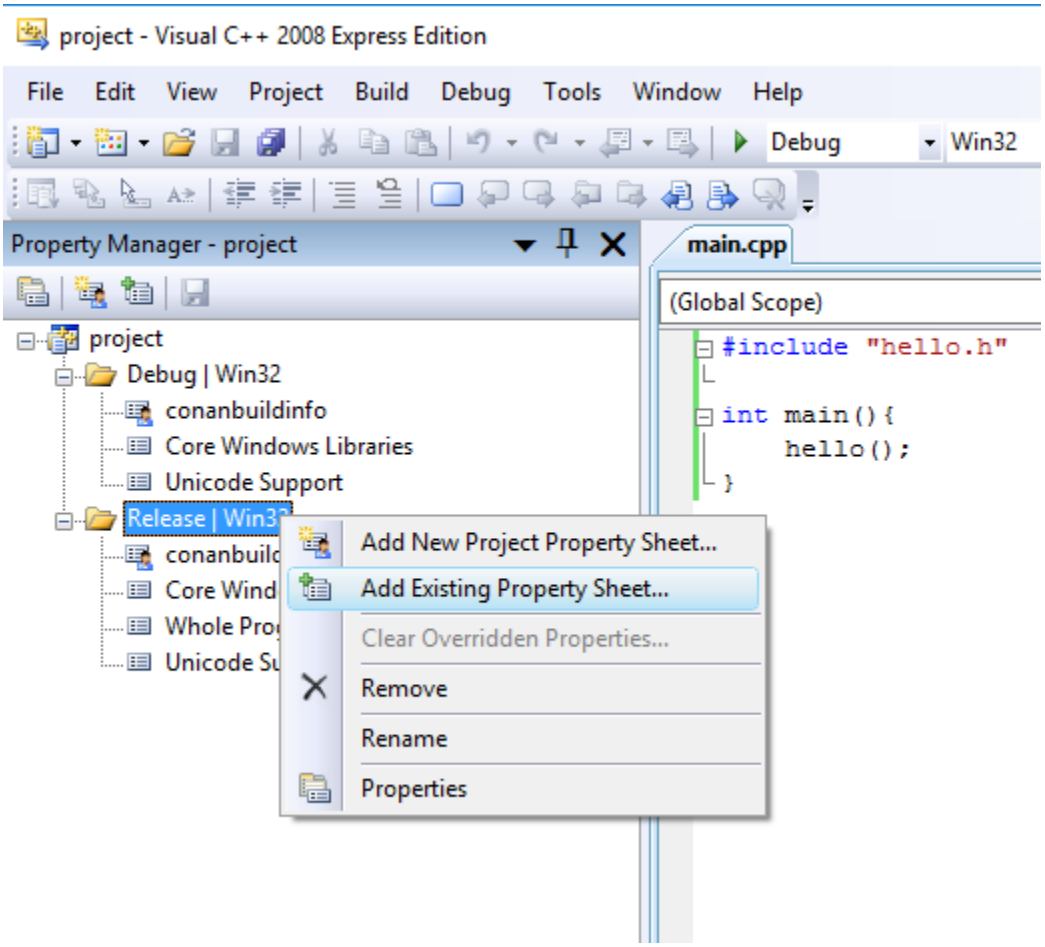
18.4.9 visual_studio_legacy

Generates a file named *conanbuildinfo.vsprops* containing an XML that can be imported to your *Visual Studio 2008* project. Note that the format of this file is different and incompatible with the *conanbuildinfo.props* file generated with the `visual_studio` generator for newer versions.

Generated XML structure:

```
<?xml version="1.0" encoding="Windows-1252"?>
<VisualStudioPropertySheet
    ProjectType="Visual C++"
    Version="8.00"
    Name="conanbuildinfo"
    >
    <Tool
        Name="VCCLCompilerTool"
        AdditionalOptions="{compiler_flags}"
        AdditionalIncludeDirectories="{include_dirs}"
        PreprocessorDefinitions="{definitions}"
    />
    <Tool
        Name="VCLinkerTool"
        AdditionalOptions="{linker_flags}"
        AdditionalDependencies="{libs}"
        AdditionalLibraryDirectories="{lib_dirs}"
    />
</VisualStudioPropertySheet>
```

This file can be loaded from the Menu->View->PropertyManager window, selecting "Add Existing Property Sheet" for the desired configuration.



Note that for single-configuration packages (which is the most typical), Conan installs Debug and Release packages separately. So a different property sheet will be generated for each configuration. The process could be:

Given for example a recipe like:

Listing 54: *conanfile.txt*

```
[requires]
pkg/0.1@user/channel

[generators]
visual_studio_legacy
```

And assuming that binary packages exist for `pkg/0.1@user/channel`, we could do:

```
$ mkdir debug && cd debug
$ conan install .. -s compiler="Visual Studio" -s compiler.version=9 -s arch=x86 -s␣
↪build_type=Debug
$ cd ..
$ mkdir release && cd release
$ conan install .. -s compiler="Visual Studio" -s compiler.version=9 -s arch=x86 -s␣
↪build_type=Release
# Now go to VS 2008 Property Manager, load the respective sheet into each configuration
```

The above process can be simplified using profiles (assuming you have created a *vs9release* profile) and you can also

specify the generators in the command line:

```
$ conan install .. -pr=vs9release -g visual_studio_legacy
```

18.4.10 xcode

This is the reference page for `xcode` generator. Go to *Integrations/Xcode* if you want to learn how to integrate your project or recipes with Xcode.

The `xcode` generator creates a file named *conanbuildinfo.xcconfig* that can be imported to your Xcode project.

The file declare these variables:

VARIABLE	VALUE
HEADER_SEARCH_PATHS	The requirements *include dirs*
LIBRARY_SEARCH_PATHS	The requirements *lib dirs*
OTHER_LDFLAGS	*-lXXX* corresponding to library and system library names
GCC_PREPROCESSOR_DEFINITION	The requirements definitions
OTHER_CFLAGS	The requirements cflags
OTHER_CPLUSPLUSFLAGS	The requirements cxxflags
FRAMEWORK_SEARCH_PATHS	The requirements framework folders, so xcode can find packaged frameworks

18.4.11 compiler_args

This is the reference page for `compiler_args` generator. Go to *Integrations/Compilers on command line* if you want to learn how to integrate your project calling your compiler in the command line.

Generates a file named *conanbuildinfo.args* containing a command line parameters to invoke `gcc`, `clang` or `cl` compiler.

You can use the `compiler_args` generator directly to build simple programs:

gcc/clang:

```
> g++ timer.cpp @conanbuildinfo.args -o bin/timer
```

cl:

```
$ cl /EHsc timer.cpp @conanbuildinfo.args
```

With gcc or clang

FLAG	MEANING
-DXXX	Corresponding to requirements *defines*
-IXXX	Corresponding to requirements *include dirs*
-Wl,-rpathXXX	Corresponding to requirements *lib dirs*
-LXXX	Corresponding to requirements *lib dirs*
-lXXX	Corresponding to requirements *libs* and *system_libs*
-m64	For x86_64 architecture
-m32	For x86 architecture
-DNDEBUG	For Release builds
-s	For Release builds (only gcc)
-g	For Debug builds
-D_GLIBCXX_USE_CXX11_ABI=0	When setting libcxx == "libstdc++"
-D_GLIBCXX_USE_CXX11_ABI=1	When setting libcxx == "libstdc++11"
-framework XXX	Corresponding to requirements *frameworks* (OSX)
-F XXX	Corresponding to requirements *framework dirs* (OSX)
Other flags	cxxflags, cflags, sharedlinkflags, exelinkflags (applied directly)

With cl (Visual Studio)

FLAG	MEANING
/DXXX	Corresponding to requirements *defines*
/IXXX	Corresponding to requirements *include dirs*
/LIBPATH:XX	Corresponding to requirements *lib dirs*
/MT, /MTd, /MD, /MDd	Corresponding to Runtime
-DNDEBUG	For Release builds
/Zi	For Debug builds

Directly inside a recipe

```
from conans import ConanFile

class PocoTimerConan(ConanFile):
    settings = "os", "compiler", "build_type", "arch"
    requires = "poco/1.9.4"
    generators = "compiler_args"
    default_options = {"poco:shared": True, "openssl:shared": True}

    def build(self):
        self.run("mkdir -p bin")
        command = 'g++ timer.cpp @conanbuildinfo.args -o bin/timer'
        self.run(command)
```

18.4.12 gcc

Deprecated, use *compiler_args* generator instead.

18.4.13 boost-build

Caution: This generator is deprecated in favor of the `b2` generator. See *generator b2*.

The `boost-build` generator creates a file named *project-root.jam* that can be used with the Boost Build build system script.

The generated *project-root.jam* file contains several sections and an alias `conan-deps` with the section names:

```
lib ssl :
    : # requirements
    <name>ssl
    <search>/path/to/package/227fb0ea22f4797212e72ba94ea89c7b3fbc2a0c/lib
    : # default-build
    : # usage-requirements
    <include>/path/to/package/227fb0ea22f4797212e72ba94ea89c7b3fbc2a0c/include
    ;

lib crypto :
    : # requirements
    <name>crypto
    <search>/path/to/package/227fb0ea22f4797212e72ba94ea89c7b3fbc2a0c/lib
    : # default-build
    : # usage-requirements
    <include>/path/to/package/227fb0ea22f4797212e72ba94ea89c7b3fbc2a0c/include
    ;

lib z :
    : # requirements
    <name>z
    <search>/path/to/package/8018a4df6e7d2b4630a814fa40c81b85b9182d2b/lib
    : # default-build
    : # usage-requirements
    <include>/path/to/package/8018a4df6e7d2b4630a814fa40c81b85b9182d2b/include
    ;

alias conan-deps :
    ssl
    crypto
    z
;
```

18.4.14 b2

This is the reference page for the `b2` (*Boost Build*) generator. It is a multi-generator to match the multi-build nature of B2.

> **Warning:** This is an **experimental** feature subject to breaking changes in future releases.

Usage

```
# Use release dependencies:
$ conan install -g b2 -s build_type=Release ...
# Optionally, also use debug dependencies:
$ conan install -g b2 -s build_type=Debug ...
# And so on for any number of configurations you need.
```

The commands will generate 3 files:

- `conanbuildinfo.jam`: Which includes the other two, and enables its use.
- `conanbuildinfo-XXX.jam`: Variables and targets adjusted only for build_type Release, where `XXX` is a key indicating the full variation built.
- `conanbuildinfo-YYY.jam`: Variables and targets adjusted only for build_type Debug, where `YYY` is a key indicating the full variation built.

Sub-projects in conanbuildinfo-XXX.jam

The `b2` generator defines sub-projects relative to the location of the B2 project you generate the Conan configuration. For each package a sub-project with the package name is created that contains targets you can use as B2 sources in your projects.

For example with this `conanfile.txt`:

```
[requires]
clara/[>=1.1.0]@bincrafters/stable
boost_predef/[>=1.66.0]@bincrafters/stable
zlib/[>=1.2.11]@conan/stable

[generators]
b2
```

You would get three sub-projects defined relative to the `conanfile.txt` location:

```
project clara ;
project boost_predef ;
project zlib ;
```

For a root level project those could be referenced with an absolute project path, for example */clara*. Or you can use relative project paths as needed, for example *../clara* or *subproject/clara*.

Targets in *conanbuildinfo-XXX.jam*

For each package a target in the corresponding package subproject is created that is specific to the variant built. There is also a general `libs` target that is an alias to all the package library targets. For header only packages this `libs` target would not contain references to the package libraries as they do not exist. But it would still contain the rest of the Usage requirements for you to make use of the headers in that package. For example, for the above *conanfile.txt*, the targets would be:

Listing 55: `clara subproject`

```
alias libs
    : # source, none as it's header only
    : # requirements specific to the build
      ...
    : # default-build
    : # usage-requirements
      <include>/absolute/path/to/conan/package/include
      <define>...
      <cflags>...
      <cxxflags>...
      <link>shared:<linkflags>...
    ;
```

Where `...` contains references to the variant specific constants. The target for `boost_predef` is equivalent as that's also a header only library. For `libz` it contains a built linkable library and hence it has additional targets for that.

Listing 56: `libz subproject`

```
alias z
    : # source, no source as it's a searched pre-built library
    : # requirements
      <name>z
      <search>/absolute/path/to/conan/package/lib
      # rest of the requirements specific to the build
    : # default-build
    : # usage-requirements
      <include>/abolute/path/to/conan/package/include
      <define>...
      <cflags>...
      <cxxflags>...
      <link>shared:<linkflags>...
    ;

alias libs
    : # source
      z
    : # requirements specific to the build
      ...
    : # default-build
    : # usage-requirements
      <include>/absolute/path/to/conan/package/include
      <define>...
      <cflags>...
      <cxxflags>...
```

(continues on next page)

(continued from previous page)

```
      <link>shared:<linkflags>...
    ;
```

Constants in *conanbuildinfo-XXX.jam*

This generator also defines constants, and path constants, in the project where the conanfile.txt is located. The constants define variant specific variables for all the packages and a transitive `conan` set of constants for all the packages.

- **Per package constants**

For each requirement `conanbuildinfo-XXX.cmake` file declares the following constants. `variation` is the name of the package and variation. That `YYY` variation takes the form of a comma separated list of: package name, address-model, architecture, target-os, toolset with version, and variant (`debug`, `release`, `relwithdebinfo`, and `minsizerel`). All are lower case and use the values of the corresponding B2 features. For example a `boost_predef` package dependency when building with apple-clang 9.0 and debug would be: `boost_predef,64,x86,darwin,clang-9.0,debug`.

NAME	VALUE
rootpath(variation)	Abs path to root package folder.
includedirs(variation)	Header's folders
libdirs(variation)	Library folders (default {rootpath}/lib)
defines(variation)	Library defines
cppflags(variation)	CXX flags
sharedlinkflags(variation)	Shared link flags
cflags(variation)	C flags
requirements(variation)	B2 requirements
usage-requirements(variation)	B2 usage requirements

Both the `requirements` and `usage-requirements` are synthesized from the other constants.

- **Global declared constants**

The generator also defines a corresponding set of constants that aggregate the values of all the package requirements. The constants for this are the same as the package-specific ones but with `conan` as the name of the project.

- **Constants from user_info**

If any of the requirements is filling the user_info object in the package_info method a set of constants will be declared following this naming:

NAME	VALUE
user(name,variation)	User declared value

`variation` is the package and variant as above and `name` the variable name in lower case. For example:

```
class MyLibConan(ConanFile):
    name = "mylib"
    version = "1.6.0"

    # ...
```

(continues on next page)

(continued from previous page)

```
    def package_info(self):
        self.user_info.var1 = 2
```

When other library requires `mylib` and uses the `b2` generator:

Listing 57: *conanbuildinfo-XXX.jam*

```
constant user(var1,mylib,...) : "2" ;
```

18.4.15 qbs

This is the reference page for `qbs` generator. Go to *Integrations/Qbs* if you want to learn how to integrate your project or recipes with Qbs.

Generates a file named *conanbuildinfo.qbs* that can be used for your Qbs builds.

A Product `ConanBasicSetup` contains the aggregated requirement values and also there is N Product declared, one per requirement.

```
import qbs 1.0

Project {
    Product {
        name: "ConanBasicSetup"
        Export {
            Depends { name: "cpp" }
            cpp.includePaths: [{INCLUDE DIRECTORIES REQUIRE 1}, {INCLUDE DIRECTORIES␣
↪REQUIRE 2}]
            cpp.libraryPaths: [{LIB DIRECTORIES REQUIRE 1}, {LIB DIRECTORIES REQUIRE 2}]
            cpp.systemIncludePaths: [{BIN DIRECTORIES REQUIRE 1}, {BIN DIRECTORIES␣
↪REQUIRE 2}]
            cpp.dynamicLibraries: [{LIB NAMES REQUIRE 1}, {LIB NAMES REQUIRE 2}]
            cpp.defines: []
            cpp.cxxFlags: []
            cpp.cFlags: []
            cpp.linkerFlags: []
        }
    }

    Product {
        name: "REQUIRE1"
        Export {
            Depends { name: "cpp" }
            cpp.includePaths: [{INCLUDE DIRECTORIES REQUIRE 1}]
            cpp.libraryPaths: [{LIB DIRECTORIES REQUIRE 1}]
            cpp.systemIncludePaths: [{BIN DIRECTORIES REQUIRE 1}]
            cpp.dynamicLibraries: ["{LIB NAMES REQUIRE 1}"]
            cpp.defines: []
            cpp.cxxFlags: []
            cpp.cFlags: []
            cpp.linkerFlags: []
        }
```

(continues on next page)

(continued from previous page)

```
    }
    // lib root path: {ROOT PATH REQUIRE 1}

    Product {
        name: "REQUIRE2"
        Export {
            Depends { name: "cpp" }
            cpp.includePaths: [{INCLUDE DIRECTORIES REQUIRE 2}]
            cpp.libraryPaths: [{LIB DIRECTORIES REQUIRE 2}]
            cpp.systemIncludePaths: [{BIN DIRECTORIES REQUIRE 2}]
            cpp.dynamicLibraries: ["{LIB NAMES REQUIRE 2}"]
            cpp.defines: []
            cpp.cxxFlags: []
            cpp.cFlags: []
            cpp.linkerFlags: []
        }
    }
    // lib root path: {ROOT PATH REQUIRE 2}

    Product {
        name: "REQUIRE3"
        Export {
            Depends { name: "cpp" }
            cpp.includePaths: [{INCLUDE DIRECTORIES REQUIRE 3}]
            cpp.libraryPaths: [{LIB DIRECTORIES REQUIRE 3}]
            cpp.systemIncludePaths: [{BIN DIRECTORIES REQUIRE 3}]
            cpp.dynamicLibraries: ["{LIB NAMES REQUIRE 3}"]
            cpp.defines: []
            cpp.cxxFlags: []
            cpp.cFlags: []
            cpp.linkerFlags: []
            Depends { name: "REQUIRE1" }
            Depends { name: "REQUIRE2" }
        }
    }
    // lib root path: {ROOT PATH REQUIRE 3}
}
```

18.4.16 qmake

This is the reference page for `qmake` generator. Go to *Integrations/Qmake* if you want to learn how to integrate your project or recipes with qmake.

Generates a file named `conanbuildinfo.pri` that can be used for your qmake builds. The file contains:

- N groups of variables, one group per require, declaring the same individual values: include_paths, libs, bin dirs, libraries, defines etc.
- One group of global variables with the aggregated values for all requirements.

Package declared vars

For each requirement `conanbuildinfo.pri` file declares the following variables. `XXX` is the name of the require in uppercase. e.k "ZLIB" for `zlib/1.2.8@lasote/stable` requirement:

NAME	VALUE
CONAN_XXX_ROOT	Abs path to root package folder.
CONAN_INCLUDEPATH_XXX	Header's folders
CONAN_LIB_DIRS_XXX	Library folders (default {CONAN_XXX_ROOT}/lib)
CONAN_BINDIRS_XXX	Binary folders (default {CONAN_XXX_ROOT}/bin)
CONAN_LIBS_XXX	Library names to link
CONAN_DEFINES_XXX	Library defines
CONAN_COMPILE_DEFINITIONS_XXX	Compile definitions
CONAN_QMAKE_CXXFLAGS_XXX	CXX flags
CONAN_QMAKE_LFLAGS_SHLIB_XXX	Linker flags (shared libs)
CONAN_QMAKE_LFLAGS_APP_XXX	Linker flags (executables)
CONAN_QMAKE_CFLAGS_XXX	C flags

Global declared vars

Conan also declares some global variables with the aggregated values of all our requirements. The values are ordered in the right order according to the dependency tree.

NAME	VALUE
CONAN_INCLUDEPATH	Aggregated header's folders
CONAN_LIB_DIRS	Aggregated library folders
CONAN_BINDIRS	Aggregated binary folders
CONAN_LIBS	Aggregated library names to link
CONAN_DEFINES	Aggregated library defines
CONAN_COMPILE_DEFINITIONS	Aggregated compile definitions
CONAN_QMAKE_CXXFLAGS	Aggregated CXX flags
CONAN_QMAKE_LFLAGS_SHLIB	Aggregated linker flags (shared libs)
CONAN_QMAKE_LFLAGS_APP	Aggregated linker flags (executables)
CONAN_QMAKE_CFLAGS	Aggregated C flags

Methods available in *conanbuildinfo.pri*

NAME	DESCRIPTION
conan_basic_setup()	Setup all the qmake vars according to our settings with the global approach

18.4.17 scons

Conan provides *integration with SCons* with this generator.

The generated `SConscript_conan` will generate several dictionaries, like:

```
"conan" : {
    "CPPPATH"     : ['/path/to/include'],
    "LIBPATH"     : ['/path/to/lib'],
    "BINPATH"     : ['/path/to/bin'],
    "LIBS"        : ['hello'],
    "CPPDEFINES"  : [],
    "CXXFLAGS"    : [],
    "CCFLAGS"     : [],
    "SHLINKFLAGS" : [],
    "LINKFLAGS"   : [],
},

"hello" : {
    "CPPPATH"     : ['/path/to/include'],
    "LIBPATH"     : ['/path/to/lib'],
    "BINPATH"     : ['/path/to/bin'],
    "LIBS"        : ['hello'],
    "CPPDEFINES"  : [],
    "CXXFLAGS"    : [],
    "CCFLAGS"     : [],
    "SHLINKFLAGS" : [],
    "LINKFLAGS"   : [],
},
```

The `conan` dictionary will contain the aggregated values for all dependencies, while the individual `"hello"` dictionaries, one per package, will contain just the values for that specific dependency.

These dictionaries can be directly loaded into the environment like:

```
conan = SConscript('{}/SConscript_conan'.format(build_path_relative_to_sconstruct))
env.MergeFlags(conan['conan'])
```

18.4.18 pkg_config

Generates pkg-config files named *<PKG-NAME>.pc* (where `<PKG-NAME` is the name declared by dependencies in `cpp_info.names["pkg_config"]` if specified), containing a valid pkg-config file syntax. The `prefix` variable is automatically adjusted to the `package_folder`.

Components

Available since: 1.28.0

If a recipe uses *components*, the files generated will be *<COMP-NAME>.pc* with their corresponding flags and require relations.

Additionally, a *<PKG-NAME>.pc* is generated to maintain compatibility for consumers with recipes that start supporting components. This *<PKG-NAME>.pc* file will declare all the components of the package as requires while the rest of the fields will be empty, relying on the propagation of flags coming from the components *<COMP-NAME>.pc* files.

Go to *Integrations/pkg-config and pc files/Use the pkg_config generator* if you want to learn how to use this generator.

Properties

The following properties affect the `pkg_config` generator:

- **pkg_config_name** property equivalent to the `names` attribute.
- **pkg_config_custom_content** property will add user defined content to the *.pc* files created by this generator.
- **component_version** property sets a custom version to be used in the `Version` field belonging to the created `*.pc` file for that component.

18.4.19 virtualenv

This is the reference page for `virtualenv` generator. Go to *Mastering/Virtual Environments* if you want to learn how to use Conan virtual environments.

Created files

- activate.{sh|bat|ps1}
- deactivate.{sh|bat|ps1}

Usage

Linux/macOS:

```
> source activate.sh
```

Windows:

```
> activate.bat
```

Variables declared

ENVIRONMENT VAR	VALUE
PS1	New shell prompt value corresponding to the current directory name
OLD_PS1	Old PS1 value, to recover it in deactivation
XXXX	Any variable declared in the `self.env_info` object of the requirements.

18.4.20 virtualenv_python

Created files

- activate_run_python.{sh|bat}
- deactivate_run_python.{sh|bat}

Usage

Linux/macOS:

```
> source activate_run_python.sh
```

Windows:

```
> activate_run_python.bat
```

Variables declared

ENVIRONMENT VAR	DESCRIPTION
PATH	With every `bin` folder of your requirements.
PYTHONPATH	Union of `PYTHONPATH` of your requirements.
LD_LIBRARY_PATH	`lib` folders of your requirements.
DYLD_LIBRARY_PATH	`lib` folders of your requirements.

18.4.21 virtualbuildenv

This is the reference page for `virtualbuildenv` generator. Go to *Mastering/Virtual Environments* if you want to learn how to use Conan virtual environments.

Created files

- *activate_build.{sh|bat}*
- *deactivate_build.{sh|bat}*

Usage

Linux/macOS:

```
$ source activate_build.sh
```

Windows:

```
$ activate_build.bat
```

Variables declared

ENVIRONMENT VAR	DESCRIPTION
LIBS	Library names to link
LDFLAGS	Link flags, (-L, -m64, -m32)
CFLAGS	Options for the C compiler (-g, -s, -m64, -m32, -fPIC)
CXXFLAGS	Options for the C++ compiler (-g, -s, -stdlib, -m64, -m32, -fPIC)
CPPFLAGS	Preprocessor definitions (-D, -I)
LIB	Library paths separated with ";" (Visual Studio)
CL	"/I" flags with include directories (Visual Studio)

In the case of using this generator to compile with Visual Studio, it also sets the environment variables needed via `tools.vcvars()` to build your project. Some of these variables are:

```
VSINSTALLDIR=C:/Program Files (x86)/Microsoft Visual Studio/2017/Community/
WINDIR=C:/WINDOWS
WindowsLibPath=C:/Program Files (x86)/Windows Kits/10/UnionMetadata/10.0.16299.0;
WindowsSdkBinPath=C:/Program Files (x86)/Windows Kits/10/bin/
WindowsSdkDir=C:/Program Files (x86)/Windows Kits/10/
WindowsSDKLibVersion=10.0.16299.0/
WindowsSdkVerBinPath=C:/Program Files (x86)/Windows Kits/10/bin/10.0.16299.0/
```

18.4.22 virtualrunenv

This is the reference page for `virtualrunenv` generator. Go to *Mastering/Virtual Environments* if you want to learn how to use Conan virtual environments.

Created files

- activate_run.{sh|bat}
- deactivate_run.{sh|bat}

Usage

Linux/macOS:

```
> source activate_run.sh
```

Windows:

```
> activate_run.bat
```

Variables declared

ENVIRONMENT VAR	DESCRIPTION
PATH	With every `bin` folder of your requirements.
LD_LIBRARY_PATH	`lib` folders of your requirements.
DYLD_LIBRARY_PATH	`lib` folders of your requirements.
DYLD_FRAMEWORK_PATH	`framework_paths` folders of your requirements.

18.4.23 youcompleteme

Go to *Integrations/YouCompleteMe* to see the details of the YouCompleteMe generator.

18.4.24 txt

This is the reference page for `txt` generator. Go to *Integrations/Custom integrations / Use the text generator* to know how to use it.

The generated `conanbuildinfo.txt` file is a generic config file with `[sections]` and values.

Package declared vars

For each requirement `conanbuildinfo.txt` file declares the following sections. `XXX` is the name of the require in lowercase. e.k "zlib" for `zlib/1.2.8@lasote/stable` requirement:

SECTION	DESCRIPTION
[includedirs_XXX]	List with the include paths of the requirement
[libdirs_XXX]	List with library paths of the requirement
[bindirs_XXX]	List with binary directories of the requirement
[resdirs_XXX]	List with the resource directories of the requirement
[builddirs_XXX]	List with the build directories of the requirement
[libs_XXX]	List with library names of the requirement
[defines_XXX]	List with the defines of the requirement
[cflags_XXX]	List with C compilation flags
[sharedlinkflags_XXX]	List with shared libraries link flags
[exelinkflags_XXX]	List with executable link flags
[cppflags_XXX]	List with C++ compilation flags
[frameworks_XXX]	List with the framework names (OSX)
[frameworkdirs_XXX]	List with the frameworks search paths (OSX).
[rootpath_XXX]	Root path of the package

Global declared vars

Conan also declares some global variables with the aggregated values of all our requirements. The values are ordered in the right order according to the dependency tree.

SECTION	DESCRIPTION
[includedirs]	List with the aggregated include paths of the requirements
[libdirs]	List with aggregated library paths of the requirements
[bindirs]	List with aggregated binary directories of the requirements
[resdirs]	List with the aggregated resource directories of the requirements
[builddirs]	List with the aggregated build directories of the requirements
[libs]	List with aggregated library names of the requirements
[system_libs]	List with aggregated system library names
[defines]	List with the aggregated defines of the requirements
[cflags]	List with aggregated C compilation flags
[sharedlinkflags]	List with aggregated shared libraries link flags
[exelinkflags]	List with aggregated executable link flags
[cppflags]	List with aggregated C++ compilation flags
[frameworks]	List with aggregated framework names (OSX)
[frameworkdirs]	List with aggregated frameworks search paths (OSX).

18.4.25 json

Warning: Actual JSON may have more fields not documented here. Those fields may change in the future without previous warning.

A file named *conanbuildinfo.json* will be generated. It will contain the information about every dependency and the installed settings and options:

```
{
  "deps_env_info": {
```

(continues on next page)

(continued from previous page)

```
    "MY_ENV_VAR": "foo"
  },
  "deps_user_info": {
    "hello": {
      "my_var": "my_value"
    }
  },
  "dependencies":
  [
    {
      "name": "fmt",
      "version": "4.1.0",
      "include_paths": [
        "/path/to/.conan/data/fmt/4.1.0/<user>/<channel>/package/<id>/include"
      ],
      "lib_paths": [
        "/path/to/.conan/data/fmt/4.1.0/<user>/<channel>/package/<id>/lib"
      ],
      "libs": [
        "fmt"
      ],
      "...": "...",
    },
    {
      "name": "poco",
      "version": "1.9.4",
      "...": "..."
    }
  ],
  "settings": {
    "os": "Linux",
    "arch": "armv7"
  },
  "options": {
    "curl": {
      "shared": true,
    }
  }
}
```

The generated *conanbuildinfo.json* file is a JSON file with the following keys:

dependencies

The dependencies is a list, with each item belonging to one dependency, and each one with the following keys:

- `name`
- `version`
- `description`
- `rootpath`
- `sysroot`

- include_paths, lib_paths, bin_paths, build_paths, res_paths, framework_paths
- libs, frameworks, system_libs
- defines, cflags, cppflags, sharedlinkflags, exelinkflags
- build_modules, build_modules_paths
- configs (only for multi config dependencies, see below)

Please note that the dependencies are ordered, it isn't a map, order is relevant. Upstream dependencies, i.e. the ones that do not depend on other packages, will be first, and their direct dependencies after them, and so on.

The node configs will appear only for *multi config recipes*, it is holding a dictionary with the data related to each configuration:

```
{
"...": "...",
"dependencies": [
    {
        "name": "hello",
        "rootpath": "/private/var/folders/yq/14hmvxm96xd7gfgl37_tnrbh0000gn/T/tmpkp9l_
↪dovconans/path with spaces/.conan/data/hello/0.1/lasote/testing/package/
↪46f53f156846659bf39ad6675fa0ee8156e859fe",
        "...": "...",
        "configs": {
            "debug": {
                "libs": ["hello_d"]
            },
            "release": {
                "libs": ["hello"]
            }
        }
    },
    {
        "...": "..."
    }
    ]
}
```

deps_env_info

The environment variables defined by upstream dependencies.

deps_user_info

The user variables defined by upstream dependencies.

settings

The settings used during `conan install`.

options

The options of each dependency.

18.4.26 premake

Warning: This is an **experimental** feature subject to breaking changes in future releases.

This is the reference page for `premake` generator. Go to *Integrations/premake* if you want to learn how to integrate your project or recipes with Premake.

Generates a file name *conanbuildinfo.premake.lua* that can be used for your Premake builds (both Premake 4 and 5 are supported).

The file contains:

- N groups of variables, one group per require, declaring the same individual values: include dirs, libs, bin dirs, defines, etc.
- One group of global variables with aggregated values for all requirements.
- Helper functions to setup the settings in your configuration.

Variables

Package declared variables

For each requirement *conanbuildinfo.premake.lua* file declares the following variables. `XXX` is the name of the require. e.g. "zlib" for `zlib/1.2.11@lasote/stable` requirement:

NAME	VALUE
conan_includedirs_XXX	Headers's folders (default {CONAN_XXX_ROOT}/include)
conan_libdirs_XXX	Library folders (default {CONAN_XXX_ROOT}/lib)
conan_bindirs_XXX	Binary folders (default {CONAN_XXX_ROOT}/bin)
conan_libs_XXX	Library names to link
conan_defines_XXX	Compile definitions
conan_cxxflags_XXX	CXX flags
conan_cflags_XXX	C flags
conan_sharedlinkflags_XXX	Shared link flags
conan_exelinkflags_XXX	Executable link flags
conan_rootpath_XXX	Abs path to root package folder
conan_frameworks_XXX	Declared cpp_info.frameworks

Global declared variables

NAME	VALUE
conan_includedirs	Aggregated headers's folders
conan_libdirs	Aggregated library folders
conan_bindirs	Aggregated binary folders
conan_libs	Aggregated library names to link
conan_defines	Aggregated compile definitions
conan_cxxflags	Aggregated CXX flags
conan_cflags	Aggregated C flags
conan_sharedlinkflags	Aggregated shared link flags
conan_exelinkflags	Aggregated executable link flags
conan_frameworks	Aggregated frameworks from cpp_info.frameworks

Note: Both the global `conan_frameworks` and each `conan_frameworks_xxx` support only system frameworks, not frameworks packaged by the requirements. See discussion here.

Functions

conan_basic_setup()

Basic function to setup the settings into your configuration. Useful to reduce the logic in Premake scripts and automate the conversion of settings:

```
function conan_basic_setup()
    configurations{conan_build_type}
    architecture(conan_arch)
    includedirs{conan_includedirs}
    libdirs{conan_libdirs}
    links{conan_libs}
    links{conan_frameworks}
    defines{conan_cppdefines}
    bindirs{conan_bindirs}
end
```

18.4.27 make

This is the reference page for `make` generator. Go to *Integrations/make* if you want to learn how to integrate your project or recipes with make.

This generators creates a file named *conanbuildinfo.mak* with information of dependencies in different variables that can be used for your make builds.

Variables

Variables per package. The <PKG-NAME> placeholder is filled with the name of the Conan package.

NAME	VALUE
CONAN_ROOT_<PKG-NAME>	Absolute path to root package folder
CONAN_SYSROOT_<PKG-NAME>	System root folder
CONAN_INCLUDE_DIRS_<PKG-NAME>	Headers folders
CONAN_LIB_DIRS_<PKG-NAME>	Library folders
CONAN_BIN_DIRS_<PKG-NAME>	Binary folders
CONAN_BUILD_DIRS_<PKG-NAME>	Build folders
CONAN_RES_DIRS_<PKG-NAME>	Resources folders
CONAN_LIBS_<PKG-NAME>	Library names to link with
CONAN_SYSTEM_LIBS_<PKG-NAME>	System library names to link with
CONAN_DEFINES_<PKG-NAME>	Library definitions
CONAN_CFLAGS_<PKG-NAME>	Options for the C compiler (-g, -s, -m64, -m32, -fPIC)
CONAN_CXXFLAGS_<PKG-NAME>	Options for the C++ compiler (-g, -s, -stdlib, -m64, -m32, -fPIC, -std)
CONAN_SHAREDLINKFLAGS_<PKG-NAME>	Library Shared linker flags
CONAN_EXELINK_FLAGS_<PKG-NAME>	Executable linker flags
CONAN_FRAMEWORKS_<PKG-NAME>	Frameworks (OSX)
CONAN_FRAMEWORK_PATHS_<PKG-NAME>	Framework folders (OSX) (default {CONAN_XXX_ROOT}/Frameworks

Conan also declares some **global variables** with the aggregated values of all our requirements. The values are ordered in the right order according to the dependency tree.

NAME	VALUE
CONAN_ROOTPATH	Aggregated root folders
CONAN_SYSROOT	Aggregated system root folders
CONAN_INCLUDE_DIRS	Aggregated header folders
CONAN_LIB_DIRS	Aggregated library folders
CONAN_BIN_DIRS	Aggregated binary folders
CONAN_BUILD_DIRS	Aggregated build folders
CONAN_RES_DIRS	Aggregated resource folders
CONAN_LIBS	Aggregated library names to link with
CONAN_SYSTEM_LIBS	Aggregated system library names to link with
CONAN_DEFINES	Aggregated library definitions
CONAN_CFLAGS	Aggregated options for the C compiler
CONAN_CXXFLAGS	Aggregated options for the C++ compiler
CONAN_SHAREDLINKFLAGS	Aggregated Shared linker flags
CONAN_EXELINKFLAGS	Aggregated Executable linker flags
CONAN_FRAMEWORKS	Aggregated frameworks (OSX)
CONAN_FRAMEWORK_PATHS	Aggregated framework folders (OSX)

Important: Note that the mapping of the Conan variables to the Make ones is done taking the following rules and we suggest to use the variables indicated under the *Makefile* column to apply to a common naming:

cpp_info	*conanbuildinfo.mak*	*Makefile*
defines	CONAN_DEFINES	CPPFLAGS
includedirs	CONAN_INCLUDE_DIRS	CPPFLAGS
libdirs	CONAN_LIB_DIRS	LDFLAGS
libs	CONAN_LIBS	LDLIBS
SYSTEM_LIBS	CONAN_SYSTEM_LIBS	
cflags	CONAN_CFLAGS	CFLAGS
cxxflags	CONAN_CXXFLAGS	CXXFLAGS

18.4.28 markdown

This generator creates a *.md* file for each requirement with useful information to consume the installed packages: libraries available, components, headers, and basic instructions to consume them using different build systems.

```
$ conan install poco/1.10.1@ --generator markdown
...
Generator markdown created poco.md
```

Although markdown files can be read in plain text, we highly recommend you to use any plugin to see it with proper rendering (browsers, IDEs,.. all of them have plugins that will render markdown documents).

poco/1.10.1

poco/1.10.1 dependencies

- pcre/8.45
- zlib/1.2.11
- expat/2.4.1
- sqlite3/3.36.0
- openssl/1.1.1l
- libpq/13.4
- apr/1.7.0
- apr-util/1.6.1
- libmysqlclient/8.0.25
- bzip2/1.0.8
- libiconv/1.16
- zstd/1.5.0
- lz4/1.9.3

Using the poco Conan Package

Conan integrates with different build systems. You can declare which build system you want your project to use setting in the **[generators]** section of the conanfile.txt or using the **generators** attribute in the conanfile.py. Here, there is some basic information you can use to integrate **poco** in your own project. For more detailed information, please check the Conan documentation.

Using poco with CMake

Conan CMake generators

- CMakeDeps: generates information about where the **poco** library and its dependencies (pcre, zlib, expat, sqlite3, openssl, libpq, apr, apr-util, libmysqlclient, bzip2, libiconv, zstd, lz4) are installed together with other information like version, flags, and directory data or configuration. CMake will use this files when you invoke `find_package()` in your *CMakeLists.txt*.
- CMakeToolchain: generates a CMake toolchain file the you can later invoke with CMake in the command line using `-DCMAKE_TOOLCHAIN_FILE=conantoolchain.cmake` .

Declare these generators in your **conanfile.txt** along with your **poco** dependency like:

```
[requires]
poco/1.10.1

[generators]
CMakeDeps
CMakeToolchain
```

To use **poco** in a simple CMake project with this structure:

```
.
├── CMakeLists.txt
├── conanfile.txt
└── src
    └── main.cpp
```

18.4.29 deploy

The deploy generator makes a bulk copy of the packages folders of all dependencies in a graph. It can be used to deploy binaries from the local cache to the user space:

```
$ conan install openssl/1.0.2u@ -g deploy
...
Installing package: openssl/1.0.2u
...
Generator deploy created deploy_manifest.txt
```

Files from dependencies are deployed under a folder with the name of the dependency.

```
$ ls -R
openssl/  conanbuildinfo.txt  deploy_manifest.txt  zlib/

./openssl:
LICENSE  include/  lib/
```

(continues on next page)

(continued from previous page)

```
./openssl/include:
openssl/

./openssl/include/openssl:
aes.h       blowfish.h  cms.h       des_old.h  ebcdic.h  evp.h       md4.h      ocsp.h  ␣
↪      pkcs12.h  ripemd.h     srtp.h   symhacks.h   whrlpool.h
applink.c   bn.h        comp.h      dh.h       ec.h      hmac.h      md5.h      ␣
↪opensslconf.h  pkcs7.h   rsa.h        ssl.h    tls1.h       x509.h
asn1.h      buffer.h    conf.h      dsa.h      ecdh.h    idea.h      mdc2.h     openssl.
↪h     pqueue.h  safestack.h  ssl2.h   ts.h         x509_vfy.h
asn1_mac.h  camellia.h  conf_api.h  dso.h      ecdsa.h   krb5_asn.h  modes.h    ossl_typ.
↪h     rand.h    seed.h       ssl23.h  txt_db.h     x509v3.h
asn1t.h     cast.h      crypto.h    dtls1.h    engine.h  kssl.h      obj_mac.h  pem.h   ␣
↪      rc2.h     sha.h        ssl3.h   ui.h
bio.h       cmac.h      des.h       e_os2.h    err.h     lhash.h     objects.h  pem2.h  ␣
↪      rc4.h     srp.h        stack.h  ui_compat.h

./openssl/lib:
libeay32.lib  ssleay32.lib

./zlib:
FindZLIB.cmake  include/  lib/  licenses/  zlib.pc

./zlib/include:
zconf.h  zlib.h

./zlib/lib:
pkgconfig/  zlib.lib

./zlib/lib/pkgconfig:
zlib.pc

./zlib/licenses:
LICENSE
```

The generated *deploy_manifest.txt* file is a manifest file with a list of all the files deployed and hash of the contents for each of them.

If any symbolic is present in the package folder, it will be preserved as well, and not copied as a new file or folder.

Tip: You can use the parameter **`--install-folder`** in the **`conan install`** to output the contents of the packages to a specific folder.

See also:

For a different approach to deploy package files in the user space folders, check the *deploy()* method.

Important: If none of these generators fit your needs, you can create your own custom_generator.

18.5 Profiles

Profiles allows users to set a complete configuration set for **settings**, **options**, **environment variables**, and **build requirements** in a file. They have this structure:

```
[settings]
setting=value

[options]
MyLib:shared=True

[env]
env_var=value

[tool_requires]
tool1/0.1@user/channel
tool2/0.1@user/channel, tool3/0.1@user/channel
*: tool4/0.1@user/channel
```

Profile can be created with `new` option in **`conan profile`**. And then edit it later.

```
$ conan profile new mynewprofile --detect
```

Profile files can be used with `-pr`/`--profile` option in many commands like **`conan install`** or **`conan create`** commands.

```
$ conan create . demo/testing -pr=myprofile
```

Profiles can be located in different folders. For example, the default *<userhome>/.conan/profiles*, and be referenced by absolute or relative path:

```
$ conan install . --profile /abs/path/to/profile   # abs path
$ conan install . --profile ./relpath/to/profile   # resolved to current dir
$ conan install . --profile ../relpath/to/profile  # resolved to relative dir
$ conan install . --profile profile  # resolved to user/.conan/profiles/profile
```

Listing existing profiles in the *profiles* folder can be done like this:

```
$ conan profile list
default
myprofile1
myprofile2
...
```

You can also show profile's content:

```
$ conan profile show myprofile1
Configuration for profile myprofile1:

[settings]
os=Windows
arch=x86_64
compiler=Visual Studio
compiler.version=15
```

(continues on next page)

(continued from previous page)

```
build_type=Release
[options]
[tool_requires]
[env]
```

Use $PROFILE_DIR in your profile and it will be replaced with the absolute path to the directory where the profile file is (this path will contain only forward slashes). It is useful to declare relative folders:

```
[env]
PATH=$PROFILE_DIR/dev_tools
```

Tip: You can manage your profiles and share them using *conan config install*.

18.5.1 Package settings and env vars

Profiles also support **package settings** and **package environment variables** definition, so you can override some settings or environment variables for some specific package:

Listing 58: *.conan/profiles/zlib_with_clang*

```
[settings]
zlib:compiler=clang
zlib:compiler.version=3.5
zlib:compiler.libcxx=libstdc++11
compiler=gcc
compiler.version=4.9
compiler.libcxx=libstdc++11

[env]
zlib:CC=/usr/bin/clang
zlib:CXX=/usr/bin/clang++
```

Your build tool will locate **clang** compiler only for the **zlib** package and **gcc** (default one) for the rest of your dependency tree.

They accept patterns too, like `-s *@myuser/*`, which means that packages that have the username "myuser" will use clang 3.5 as compiler, and gcc otherwise:

```
[settings]
*@myuser/*:compiler=clang
*@myuser/*:compiler.version=3.5
*@myuser/*:compiler.libcxx=libstdc++11
compiler=gcc
compiler.version=4.9
compiler.libcxx=libstdc++11
```

Also, as a **experimental** feature, *&* can be specified as the package name. It will apply only to the consumer conanfile (.py or .txt). This is a special case because the consumer conanfile might not declare a *name* so it would be impossible to reference it.

```
[settings]
&:compiler=gcc
&:compiler.version=4.9
&:compiler.libcxx=libstdc++11
```

Note: If you want to override existing system environment variables, you should use the `key=value` syntax. If you need to pre-pend to the system environment variables you should use the syntax `key=[value]` or `key=[value1, value2, ...]`. A typical example is the `PATH` environment variable, when you want to add paths to the existing system PATH, not override it, you would use:

```
[env]
PATH=[/some/path/to/my/tool]
```

18.5.2 Tools configurations

Warning: This is an **experimental** feature subject to breaking changes in future releases.

Tools configurations can also be used in profile files and *global.conf* one. Profile values will have priority over globally defined ones in *global.conf*, and can be defined as:

```
[settings]
...

[conf]
tools.microsoft.msbuild:verbosity=Diagnostic
tools.microsoft.msbuild:max_cpu_count=2
tools.microsoft.msbuild:vs_version = 16
tools.build:jobs=10
```

See also:

You can see more information about configurations in *global.conf section*.

18.5.3 Profile composition

You can specify multiple profiles in the command line. The applied configuration will be the composition of all the profiles applied in the order they are specified.

If, for example, you want to apply a *tool require*, like a `cmake` installer to your dependency tree, it won't be very practical adding the *cmake* installer reference, e.g `cmake/3.16.3` to all your profiles where you could need to inject `cmake` as a tool require.

You can specify both profiles instead:

Listing 59: *.conan/profiles/cmake_316*

```
[tool_requires]
cmake/3.16.3
```

```
$ conan install . --profile clang --profile cmake_316
```

18.5.4 Profile includes

You can include other profiles using the `include()` statement. The path can be relative to the current profile, absolute, or a profile name from the default profile location in the local cache.

The `include()` statement has to be at the top of the profile file:

Listing 60: *gcc_49*

```
[settings]
compiler=gcc
compiler.version=4.9
compiler.libcxx=libstdc++11
```

Listing 61: *myprofile*

```
include(gcc_49)

[settings]
zlib:compiler=clang
zlib:compiler.version=3.5
zlib:compiler.libcxx=libstdc++11

[env]
zlib:CC=/usr/bin/clang
zlib:CXX=/usr/bin/clang++
```

18.5.5 Variable declaration

In a profile you can declare variables that will be replaced automatically by Conan before the profile is applied. The variables have to be declared at the top of the file, after the `include()` statements.

Listing 62: *myprofile*

```
include(gcc_49)
CLANG=/usr/bin/clang

[settings]
zlib:compiler=clang
zlib:compiler.version=3.5
zlib:compiler.libcxx=libstdc++11

[env]
```

(continues on next page)

(continued from previous page)

```
zlib:CC=$CLANG/clang
zlib:CXX=$CLANG/clang++
```

The variables will be inherited too, so you can declare variables in a profile and then include the profile in a different one, all the variables will be available:

Listing 63: *gcc_49*

```
GCC_PATH=/my/custom/toolchain/path/

[settings]
compiler=gcc
compiler.version=4.9
compiler.libcxx=libstdc++11
```

Listing 64: *myprofile*

```
include(gcc_49)

[settings]
zlib:compiler=clang
zlib:compiler.version=3.5
zlib:compiler.libcxx=libstdc++11

[env]
zlib:CC=$GCC_PATH/gcc
zlib:CXX=$GCC_PATH/g++
```

18.5.6 Build profiles and host profiles

Warning: This is an **experimental feature** subject to breaking changes in future releases.

All the commands that take a profile as an argument, from Conan v1.24 are starting to accept two profiles with command line arguments `-pr:h`/`--profile:host` and `-pr:b`/`--profile:build`. If both profiles are provided, Conan will build a graph with some packages associated with the `host` platform and some build requirements associated to the `build` platform. There are two scenarios where this feature is extremly useful:

- *Creating conan packages to install dev tools*
- *Cross building*

The default build profile in Conan 1.X is not defined by default, and needs to be specified in command line. However, it is also possible to define a default one in `global.conf` configuration file with:

Listing 65: *global.conf*

```
core:default_build_profile=default
core:default_profile=linux_armv8
```

The default host profile can be defaulted as well using this configuration method.

18.5.7 Profile templates

Warning: This is an **experimental** feature subject to breaking changes in future releases.

From Conan 1.38 it is possible to use **jinja2** template engine for profiles. This feature is enabled by naming the profile file with the `.jinja` extension. When Conan loads a profile with this extension, immediately parses and renders the template, which must result in a standard text profile.

Some of the capabilities of the profile templates are:

- Using the platform information, like obtaining the current OS is possible because the Python `platform` module is added to the render context.:

```
[settings]
os = {{ {"Darwin": "Macos"}.get(platform.system(), platform.system()) }}
```

- Reading environment variables can be done because the Python `os` module is added to the render context.:

```
[settings]
build_type = {{ os.getenv("MY_BUILD_TYPE") }}
```

- Defining your own variables and using them in the profile:

```
{% set a = "FreeBSD" %}
[settings]
os = {{ a }}
```

- Joining and defining paths, including referencing the current profile directory. For example, defining a toolchain which file is located besides the profile can be done. Besides the `os` Python module, the variable `profile_dir` pointing to the current profile folder is added to the context.

```
[conf]
tools.cmake.cmaketoolchain:toolchain_file = {{ os.path.join(profile_dir, "toolchain.
↪cmake") }}
```

- Including or importing other files from `profiles` folder:

Listing 66: profile_vars.jinja

```
{% set a = "Debug" %}
```

Listing 67: profile1.jinja

```
{% import "profile_vars.jinja" as vars %}
[settings]
build_type = {{ vars.a }}
```

- Any other feature supported by *jinja2* is possible: for loops, if-else, etc. This would be useful to define custom per-package settings or options for multiple packages in a large dependency graph.

18.5.8 Examples

If you are working with Linux and you usually work with **gcc** compiler, but you have installed **clang** compiler and want to install some package for `clang` compiler, you could do:

- Create a `.conan/profiles/clang` file:

```
[settings]
compiler=clang
compiler.version=3.5
compiler.libcxx=libstdc++11

[env]
CC=/usr/bin/clang
CXX=/usr/bin/clang++
```

- Execute an install command passing the **--profile** or **-pr** parameter:

```
$ conan install . --profile clang
```

Without profiles you would have needed to set CC and CXX variables in the environment to point to your clang compiler and use **-s** parameters to specify the settings:

```
$ export CC=/usr/bin/clang
$ export CXX=/usr/bin/clang++
$ conan install -s compiler=clang -s compiler.version=3.5 -s compiler.libcxx=libstdc++11
```

A profile can also be used in **conan create** and **conan info**:

```
$ conan create . demo/testing --profile clang
```

See also:

- Check the section *Tool requirements* to read more about its usage in a profile.
- Check *conan profile* and *profiles/default* for full reference.
- Related section: *Cross building*.

18.6 Build helpers

Build helpers are Python wrappers of a build tool that help with the conversion of the Conan settings to the build system's ones. They assist users with the compilation of libraries and applications in the `build()` method of a recipe.

Contents:

18.6.1 CMake

The *CMake* class helps us to invoke *cmake* command with the generator, flags and definitions, reflecting the specified Conan settings.

There are two ways to invoke your cmake tools:

- Using the helper attributes `cmake.command_line` and `cmake.build_config`:

```
from conans import ConanFile, CMake

 class ExampleConan(ConanFile):
     ...

     def build(self):
         cmake = CMake(self)
         self.run('cmake "%s" %s' % (self.source_folder, cmake.command_line))
         self.run('cmake --build . %s' % cmake.build_config)
         self.run('cmake --build . --target install')
```

- Using the helper methods:

```
from conans import ConanFile, CMake

class ExampleConan(ConanFile):
    ...

    def build(self):
        cmake = CMake(self)
        # same as cmake.configure(source_folder=self.source_folder, build_folder=self.
↪build_folder)
        cmake.configure()
        cmake.build()
        cmake.test() # Build the "RUN_TESTS" or "test" target
        # Build the "install" target, defining CMAKE_INSTALL_PREFIX to self.package_
↪folder
        cmake.install()
```

Constructor

```
class CMake(object):

    def __init__(self, conanfile, generator=None, cmake_system_name=True,
                 parallel=True, build_type=None, toolset=None, make_program=None,
                 set_cmake_flags=False, msbuild_verbosity='minimal', cmake_program=None,
                 generator_platform=None, append_vcvars=False)
```

Parameters:

- **conanfile** (Required): Conanfile object. Usually `self` in a *conanfile.py*
- **generator** (Optional, Defaulted to `None`): Specify a custom generator instead of autodetect it. e.g., "MinGW Makefiles"
- **cmake_system_name** (Optional, Defaulted to `True`): Specify a custom value for `CMAKE_SYSTEM_NAME` instead of autodetect it.
- **parallel** (Optional, Defaulted to `True`): If `True`, will append the *-jN* attribute for parallel building being N the *cpu_count()*. Also applies to parallel test execution (by defining `CTEST_PARALLEL_LEVEL` environment variable).
- **build_type** (Optional, Defaulted to `None`): Force the build type instead of taking the value from the settings. Note that `CMAKE_BUILD_TYPE` will not be declared when using CMake multi-configuration generators such as Visual Studio or XCode as it will not have effect.
- **toolset** (Optional, Defaulted to `None`): Specify a toolset for Visual Studio.
- **make_program** (Optional, Defaulted to `None`): Indicate path to `make`.
- **set_cmake_flags** (Optional, Defaulted to `None`): Whether or not to set CMake flags like `CMAKE_CXX_FLAGS`, `CMAKE_C_FLAGS`, etc.
- **msbuild_verbosity** (Optional, Defaulted to `minimal`): verbosity level for MSBuild (in case of Visual Studio generator). Set this parameter to `None` to avoid using it in the command line.
- **cmake_program** (Optional, Defaulted to `None`): Path to the custom cmake executable.
- **generator_platform** (Optional, Defaulted to `None`): Generator platform name or none to autodetect (-A cmake option).
- **append_vcvars** (Optional, Defaulted to `False`): When a Visual Studio environment is activated by the build helper, append it to respect existing environment. `CMake` helper sometimes, like when using the Ninja generator, needs to call `vcvars` to set the VS environment. By default the `vcvars` is pre-pended to the environment, taking precedence. With `append_vcvars=True`, the `vcvars` will append to the end of the environment (for "list" environment variables, like `PATH`), instead of pre-pending, so the existing environment takes precedence.

Attributes

generator

Specifies a custom CMake generator to use, see also cmake-generators documentation.

generator_platform

Specifies a custom CMake generator platform to use, see also CMAKE_GENERATOR_PLATFORM documentation.

verbose

Defaulted to: `False`

Set it to `True` or `False` to automatically set the definition `CMAKE_VERBOSE_MAKEFILE`.

```
from conans import ConanFile, CMake

class ExampleConan(ConanFile):
    ...

    def build(self):
        cmake = CMake(self)
        cmake.verbose = True
        cmake.configure()
        cmake.build()
```

build_folder (Read only)

Build folder where the `configure()` and `build()` methods will be called.

build_type [Deprecated]

Build type can be forced with this variable instead of taking it from the settings.

flags (Read only)

Flag conversion of `definitions` to be used in the command line invocation (`-D`).

is_multi_configuration (Read only)

Indicates whether the generator selected allows builds with multi configuration: Release, Debug... Multi configuration generators are Visual Studio and Xcode ones.

command_line (Read only)

Arguments and flags calculated by the build helper that will be applied. It indicates the generator, the Conan definitions and the flags converted from the specified Conan settings. For example:

```
-G "Unix Makefiles" -DCMAKE_BUILD_TYPE=Release ... -DCONAN_C_FLAGS=-m64 -Wno-dev
```

build_config (Read only)

Value for **--config** option for Multi-configuration IDEs. This flag will only be set if the generator `is_multi_configuration` and `build_type` was not forced in constructor class.

An example of the value of this property could be:

```
--config Release
```

parallel

Defaulted to: `True`

Run CMake process in parallel for compilation, installation and testing. This is translated into the proper command line argument: For `Unix Makefiles` it is `-jX` and for `Visual Studio` it is `/m:X`.

However, the parallel executing can be changed for testing like this:

```
cmake = CMake(self)
cmake.configure()
cmake.build()  # 'parallel' is enabled by default
cmake.parallel = False
cmake.test()
```

In the case of `cmake.test()` this flag sets the `CTEST_PARALLEL_LEVEL` variable to the according value in *tools.cpu_count()*.

definitions

The CMake helper will automatically append some definitions based on your settings:

Variable	Description
ANDROID_ABI	Just alias for CMAKE_ANDROID_ARCH_ABI
ANDROID_NDK	Defined when one of ANDROID_NDK_ROOT or ANDROID_NDK_HOME environment variables presented
BUILD_SHARED_LIBS	Only if your recipe has a `shared` option
CMAKE_ANDROID_ARCH_ABI	Set to a suitable value if cross-building to an Android is detected
CMAKE_BUILD_TYPE	Debug, Release... from `self.settings.build_type` or `build_type` attribute **only** if `is_multi_configuration`
CMAKE_EXPORT_NO_PACKAGE_R	Defined by default to disable the package registry
CMAKE_MODULE_PATH	Set to `conanfile.install_folder` when using `cmake_find_package` or `cmake_find_package_multi`
CMAKE_OSX_ARCHITECTURES	`i386` if architecture is x86 in an OSX system
CMAKE_PREFIX_PATH	Set to `conanfile.install_folder` when using `cmake_find_package_multi`
CMAKE_SYSTEM_NAME	Set to `self.settings.os` value if cross-building is detected
CMAKE_SYSROOT	Defined if CONAN_CMAKE_SYSROOT is defined as environment variable
CMAKE_SYSTEM_VERSION	Set to `self.settings.os.version` value if cross-building is detected
CONAN_CMAKE_CXX_EXTENSIOI	Set to `ON` or `OFF` value when GNU extensions for the given C++ standard are enabled
CONAN_CMAKE_CXX_STANDARI	Set to the `self.settings.compiler.cppstd` value (or `self.settings.cppstd` for backward compatibility)
CONAN_CMAKE_FIND_ROOT_PAT	Definition set only if same environment variable is declared by user
CONAN_CMAKE_FIND_ROOT_PAT	Definition set only if same environment variable is declared by user
CONAN_CMAKE_FIND_ROOT_PAT	Definition set only if same environment variable is declared by user
CONAN_CMAKE_FIND_ROOT_PAT	Definition set only if same environment variable is declared by user
CONAN_CMAKE_POSITION_INDEI	Set when `fPIC` option exists and `True` or `fPIC` exists and `False` but `shared` option exists and `True`
CONAN_CMAKE_SYSTEM_PROCES	Definition set only if same environment variable is declared by user
CONAN_COMPILER	Conan internal variable to check the compiler
CONAN_CXX_FLAGS	Set to `-m32` or `-m64` values based on the architecture and `/MP` for MSVS
CONAN_C_FLAGS	Set to `-m32` or `-m64` values based on the architecture and `/MP` for MSVS
CONAN_EXPORTED	Defined when CMake is called using Conan CMake helper
CONAN_IN_LOCAL_CACHE	`ON` if the build runs in local cache, `OFF` if running in a user folder
CONAN_LIBCXX	Set to `self.settings.compiler.libcxx` value
CONAN_LINK_RUNTIME	Set to the runtime value from `self.settings.compiler.runtime` for MSVS
CONAN_SHARED_LINKER_FLAGS	Set to `-m32` or `-m64` values based on the architecture
CONAN_STD_CXX_FLAG	Set to the flag corresponding to the C++ standard defined in `self.settings.compiler.cppstd`. Used for CMake < 3.1)

There are some definitions set to be used later on the the `install()` step too:

Variable	Description
CMAKE_INSTALL_BINDIR	Set to *bin* inside the package folder.
CMAKE_INSTALL_DATAROOTDIR	Set to *share* inside the package folder.
CMAKE_INSTALL_INCLUDEDIR	Set to *include* inside the package folder.
CMAKE_INSTALL_LIBDIR	Set to *lib* inside the package folder.
CMAKE_INSTALL_LIBEXECDIR	Set to *bin* inside the package folder.
CMAKE_INSTALL_OLDINCLUDEDIR	Set to *include* inside the package folder.
CMAKE_INSTALL_PREFIX	Set to `conanfile.package_folder` value.
CMAKE_INSTALL_SBINDIR	Set to *bin* inside the package folder.

But you can change the automatic definitions after the `CMake()` object creation using the `definitions` property or even add your own ones:

```
from conans import ConanFile, CMake

class ExampleConan(ConanFile):
    ...

    def build(self):
        cmake = CMake(self)
        cmake.definitions["CMAKE_SYSTEM_NAME"] = "Generic"
        cmake.definitions["MY_CUSTOM_DEFINITION"] = "OFF"
        cmake.configure()
        cmake.build()
        cmake.install()  # Build --target=install
```

Note that definitions changed **after** the `configure()` call will **not** take effect later on the `build()`, `test()` or `install()` ones.

Methods

configure()

```
def configure(self, args=None, defs=None, source_dir=None, build_dir=None,
              source_folder=None, build_folder=None, cache_build_folder=None,
              pkg_config_paths=None)
```

Configures *CMake* project with the given parameters.

Parameters:

- **args** (Optional, Defaulted to `None`): A list of additional arguments to be passed to the `cmake` command. Each argument will be escaped according to the current shell. No extra arguments will be added if `args=None`
- **defs** (Optional, Defaulted to `None`): A dict that will be converted to a list of CMake command line variable definitions of the form `-DKEY=VALUE`. Each value will be escaped according to the current shell and can be either `str`, `bool` or of numeric type
- **source_dir** (Optional, Defaulted to `None`): **[DEPRECATED]** Use `source_folder` instead. CMake's source directory where *CMakeLists.txt* is located. The default value is the `build` folder if `None` is specified (or the `source` folder if `no_copy_source` is specified). Relative paths are allowed and will be relative to `build_folder`.

- **build_dir** (Optional, Defaulted to `None`): **[DEPRECATED]** Use `build_folder` instead. CMake's output directory. The default value is the package `build` root folder if `None` is specified. The `CMake` object will store `build_folder` internally for subsequent calls to `build()`.
- **source_folder**: CMake's source directory where `CMakeLists.txt` is located. The default value is the `self.source_folder`. Relative paths are allowed and will be relative to `self.source_folder`.
- **build_folder**: CMake's output directory. The default value is the `self.build_folder` if `None` is specified. The `CMake` object will store `build_folder` internally for subsequent calls to `build()`.
- **cache_build_folder** (Optional, Defaulted to `None`): Use the given subfolder as build folder when building the package in the local cache. This argument doesn't have effect when the package is being built in user folder with **conan build** but overrides **build_folder** when working in the local cache. See *self.in_local_cache*.
- **pkg_config_paths** (Optional, Defaulted to `None`): Specify folders (in a list) of relative paths to the install folder or absolute ones where to find `*.pc` files (by using the env var `PKG_CONFIG_PATH`). If `None` is specified but the conanfile is using the `pkg_config` generator, the `self.install_folder` will be added to the `PKG_CONFIG_PATH` in order to locate the pc files of the requirements of the conanfile.

build()

```
def build(self, args=None, build_dir=None, target=None)
```

Builds *CMake* project with the given parameters.

Parameters:

- **args** (Optional, Defaulted to `None`): A list of additional arguments to be passed to the `cmake` command. Each argument will be escaped according to the current shell. No extra arguments will be added if `args=None`
- **build_dir** (Optional, Defaulted to `None`): CMake's output directory. If `None` is specified the `build_dir` from `configure()` will be used.
- **target** (Optional, Defaulted to `None`): Specifies the target to execute. The default *all* target will be built if `None` is specified. `"install"` can be used to relocate files to aid packaging.

test()

```
def test(args=None, build_dir=None, target=None, output_on_failure=False)
```

Build *CMake* test target (could be RUN_TESTS in multi-config projects or `test` in single-config projects), which usually means building and running unit tests. When this function is called *CONAN_RUN_TESTS* will be evaluated to check if tests should run.

Parameters:

- **args** (Optional, Defaulted to `None`): A list of additional arguments to be passed to the `cmake` command. Each argument will be escaped according to the current shell. No extra arguments will be added if `args=None`.
- **build_dir** (Optional, Defaulted to `None`): CMake's output directory. If `None` is specified the `build_folder` from `configure()` will be used.
- **target** (Optional, default to `None`). Alternative target name for running the tests. If not defined RUN_TESTS or `test` will be used.

- **output_on_failure** (Optional, default to `False`). Enables ctest to show output of failed tests by defining `CTEST_OUTPUT_ON_FAILURE` environment variable (same effect as `ctest --output-on-failure`).

This method can be globally skipped by `tools.build:skip_test` [conf], or `CONAN_RUN_TESTS` environment variable.

install()

```
def install(args=None, build_dir=None)
```

Installs *CMake* project with the given parameters.

Parameters:

- **args** (Optional, Defaulted to `None`): A list of additional arguments to be passed to the `cmake` command. Each argument will be escaped according to the current shell. No extra arguments will be added if `args=None`.
- **build_dir** (Optional, Defaulted to `None`): CMake's output directory. If `None` is specified the `build_folder` from `configure()` will be used.

patch_config_paths() [EXPERIMENTAL]

```
def patch_config_paths()
```

> **Warning:** This is an **experimental** feature subject to breaking changes in future releases.

This method changes references to the absolute path of the installed package in exported CMake config files to the appropriate Conan variable. Method also changes references to other packages installation paths in export CMake config files to Conan variable with their installation roots. This makes most CMake config files portable.

For example, if a package foo installs a file called *fooConfig.cmake* to be used by cmake's `find_package()` method, normally this file will contain absolute paths to the installed package folder, for example it will contain a line such as:

```
SET(Foo_INSTALL_DIR /home/developer/.conan/data/foo/1.0.0/...)
```

This will cause cmake's `find_package()` method to fail when someone else installs the package via Conan. This function will replace such paths to:

```
SET(Foo_INSTALL_DIR ${CONAN_FOO_ROOT})
```

Which is a variable that is set by *conanbuildinfo.cmake*, so that `find_package()` now correctly works on this Conan package.

For dependent packages method replaces lines with references to dependencies installation paths such as:

```
SET_TARGET_PROPERTIES(foo PROPERTIES INTERFACE_INCLUDE_DIRECTORIES "/home/developer/.
↪conan/data/bar/1.0.0/user/channel/id/include")
```

to following lines:

```
SET_TARGET_PROPERTIES(foo PROPERTIES INTERFACE_INCLUDE_DIRECTORIES "${CONAN_BAR_ROOT}/
↪include")
```

If the `install()` method of the CMake object in the conanfile is used, this function should be called **after** that invocation. For example:

```
def build(self):
    cmake = CMake(self)
    cmake.configure()
    cmake.build()
    cmake.install()
    cmake.patch_config_paths()
```

get_version()

```
@staticmethod
def get_version()
```

Returns the CMake version in a `conans.model.Version` object as it is evaluated by the command line. Will raise if cannot resolve it to valid version.

Environment variables

There are some environment variables that will also affect the `CMake()` helper class. Check them in the *CMAKE RELATED VARIABLES* section.

Example

The following example of `conanfile.py` shows you how to manage a project with conan and CMake.

```
from conans import ConanFile, CMake

class SomePackage(ConanFile):
    name = "somepkg"
    version = "1.0.0"
    settings = "os", "compiler", "build_type", "arch"
    generators = "cmake"

    def configure_cmake(self):
        cmake = CMake(self)

        # put definitions here so that they are re-used in cmake between
        # build() and package()
        cmake.definitions["SOME_DEFINITION_NAME"] = "On"

        cmake.configure()
        return cmake

    def build(self):
        cmake = self.configure_cmake()
        cmake.build()

        # run unit tests after the build
```

(continues on next page)

(continued from previous page)

```
        cmake.test()

        # run custom make command
        self.run("make -j3 check)

    def package(self):
        cmake = self.configure_cmake()
        cmake.install()
```

Default used generators

When a compiler or its version is not detected, the CMake helper uses a default generator based on the platform operating system. For Unix systems it generates `Unix Makefiles`. For Windows there is no default generator, it will be detected by CMake automatically.

18.6.2 AutoToolsBuildEnvironment (configure/make)

If you are using **configure/make** you can use **AutoToolsBuildEnvironment** helper. This helper sets `LIBS`, `LDFLAGS`, `CFLAGS`, `CXXFLAGS` and `CPPFLAGS` environment variables based on your requirements.

```
from conans import ConanFile, AutoToolsBuildEnvironment

class ExampleConan(ConanFile):
   settings = "os", "compiler", "build_type", "arch"
   requires = "poco/1.9.4"
   default_options = {"poco:shared": True, "openssl:shared": True}

   def imports(self):
      self.copy("*.dll", dst="bin", src="bin")
      self.copy("*.dylib*", dst="bin", src="lib")

   def build(self):
      autotools = AutoToolsBuildEnvironment(self)
      autotools.configure()
      autotools.make()
```

It also works using the *environment_append* context manager applied to your **configure and make** commands, calling *configure* and *make* manually:

```
from conans import ConanFile, AutoToolsBuildEnvironment, tools

class ExampleConan(ConanFile):
    ...

    def build(self):
        env_build = AutoToolsBuildEnvironment(self)
        with tools.environment_append(env_build.vars):
            self.run("./configure")
            self.run("make")
```

You can change some variables like `fpic`, `libs`, `include_paths` and `defines` before accessing the `vars` to override an automatic value or add new values:

```
from conans import ConanFile, AutoToolsBuildEnvironment

class ExampleConan(ConanFile):
    ...

    def build(self):
        env_build = AutoToolsBuildEnvironment(self)
        env_build.fpic = True
        env_build.libs.append("pthread")
        env_build.defines.append("NEW_DEFINE=23")
        env_build.configure()
        env_build.make()
```

You can use it also with `MSYS2/MinGW` subsystems installed by setting the *win_bash* parameter in the constructor. It will run the the `configure` and `make` commands inside a `bash` that has to be in the path or declared in `CONAN_BASH_PATH`:

```
from conans import ConanFile, AutoToolsBuildEnvironment, tools

class ExampleConan(ConanFile):
   settings = "os", "compiler", "build_type", "arch"

   def imports(self):
     self.copy("*.dll", dst="bin", src="bin")
     self.copy("*.dylib*", dst="bin", src="lib")

   def build(self):
      env_build = AutoToolsBuildEnvironment(self, win_bash=tools.os_info.is_windows)
      env_build.configure()
      env_build.make()
```

Constructor

```
class AutoToolsBuildEnvironment(object):

    def __init__(self, conanfile, win_bash=False)
```

Parameters:

- **conanfile** (Required): Conanfile object. Usually `self` in a conanfile.py
- **win_bash**: (Optional, Defaulted to `False`): When True, it will run the configure/make commands inside a bash.

Attributes

You can adjust the automatically filled values modifying the attributes like this:

```
from conans import ConanFile, AutoToolsBuildEnvironment

class ExampleConan(ConanFile):
    ...

    def build(self):
        autotools = AutoToolsBuildEnvironment(self)
        autotools.fpic = True
        autotools.libs.append("pthread")
        autotools.defines.append("NEW_DEFINE=23")
        autotools.configure()
        autotools.make()
```

fpic

Defaulted to: `True` if `fPIC` option exists and `True` or when `fPIC` exists and `False` but option `shared` exists and `True`. Otherwise `None`.

Set it to `True` if you want to append the `-fPIC` flag.

libs

List with library names of the requirements (`-l` in `LIBS`).

include_paths

List with the include paths of the requires (-I in CPPFLAGS).

library_paths

List with library paths of the requirements (-L in LDFLAGS).

defines

List with variables that will be defined with `-D` in `CPPFLAGS`.

flags

List with compilation flags (`CFLAGS` and `CXXFLAGS`).

cxx_flags

List with only C++ compilation flags (`CXXFLAGS`).

link_flags

List with linker flags

Properties

vars

Environment variables `CPPFLAGS`, `CXXFLAGS`, `CFLAGS`, `LDFLAGS`, `LIBS` generated by the build helper to use them in the configure, make and install steps. This variables are generated dynamically with the values of the attributes and can also be modified to be used in the following configure, make or install steps:

```
def build():
    autotools = AutoToolsBuildEnvironment()
    autotools.fpic = True
    env_build_vars = autotools.vars
    env_build_vars['RCFLAGS'] = '-O COFF'
    autotools.configure(vars=env_build_vars)
    autotools.make(vars=env_build_vars)
    autotools.install(vars=env_build_vars)
```

vars_dict

Same behavior as `vars` but this property returns each variable `CPPFLAGS`, `CXXFLAGS`, `CFLAGS`, `LDFLAGS`, `LIBS` as dictionaries.

Methods

configure()

```
def configure(self, configure_dir=None, args=None, build=None, host=None, target=None,
              pkg_config_paths=None, vars=None)
```

Configures *Autotools* project with the given parameters.

Important: This method sets by default the `--prefix` argument to `self.package_folder` whenever `--prefix` is not provided in the `args` parameter during the configure step.

There are other flags set automatically to fix the install directories by default:

- `--bindir`, `--sbindir` and `--libexecdir` set to *bin* folder.
- `--libdir` set to *lib* folder.
- `--includedir`, `--oldincludedir` set to *include* folder.
- `--datarootdir` set to *share* folder.

These flags will be set on demand, so only the available options in the *./configure* are actually set. They can also be totally skipped using `use_default_install_dirs=False` as described in the section below.

Warning: Since Conan 1.8 this build helper sets the output library directory via `--libdir` automatically to `${prefix}/lib`. This means that if you are using the `install()` method to package with AutoTools, library artifacts will be stored in the `lib` directory unless indicated explicitly by the user.

This change was introduced in order to fix issues detected in some Linux distributions where libraries were being installed to the `lib64` folder (instead of `lib`) when rebuilding a package from sources. In those cases, if `package_info()` was declaring `self.cpp_info.libdirs` as `lib`, the consumption of the package was broken.

This was considered a bug in the build helper, as it should be as much deterministic as possible when building the same package for the same settings and generally for any other user input.

If you were already modeling the `lib64` folder in your recipe, make sure you use `lib` for `self.cpp_info.libdirs` or inject the argument in the Autotools' `configure()` method:

```
atools = AutoToolsBuildEnvironment()
atools.configure(args=["--libdir=${prefix}/lib64"])
atools.install()
```

You can also skip its default value using the parameter `use_default_install_dirs=False`.

Parameters:

- **configure_dir** (Optional, Defaulted to `None`): Directory where the `configure` script is. If `None`, it will use the current directory.
- **args** (Optional, Defaulted to `None`): A list of additional arguments to be passed to the `configure` script. Each argument will be escaped according to the current shell. `--prefix` and `--libdir`, will be adjusted automatically if not indicated specifically.
- **build** (Optional, Defaulted to `None`): To specify a value for the parameter `--build`. If `None` it will try to detect the value if cross-building is detected according to the settings. If `False`, it will not use this argument at all.
- **host** (Optional, Defaulted to `None`): To specify a value for the parameter `--host`. If `None` it will try to detect the value if cross-building is detected according to the settings. If `False`, it will not use this argument at all.
- **target** (Optional, Defaulted to `None`): To specify a value for the parameter `--target`. If `None` it will try to detect the value if cross-building is detected according to the settings. If `False`, it will not use this argument at all.
- **pkg_config_paths** (Optional, Defaulted to `None`): Specify folders (in a list) of relative paths to the install folder or absolute ones where to find `*.pc` files (by using the env var `PKG_CONFIG_PATH`). If `None` is specified but the conanfile is using the `pkg_config` generator, the `self.install_folder` will be added to the `PKG_CONFIG_PATH` in order to locate the pc files of the requirements of the conanfile.
- **vars** (Optional, Defaulted to `None`): Overrides custom environment variables in the configure step.

- **use_default_install_dirs** (Optional, Defaulted to `True`): Use or not the defaulted installation dirs such as `--libdir`, `--bindir`...

make()

```
def make(self, args="", make_program=None, target=None, vars=None)
```

Builds *Autotools* project with the given parameters.

Parameters:

- **args** (Optional, Defaulted to ""): A list of additional arguments to be passed to the `make` command. Each argument will be escaped accordingly to the current shell. No extra arguments will be added if `args=""`.
- **make_program** (Optional, Defaulted to `None`): Allows to specify a different `make` executable, e.g., `mingw32-make`. The environment variable *CONAN_MAKE_PROGRAM* can be used too.
- **target** (Optional, Defaulted to `None`): Choose which target to build. This allows building of e.g., docs, shared libraries or install for some AutoTools projects.
- **vars** (Optional, Defaulted to `None`): Overrides custom environment variables in the make step.

install()

```
def install(self, args="", make_program=None, vars=None)
```

Performs the install step of autotools calling `make(target="install")`.

Parameters:

- **args** (Optional, Defaulted to ""): A list of additional arguments to be passed to the `make` command. Each argument will be escaped accordingly to the current shell. No extra arguments will be added if `args=""`.
- **make_program** (Optional, Defaulted to `None`): Allows to specify a different `make` executable, e.g., `mingw32-make`. The environment variable *CONAN_MAKE_PROGRAM* can be used too.
- **vars** (Optional, Defaulted to `None`): Overrides custom environment variables in the install step.

Environment variables

The following environment variables will also affect the *AutoToolsBuildEnvironment* helper class.

NAME	DESCRIPTION
LIBS	Library names to link
LDFLAGS	Link flags, (-L, -m64, -m32)
CFLAGS	Options for the C compiler (-g, -s, -m64, -m32, -fPIC)
CXXFLAGS	Options for the C++ compiler (-g, -s, -stdlib, -m64, -m32, -fPIC, -std)
CPPFLAGS	Preprocessor definitions (-D, -I)

See also:

- *Reference/Tools/environment_append*

18.6.3 MSBuild

Calls Visual Studio **MSBuild** command to build a *.sln* project:

```
from conans import ConanFile, MSBuild

class ExampleConan(ConanFile):
    ...

    def build(self):
        msbuild = MSBuild(self)
        msbuild.build("MyProject.sln")
```

Internally the `MSBuild` build helper uses *VisualStudioBuildEnvironment* to adjust the `LIB` and `CL` environment variables with all the information from the requirements: include directories, library names, flags etc. and then calls **MSBuild**.

- *VisualStudioBuildEnvironment* to adjust the `LIB` and `CL` environment variables with all the information from the requirements: include directories, library names, flags etc.
- *tools.msvc_build_command() [DEPRECATED]* to call :command:`MSBuild`.

You can adjust all the information from the requirements accessing to the `build_env` that it is a *VisualStudioBuildEnvironment* object:

```
from conans import ConanFile, MSBuild

class ExampleConan(ConanFile):
    ...

    def build(self):
        msbuild = MSBuild(self)
        msbuild.build_env.include_paths.append("mycustom/directory/to/headers")
        msbuild.build_env.lib_paths.append("mycustom/directory/to/libs")
        msbuild.build_env.link_flags = []

        msbuild.build("MyProject.sln")
```

To inject the flags corresponding to the `compiler.runtime`, `build_type` and `compiler.cppstd` settings, this build helper also generates a properties file (in the build folder) that is passed to :command:`MSBuild` with :command:`/p:ForceImportBeforeCppTargets="conan_build.props"`.

Constructor

```
class MSBuild(object):

    def __init__(self, conanfile)
```

Parameters:

- **conanfile** (Required): ConanFile object. Usually `self` in a *conanfile.py*.

Attributes

build_env

A *VisualStudioBuildEnvironment* object with the needed environment variables.

Methods

build()

```
def build(self, project_file, targets=None, upgrade_project=True, build_type=None,␣
↪arch=None,
          parallel=True, force_vcvars=False, toolset=None, platforms=None, use_env=True,
          vcvars_ver=None, winsdk_version=None, properties=None, output_binary_log=None,
          property_file_name=None, verbosity=None, definitions=None,
          user_property_file_name=None)
```

Builds Visual Studio project with the given parameters.

Parameters:

- **project_file** (Required): Path to the *.sln* file.
- **targets** (Optional, Defaulted to `None`): Sets `/target` flag to the specified list of targets to build.
- **upgrade_project** (Optional, Defaulted to `True`): Will call **`devenv /upgrade`** to upgrade the solution to your current Visual Studio.
- **build_type** (Optional, Defaulted to `None`): Sets `/p:Configuration` flag to the specified value. It will override the value from `settings.build_type`.
- **arch** (Optional, Defaulted to `None`): Sets `/p:Platform` flag to the specified value. It will override the value from `settings.arch`. This value (or the `settings.arch` one if not overridden) will be used as the key for the `msvc_arch` dictionary that returns the final string used for the `/p:Platform` flag (see **platforms** argument documentation below).
- **parallel** (Optional, Defaulted to `True`): Will use the configured number of cores in the *conan.conf* file or *tools.cpu_count()*:
  - **In the solution**: Building the solution with the projects in parallel. (`/m:` parameter).
  - **CL compiler**: Building the sources in parallel. (`/MP:` compiler flag).
- **force_vcvars** (Optional, Defaulted to `False`): Will ignore if the environment is already set for a different Visual Studio version.
- **toolset** (Optional, Defaulted to `None`): Sets `/p:PlatformToolset` to the specified toolset. When `None` it will apply the setting `compiler.toolset` if specified. When `False` it will skip adjusting the `/p:PlatformToolset`.
- **platforms** (Optional, Defaulted to `None`): This dictionary will update the default one (see `msvc_arch` below) and will be used to get the mapping of architectures to platforms from the Conan naming to another one. It is useful for Visual Studio solutions that have a different naming in architectures. Example: `platforms={"x86":"Win32"}` (Visual solution uses "Win32" instead of "x86").

```
msvc_arch = {'x86': 'x86',
             'x86_64': 'x64',
```

(continues on next page)

(continued from previous page)

```
                'armv7': 'ARM',
                'armv8': 'ARM64'}
```

- **use_env** (Optional, Defaulted to `True`: Sets `/p:UseEnv=true` flag. Note that this setting does not guarantee that environment variables from Conan will not be used by the compiler or linker. This is an MSBuild setting which simply specifies the behavior when environment variables conflict with equivalent properties from the project (via *.vcxproj*, *.props* or *.targets* files). Conan will still apply the relevant compiler and linker environment variables when spawning the MSBuild process. For example, if `use_env=False` is specified **and** if there is no `AdditionalDependencies` variable defined in the project, the `LINK` environment variable passed by Conan will still be used by the linker because it technically doesn't conflict with the project variable.
- **vcvars_ver** (Optional, Defaulted to `None`): Specifies the Visual Studio compiler toolset to use.
- **winsdk_version** (Optional, Defaulted to `None`): Specifies the version of the Windows SDK to use.
- **properties** (Optional, Defaulted to `None`): Dictionary with new properties, for each element in the dictionary `{name: value}` it will append a `/p:name="value"` option.
- **output_binary_log** (Optional, Defaulted to `None`): Sets `/bl` flag. If set to `True` then MSBuild will output a binary log file called *msbuild.binlog* in the working directory. It can also be used to set the name of log file like this `output_binary_log="my_log.binlog"`. This parameter is only supported starting from MSBuild version 15.3 and onwards.
- **property_file_name** (Optional, Defaulted to `None`): Sets `p:ForceImportBeforeCppTargets`. When `None` it will generate a file named *conan_build.props*. You can specify a different name for the generated properties file.
- **verbosity** (Optional, Defaulted to `None`): Sets the `/verbosity` flag to the specified verbosity level. Possible values are `"quiet"`, `"minimal"`, `"normal"`, `"detailed"` and `"diagnostic"`.
- **definitions** (Optional, Defaulted to `None`): Dictionary with additional compiler definitions to be applied during the build. Use a dictionary with the desired key and its value set to `None` to set a compiler definition with no value.
- **user_property_file_name** (Optional, Defaulted to `None`): Filename or list of filenames of user properties files to be automatically passed to the build command. These files have priority over the *conan_build.props* file (user can override that file values), and if a list of file names is provided, later file names also have priority over the former ones. These filenames will be passed, together with *conan_build.props* files as `/p:ForceImportBeforeCppTargets` argument.

Note: The `MSBuild()` build helper will, before calling to **`MSBuild`**, call *tools.vcvars_command()* to adjust the environment according to the settings. When cross-building from x64 to x86 the toolchain by default is `x86`. If you want to use `amd64_x86` instead, set the environment variable `PreferredToolArchitecture=x64`.

---

get_command()

Returns a string command calling **MSBuild**.

```
def get_command(self, project_file, props_file_path=None, targets=None, upgrade_
↪project=True,
                build_type=None, arch=None, parallel=True, toolset=None, platforms=None,
                use_env=False, properties=None, output_binary_log=None, verbosity=None,
                user_property_file_name=None)
```

Parameters:

- **props_file_path** (Optional, Defaulted to `None`): Path to a property file to be included in the compilation command. This parameter is automatically set by the `build()` method to set the runtime from settings.
- Same parameters as the `build()` method.

get_version()

Static method that returns the version of MSBuild for the specified settings.

```
def get_version(settings)
```

Result is returned in a `conans.model.Version` object as it is evaluated by the command line. It will raise an exception if it cannot resolve it to a valid result.

Parameters:

- **settings** (Required): Conanfile settings. Use `self.settings`.

18.6.4 VisualStudioBuildEnvironment

Prepares the needed environment variables to invoke the Visual Studio compiler. Use it together with *tools.vcvars_command()*.

```
from conans import ConanFile, VisualStudioBuildEnvironment

class ExampleConan(ConanFile):

    ...

    def build(self):
        if self.settings.compiler == "Visual Studio":
            env_build = VisualStudioBuildEnvironment(self)
            with tools.environment_append(env_build.vars):
                vcvars = tools.vcvars_command(self.settings)
                self.run('%s && cl /c /EHsc hello.cpp' % vcvars)
                self.run('%s && lib hello.obj -OUT:hello.lib' % vcvars
```

You can adjust the automatically filled attributes:

```
def build(self):
    if self.settings.compiler == "Visual Studio":
    env_build = VisualStudioBuildEnvironment(self)
```

(continues on next page)

(continued from previous page)

```
    env_build.include_paths.append("mycustom/directory/to/headers")
    env_build.lib_paths.append("mycustom/directory/to/libs")
    env_build.link_flags = []
    with tools.environment_append(env_build.vars):
        vcvars = tools.vcvars_command(self.settings)
        self.run('%s && cl /c /EHsc hello.cpp' % vcvars)
        self.run('%s && lib hello.obj -OUT:hello.lib' % vcvars
```

Constructor

```
class VisualStudioBuildEnvironment(object):

    def __init__(self, conanfile, with_build_type_flags=True)
```

Parameters:

- **conanfile** (Required): ConanFile object. Usually `self` in a *conanfile.py*.
- **with_build_type_flags** (Optional, Defaulted to `True`): If `True`, it adjusts the compiler flags according to the `build_type` setting. e.g: *-Zi*, *-Ob0*, *-Od*…

Environment variables

NAME	DESCRIPTION
LIB	Library paths separated with ";"
CL	"/I" flags with include directories, Runtime (/MT, /MD…), Definitions (/DXXX), and any other C and CXX flags.

Attributes

include_paths

List with directories of include paths.

lib_paths

List with directories of libraries.

defines

List with definitions from requirements' `cpp_info.defines`.

runtime

List with directories from `settings.compiler.runtime`.

flags

List with flags from requirements' `cpp_info.cflags`.

cxx_flags

List with cxx flags from requirements' `cpp_info.cxxflags`.

link_flags

List with linker flags from requirements' `cpp_info.sharedlinkflags` and `cpp_info.exelinkflags`

std

This property contains the flag corresponding to the C++ standard. If you are still using the deprecated setting `cppstd` (see *How to manage C++ standard [EXPERIMENTAL]*) and you are not providing any value for this setting, the property will be `None`.

parallel

Defaulted to `False`.

Sets the flag `/MP` in order to compile the sources in parallel using cores found by *tools.cpu_count()*.

See also:

Read more about *tools.environment_append()*.

18.6.5 Meson

If you are using **Meson Build** as your build system, you can use the **Meson** build helper. Specially useful with the *pkg_config* that will generate the *.pc* files of our requirements, then `Meson()` build helper will locate them automatically.

```
from conans import ConanFile, tools, Meson
import os

class ConanFileToolsTest(ConanFile):
    generators = "pkg_config"
    requires = "lib_a/0.1@conan/stable"
    settings = "os", "compiler", "build_type"
```

(continues on next page)

(continued from previous page)

```
    def build(self):
        meson = Meson(self)
        meson.configure(build_folder="build")
        meson.build()
```

Constructor

```
class Meson(object):

    def __init__(self, conanfile, backend=None, build_type=None)
```

Parameters:

- **conanfile** (Required): Use `self` inside a `conanfile.py`.
- **backend** (Optional, Defaulted to `None`): Specify a backend to be used, otherwise it will use `"Ninja"`.
- **build_type** (Optional, Defaulted to `None`): Force to use a build type, ignoring the value from the settings.

Methods

configure()

```
def configure(self, args=None, defs=None, source_folder=None, build_folder=None,
              pkg_config_paths=None, cache_build_folder=None, append_vcvars=False)
```

Configures Meson project with the given parameters.

Parameters:

- **args** (Optional, Defaulted to `None`): A list of additional arguments to be passed to the `configure` script. Each argument will be escaped according to the current shell. No extra arguments will be added if `args=None`.
- **defs** (Optional, Defaulted to `None`): A list of definitions.
- **source_folder** (Optional, Defaulted to `None`): Meson's source directory where `meson.build` is located. The default value is the `self.source_folder`. Relative paths are allowed and will be relative to `self.source_folder`.
- **build_folder** (Optional, Defaulted to `None`): Meson's output directory. The default value is the `self.build_folder` if `None` is specified. The `Meson` object will store `build_folder` internally for subsequent calls to `build()`.
- **pkg_config_paths** (Optional, Defaulted to `None`): A list containing paths to locate the pkg-config files (*.*pc*). If `None`, it will be set to `conanfile.build_folder`.
- **cache_build_folder** (Optional, Defaulted to `None`): Subfolder to be used as build folder when building the package in the local cache. This argument doesn't have effect when the package is being built in user folder with **`conan build`** but overrides **build_folder** when working in the local cache. See *self.in_local_cache*.
- **append_vcvars** (Optional, Defaulted to `False`): When a Visual Studio environment is activated by the build helper, append it to respect existing environment. `Meson` helper uses the Ninja generator and needs to call `vcvars` to set the VS environment. By default the `vcvars` is pre-pended to the environment, taking

precedence. With `append_vcvars=True`, the `vcvars` will append to the end of the environment (for "list" environment variables, like `PATH`), instead of pre-pending, so the existing environment takes precedence.

build()

```
def build(self, args=None, build_dir=None, targets=None)
```

Builds Meson project with the given parameters.

Parameters:

- **args** (Optional, Defaulted to `None`): A list of additional arguments to be passed to the `ninja` command. Each argument will be escaped according to the current shell. No extra arguments will be added if `args=None`.
- **build_dir** (Optional, Defaulted to `None`): Build folder. If `None` is specified the `build_folder` from `configure()` will be used. If `build_folder` from `configure()` is `None`, it will be set to `conanfile.build_folder`.
- **targets** (Optional, Defaulted to `None`): Specifies the targets to build. The default *all* target will be built if `None` is specified.

test()

```
def test(args=None, build_dir=None, target=None)
```

Executes ninja test target, which usually means building and running unit tests. When this function is called *CONAN_RUN_TESTS* will be evaluated to check if tests should run.

Parameters:

- **args** (Optional, Defaulted to `None`): A list of additional arguments to be passed to the `ninja` command. Each argument will be escaped according to the current shell. No extra arguments will be added if `args=None`.
- **build_dir** (Optional, Defaulted to `None`): Build folder. If `None` is specified the `build_folder` from `configure()` will be used. If `build_folder` from `configure()` is `None`, it will be set to `conanfile.build_folder`.
- **targets** (Optional, Defaulted to `None`): Specifies the targets to be executed. The *test* target will be executed if `None` is specified.

This method can be globally skipped by `tools.build:skip_test` [conf], or `CONAN_RUN_TESTS` environment variable.

install()

```
def install(args=None, build_dir=None)
```

Executes ninja install target.

Parameters:

- **args** (Optional, Defaulted to `None`): A list of additional arguments to be passed to the `ninja` command. Each argument will be escaped according to the current shell. No extra arguments will be added if `args=None`.

- **build_dir** (Optional, Defaulted to None): Build folder. If None is specified the build_folder from configure() will be used. If build_folder from configure() is None, it will be set to conanfile.build_folder.

meson_test()

```
def meson_test(args=None, build_dir=None)
```

Executes meson test command.

Parameters:

- **args** (Optional, Defaulted to None): A list of additional arguments to be passed to the meson test command. Each argument will be escaped according to the current shell. No extra arguments will be added if args=None.
- **build_dir** (Optional, Defaulted to None): Build folder. If None is specified the build_folder from configure() will be used. If build_folder from configure() is None, it will be set to conanfile.build_folder.

meson_install()

```
def meson_install(args=None, build_dir=None)
```

Executes meson install command.

Parameters:

- **args** (Optional, Defaulted to None): A list of additional arguments to be passed to the meson install command. Each argument will be escaped according to the current shell. No extra arguments will be added if args=None.
- **build_dir** (Optional, Defaulted to None): Build folder. If None is specified the build_folder from configure() will be used. If build_folder from configure() is None, it will be set to conanfile.build_folder.

Example

A typical usage of the Meson build helper, if you want to be able to both execute **conan create** and also build your package for a library locally (in your user folder, not in the local cache), could be:

```
from conans import ConanFile, Meson

class HelloConan(ConanFile):
    name = "hello"
    version = "0.1"
    settings = "os", "compiler", "build_type", "arch"
    generators = "pkg_config"
    exports_sources = "src/*"
    requires = "zlib/1.2.11"

    def build(self):
        meson = Meson(self)
```

(continues on next page)

(continued from previous page)

```
        meson.configure(source_folder="%s/src" % self.source_folder,
                        build_folder="build")
        meson.build()

    def package(self):
        self.copy("*.h", dst="include", src="src")
        self.copy("*.lib", dst="lib", keep_path=False)
        self.copy("*.dll", dst="bin", keep_path=False)
        self.copy("*.dylib*", dst="lib", keep_path=False)
        self.copy("*.so", dst="lib", keep_path=False)
        self.copy("*.a", dst="lib", keep_path=False)

    def package_info(self):
        self.cpp_info.libs = ["hello"]
```

Note the `pkg_config` generator, which generates *.pc* files (*zlib.pc* from the example above), which are understood by Meson to process dependencies information (no need for a `meson` generator).

The layout is:

```
<folder>
  | - conanfile.py
  | - src
      | - meson.build
      | - hello.cpp
      | - hello.h
```

And the *meson.build* could be as simple as:

```
project('hello',
        'cpp',
        version : '0.1.0'
        default_options : ['cpp_std=c++11']
        )

library('hello',
        ['hello.cpp'],
        dependencies: [dependency('zlib')]
        )
```

This allows, to create the package with **`conan create`** as well as to build the package locally:

```
$ cd <folder>
$ conan create . user/testing
# Now local build
$ mkdir build && cd build
$ conan install ..
$ conan build ..
```

18.6.6 RunEnvironment

The `RunEnvironment` helper prepares `PATH`, `LD_LIBRARY_PATH`, `DYLD_LIBRARY_PATH` and `DYLD_FRAMEWORK_PATH` environment variables to locate shared libraries, frameworks and executables of your requirements at runtime.

> **Warning:** The `RunEnvironment` is no longer needed, at least explicitly in *conanfile.py*. It has been integrated into the `self.run(..., run_environment=True)` argument. Check *self.run()*.

This helper is specially useful if:

- You are requiring packages with shared libraries and you are running some executable that needs those libraries.
- You have a requirement with some tool (executable) and you need it to be in the path.

```
from conans import ConanFile, RunEnvironment

class ExampleConan(ConanFile):
   ...

   def build(self):
      env_build = RunEnvironment(self)
      with tools.environment_append(env_build.vars):
         self.run("....")
         # All the requirements bin folder will be available at PATH
         # All the lib folders will be available in LD_LIBRARY_PATH and DYLD_LIBRARY_PATH
         # All the framework_paths folders will be available in DYLD_FRAMEWORK_PATH
```

It sets the following environment variables:

NAME	DESCRIPTION
PATH	Containing all the requirements `bin` folders.
LD_LIBRARY_PATH	Containing all the requirements `lib` folders. (Linux)
DYLD_LIBRARY_PATH	Containing all the requirements `lib` folders. (OSX)
DYLD_FRAMEWORK_PATH	Containing all the requirements `framework_paths` folders. (OSX)

Important: Security restrictions might apply in OSX (read this thread), so the `DYLD_LIBRARY_PATH` and `DYLD_FRAMEWORK_PATH` environment variables are not directly transferred to the child process. In that case, you have to use it explicitly in your *conanfile.py*:

```
def build(self):
    env_build = RunEnvironment(self)
    with tools.environment_append(env_build.vars):
        # self.run("./myexetool") # won't work, even if 'DYLD_LIBRARY_PATH' and 'DYLD_
↪FRAMEWORK_PATH' are in the env
        self.run("DYLD_LIBRARY_PATH=%s DYLD_FRAMEWORK_PATH=%s ./myexetool" % (os.environ[
↪'DYLD_LIBRARY_PATH'], os.environ['DYLD_FRAMEWORK_PATH']))
```

This is already handled automatically by the `self.run(..., run_environment=True)` argument.

See also:

- *Manage Shared Libraries with Environment Variables*

- *tools.environment_append()*

Important: If you need a build helper for any other tools, please check how you can create your own *Creating a custom build helper for Conan*.

18.7 Tools

Under the tools module there are several functions and utilities that can be used in Conan package recipes:

```
from conans import ConanFile
from conans import tools

class ExampleConan(ConanFile):
    ...
```

18.7.1 tools.cpu_count()

```
def tools.cpu_count()
```

Returns the number of CPUs available, for parallel builds. If processor detection is not enabled, it will safely return 1. When running in Docker, it reads cgroup to detect the configured number of CPUs. It Can be overwritten with the environment variable *CONAN_CPU_COUNT* and configured in the *conan.conf*.

18.7.2 tools.vcvars_command()

```
def vcvars_command(conanfile, arch=None, compiler_version=None, force=False, vcvars_
↪ver=None,
                   winsdk_version=None)
```

Returns, for given settings, the command that should be called to load the Visual Studio environment variables for a certain Visual Studio version. It wraps the functionality of vcvarsall but does not execute the command, as that typically have to be done in the same command as the compilation, so the variables are loaded for the same subprocess. It will be typically used in the `build()` method, like this:

```
from conans import tools

def build(self):
    if self.settings.build_os == "Windows":
        vcvars_command = tools.vcvars_command(self)
        build_command = ...
        self.run("%s && configure %s" % (vcvars_command, " ".join(args)))
        self.run("%s && %s %s" % (vcvars, build_command, " ".join(build_args)))
```

The `vcvars_command` string will contain something like `call "%vsXX0comntools%../../VC/vcvarsall.bat"` for the corresponding Visual Studio version for the current settings.

This is typically not needed if using CMake, as the `cmake` generator will handle the correct Visual Studio version.

If **arch** or **compiler_version** is specified, it will ignore the settings and return the command to set the Visual Studio environment for these parameters.

Parameters:

- **conanfile** (Required): Conanfile object. Use `self` in a `conanfile.py`.
- **arch** (Optional, Defaulted to `None`): Will use `settings.arch`.
- **compiler_version** (Optional, Defaulted to `None`): Will use `settings.compiler.version`.
- **force** (Optional, Defaulted to `False`): Will ignore if the environment is already set for a different Visual Studio version.
- **winsdk_version** (Optional, Defaulted to `None`): Specifies the version of the Windows SDK to use.
- **vcvars_ver** (Optional, Defaulted to `None`): Specifies the Visual Studio compiler toolset to use.

Note: When cross-building from x64 to x86 the toolchain by default is `x86`. If you want to use `amd64_x86` instead, set the environment variable `PreferredToolArchitecture=x64`.

18.7.3 tools.vcvars_dict()

```
vcvars_dict(conanfile, arch=None, compiler_version=None, force=False, filter_known_
↪paths=False,
            vcvars_ver=None, winsdk_version=None, only_diff=True)
```

Returns a dictionary with the variables set by the *tools.vcvars_command()* that can be directly applied to *tools.environment_append()*.

The values of the variables `INCLUDE`, `LIB`, `LIBPATH` and `PATH` will be returned as a list. When used with *tools.environment_append()*, the previous environment values that these variables may have will be appended automatically.

```
from conans import tools

def build(self):
    env_vars = tools.vcvars_dict(self)
    with tools.environment_append(env_vars):
        # Do something
```

Parameters:

- Same as *tools.vcvars_command()*.
- **filter_known_paths** (Optional, Defaulted to `False`): When True, the function will only keep the `PATH` entries that follows some known patterns, filtering all the non-Visual Studio ones. When False, it will keep the `PATH` will all the system entries.
- **only_diff** (Optional, Defaulted to `True`): When True, the command will return only the variables set by `vcvarsall` and not the whole environment. If *vcvars* modifies an environment variable by appending values to the old value (separated by `;`), only the new values will be returned, as a list.

18.7.4 tools.vcvars()

```
vcvars(conanfile, arch=None, compiler_version=None, force=False, filter_known_
↪paths=False)
```

Note: This context manager tool has no effect if used in a platform different from Windows.

This is a context manager that allows to append to the environment all the variables set by the *tools.vcvars_dict()*. You can replace *tools.vcvars_command()* and use this context manager to get a cleaner way to activate the Visual Studio environment:

```
from conans import tools

def build(self):
    with tools.vcvars(self):
        do_something()
```

18.7.5 tools.build_sln_command() [DEPRECATED]

Warning: This tool is deprecated and will be removed in Conan 2.0. Use *MSBuild()* build helper instead.

```
def build_sln_command(settings, sln_path, targets=None, upgrade_project=True, build_
↪type=None,
                      arch=None, parallel=True, toolset=None, platforms=None,␣
↪verbosity=None,
                      definitions=None)
```

Returns the command to call *devenv* and *msbuild* to build a Visual Studio project. It's recommended to use it with *tools.vcvars_command()*, so that the Visual Studio tools will be in path.

```
from conans import tools

def build(self):
    build_command = build_sln_command(self.settings, "myfile.sln", targets=["SDL2_image
↪"])
    command = "%s && %s" % (tools.vcvars_command(self.settings), build_command)
    self.run(command)
```

Parameters:

- **settings** (Required): Conanfile settings. Use "self.settings".
- **sln_path** (Required): Visual Studio project file path.
- **targets** (Optional, Defaulted to `None`): List of targets to build.
- **upgrade_project** (Optional, Defaulted to `True`): If `True`, the project file will be upgraded if the project's VS version is older than current. When *CONAN_SKIP_VS_PROJECTS_UPGRADE* environment variable is set to `True`/`1`, this parameter will be ignored and the project won't be upgraded.
- **build_type** (Optional, Defaulted to `None`): Override the build type defined in the settings (`settings.build_type`).

- **arch** (Optional, Defaulted to `None`): Override the architecture defined in the settings (`settings.arch`).
- **parallel** (Optional, Defaulted to `True`): Enables Visual Studio parallel build with `/m:X` argument, where X is defined by *CONAN_CPU_COUNT* environment variable or by the number of cores in the processor by default.
- **toolset** (Optional, Defaulted to `None`): Specify a toolset. Will append a `/p:PlatformToolset` option.
- **platforms** (Optional, Defaulted to `None`): Dictionary with the mapping of archs/platforms from Conan naming to another one. It is useful for Visual Studio solutions that have a different naming in architectures. Example: `platforms={"x86":"Win32"}` (Visual solution uses "Win32" instead of "x86"). This dictionary will update the following default one:

```
msvc_arch = {'x86': 'x86',
             'x86_64': 'x64',
             'armv7': 'ARM',
             'armv8': 'ARM64'}
```

- **verbosity** (Optional, Defaulted to `None`): Specifies verbosity level (`/verbosity:` parameter).
- **definitions** (Optional, Defaulted to `None`): Dictionary with additional compiler definitions to be applied during the build. Use value of None to set compiler definition with no value.

18.7.6 tools.msvc_build_command() [DEPRECATED]

Warning: This tool is deprecated and will be removed in Conan 2.0. Use *MSBuild()*.get_command() instead.

```
def msvc_build_command(settings, sln_path, targets=None, upgrade_project=True, build_
↪type=None,
                       arch=None, parallel=True, force_vcvars=False, toolset=None,␣
↪platforms=None)
```

Returns a string with a joint command consisting in setting the environment variables via `vcvars.bat` with the above *tools.vcvars_command()* function, and building a Visual Studio project with the *tools.build_sln_command() [DEPRECATED]* function.

Parameters:

- Same parameters as the above *tools.build_sln_command() [DEPRECATED]*.
- **force_vcvars**: Optional. Defaulted to False. Will set `tools.vcvars_command(force=force_vcvars)`.

18.7.7 tools.unzip()

```
def unzip(filename, destination=".", keep_permissions=False, pattern=None, strip_
↪root=False)
```

Function mainly used in `source()`, but could be used in `build()` in special cases, as when retrieving pre-built binaries from the Internet.

This function accepts `.tar.gz`, `.tar`, `.tzb2`, `.tar.bz2`, `.tgz`, `.txz`, `tar.xz`, and `.zip` files, and decompresses them into the given destination folder (the current one by default).

It also accepts gzipped files, with extension `.gz` (not matching any of the above), and it will unzip them into a file with the same name but without the extension, or to a filename defined by the `destination` argument.

```
from conans import tools

tools.unzip("myfile.zip")
# or to extract in "myfolder" sub-folder
tools.unzip("myfile.zip", "myfolder")
```

You can keep the permissions of the files using the `keep_permissions=True` parameter.

```
from conans import tools

tools.unzip("myfile.zip", "myfolder", keep_permissions=True)
```

Use `pattern=None` if you want to filter specific files and paths to decompress from the archive.

```
from conans import tools

# Extract only files inside relative folder "small"
tools.unzip("bigfile.zip", pattern="small/*")
# Extract only txt files
tools.unzip("bigfile.zip", pattern="*.txt")
```

Parameters:

- **filename** (Required): File to be unzipped.
- **destination** (Optional, Defaulted to `"."`): Destination folder for unzipped files.
- **keep_permissions** (Optional, Defaulted to `False`): Keep permissions of files. **WARNING:** Can be dangerous if the zip was not created in a NIX system, the bits could produce undefined permission schema. Use only this option if you are sure that the zip was created correctly.
- **pattern** (Optional, Defaulted to `None`): Extract from the archive only paths matching the pattern. This should be a Unix shell-style wildcard. See fnmatch documentation for more details.
- **strip_root** (Optional, Defaulted to `False`): When `True` and the ZIP file contains one folder containing all the contents, it will strip the root folder moving all its contents to the root. E.g: *mylib-1.2.8/main.c* will be extracted as *main.c*. If the compressed file contains more than one folder or only a file it will raise a `ConanException`.

18.7.8 tools.untargz()

```
def untargz(filename, destination=".", pattern=None, strip_root=False)
```

Extract *.tar.gz* files (or in the family). This is the function called by the previous `unzip()` for the matching extensions, so generally not needed to be called directly, call `unzip()` instead unless the file had a different extension.

```
from conans import tools

tools.untargz("myfile.tar.gz")
# or to extract in "myfolder" sub-folder
tools.untargz("myfile.tar.gz", "myfolder")
# or to extract only txt files
tools.untargz("myfile.tar.gz", pattern="*.txt")
```

Parameters:

- **filename** (Required): File to be unzipped.
- **destination** (Optional, Defaulted to "."): Destination folder for *untargzed* files.
- **pattern** (Optional, Defaulted to None): Extract from the archive only paths matching the pattern. This should be a Unix shell-style wildcard. See fnmatch documentation for more details.
- **strip_root** (Optional, Defaulted to False): When True and the tar.gz file contains one folder containing all the contents, it will strip the root folder moving all its contents to the root. E.g: *mylib-1.2.8/main.c* will be extracted as *main.c*. If the compressed file contains more than one folder or only a file it will raise a ConanException.

18.7.9 tools.get()

```
def get(url, md5='', sha1='', sha256='', destination=".", filename="", keep_
↪permissions=False,
        pattern=None, requester=None, output=None, verify=True, retry=None, retry_
↪wait=None,
        overwrite=False, auth=None, headers=None, strip_root=False)
```

Just a high level wrapper for download, unzip, and remove the temporary zip file once unzipped. You can pass hash checking parameters: md5, sha1, sha256. All the specified algorithms will be checked. If any of them doesn't match, it will raise a ConanException.

```
from conans import tools

tools.get("http://url/file", md5='d2da0cd0756cd9da6560b9a56016a0cb')
# also, specify a destination folder
tools.get("http://url/file", destination="subfolder")
```

Parameters:

- **url** (Required): URL to download. It can be a list, which only the first one will be downloaded, and the follow URLs will be used as mirror in case of a download error.
- **md5** (Optional, Defaulted to ""): MD5 hash code to check the downloaded file.
- **sha1** (Optional, Defaulted to ""): SHA-1 hash code to check the downloaded file.
- **sha256** (Optional, Defaulted to ""): SHA-256 hash code to check the downloaded file.
- **filename** (Optional, Defaulted to ""): Specify the name of the compressed file if it cannot be deduced from the URL.
- **keep_permissions** (Optional, Defaulted to False): Propagates the parameter to *tools.unzip()*.
- **pattern** (Optional, Defaulted to None): Propagates the parameter to *tools.unzip()*.
- **requester** (Optional, Defaulted to None): HTTP requests instance
- **output** (Optional, Defaulted to None): Stream object.
- **verify** (Optional, Defaulted to True): When False, disables https certificate validation.
- **retry** (Optional, Defaulted to 2): Number of retries in case of failure. Default is overriden by general.retry in the *conan.conf* file or an env variable CONAN_RETRY.
- **retry_wait** (Optional, Defaulted to 5): Seconds to wait between download attempts. Default is overriden by general.retry_wait in the *conan.conf* file or an env variable CONAN_RETRY_WAIT.

- **overwrite**: (Optional, Defaulted to `False`): When `True` Conan will overwrite the destination file if it exists. Otherwise it will raise.
- **auth** (Optional, Defaulted to `None`): A tuple of user, password can be passed to use HTTPBasic authentication. This is passed directly to the `requests` Python library. Check here other uses of the **auth** parameter: https://requests.readthedocs.io/en/master/user/authentication/#basic-authentication
- **headers** (Optional, Defaulted to `None`): A dictionary with additional headers.
- **strip_root** (Optional, Defaulted to `False`): When `True` and the compressed file contains one folder containing all the contents, it will strip the root folder moving all its contents to the root. E.g: *mylib-1.2.8/main.c* will be extracted as *main.c*. If the compressed file contains more than one folder or only a file it will raise a `ConanException`.

18.7.10 tools.get_env()

```
def get_env(env_key, default=None, environment=None)
```

Parses an environment and cast its value against the **default** type passed as an argument. Following Python conventions, returns **default** if **env_key** is not defined.

This is a usage example with an environment variable defined while executing Conan:

```
$ TEST_ENV="1" conan <command> ...
```

```
from conans import tools

tools.get_env("TEST_ENV") # returns "1", returns current value
tools.get_env("TEST_ENV_NOT_DEFINED") # returns None, TEST_ENV_NOT_DEFINED not declared
tools.get_env("TEST_ENV_NOT_DEFINED", []) # returns [], TEST_ENV_NOT_DEFINED not declared
tools.get_env("TEST_ENV", "2") # returns "1"
tools.get_env("TEST_ENV", False) # returns True (default value is boolean)
tools.get_env("TEST_ENV", 2) # returns 1
tools.get_env("TEST_ENV", 2.0) # returns 1.0
tools.get_env("TEST_ENV", []) # returns ["1"]
```

Parameters:

- **env_key** (Required): environment variable name.
- **default** (Optional, Defaulted to `None`): default value to return if not defined or cast value against.
- **environment** (Optional, Defaulted to `None`): `os.environ` if `None` or environment dictionary to look for.

18.7.11 tools.download()

```
def download(url, filename, verify=True, out=None, retry=None, retry_wait=None,␣
↪overwrite=False,
             auth=None, headers=None, requester=None, md5='', sha1='', sha256='')
```

Retrieves a file from a given URL into a file with a given filename. It uses certificates from a list of known verifiers for https downloads, but this can be optionally disabled.

You can pass hash checking parameters: `md5`, `sha1`, `sha256`. All the specified algorithms will be checked. If any of them doesn't match, the downloaded file will be removed and it will raise a `ConanException`.

```
from conans import tools

tools.download("http://someurl/somefile.zip", "myfilename.zip")

# to disable verification:
tools.download("http://someurl/somefile.zip", "myfilename.zip", verify=False)

# to retry the download 2 times waiting 5 seconds between them
tools.download("http://someurl/somefile.zip", "myfilename.zip", retry=2, retry_wait=5)

# Use https basic authentication
tools.download("http://someurl/somefile.zip", "myfilename.zip", auth=("user", "password
↪"))

# Pass some header
tools.download("http://someurl/somefile.zip", "myfilename.zip", headers={"Myheader": "My␣
↪value"})

# Download and check file checksum
tools.download("http://someurl/somefile.zip", "myfilename.zip", md5=
↪"e5d695597e9fa520209d1b41edad2a27")

# to add mirrors
tools.download(["https://ftp.gnu.org/gnu/gcc/gcc-9.3.0/gcc-9.3.0.tar.gz",
                "http://mirror.linux-ia64.org/gnu/gcc/releases/gcc-9.3.0/gcc-9.3.0.tar.gz
↪"], "gcc-9.3.0.tar.gz",
               sha256="5258a9b6afe9463c2e56b9e8355b1a4bee125ca828b8078f910303bc2ef91fa6")
```

Parameters:

- **url** (Required): URL to download. It can be a list, which only the first one will be downloaded, and the follow URLs will be used as mirror in case of download error.
- **filename** (Required): Name of the file to be created in the local storage
- **verify** (Optional, Defaulted to `True`): When False, disables https certificate validation.
- **out**: (Optional, Defaulted to `None`): An object with a `write()` method can be passed to get the output. `stdout` will use if not specified.
- **retry** (Optional, Defaulted to 1): Number of retries in case of failure. Default is overriden by `general.retry` in the *conan.conf* file or an env variable `CONAN_RETRY`.
- **retry_wait** (Optional, Defaulted to 5): Seconds to wait between download attempts. Default is overriden by `general.retry_wait` in the *conan.conf* file or an env variable `CONAN_RETRY_WAIT`.
- **overwrite**: (Optional, Defaulted to `False`): When `True`, Conan will overwrite the destination file if exists. Otherwise it will raise an exception.
- **auth** (Optional, Defaulted to `None`): A tuple of user and password to use HTTPBasic authentication. This is used directly in the `requests` Python library. Check other uses here: https://requests.readthedocs.io/en/master/user/authentication/#basic-authentication
- **headers** (Optional, Defaulted to `None`): A dictionary with additional headers.
- **requester** (Optional, Defaulted to `None`): HTTP requests instance
- **md5** (Optional, Defaulted to `""`): MD5 hash code to check the downloaded file.

- **sha1** (Optional, Defaulted to ""): SHA-1 hash code to check the downloaded file.
- **sha256** (Optional, Defaulted to ""): SHA-256 hash code to check the downloaded file.

18.7.12 tools.ftp_download()

```
def ftp_download(ip, filename, login="", password="")
```

Retrieves a file from an FTP server. This doesn't support SSL, but you might implement it yourself using the standard Python FTP library.

```
from conans import tools

def source(self):
    tools.ftp_download('ftp.debian.org', "debian/README")
    self.output.info(load("README"))
```

Parameters:

- **ip** (Required): The IP or address of the ftp server.
- **filename** (Required): The filename, including the path/folder where it is located.
- **login** (Optional, Defaulted to ""): Login credentials for the ftp server.
- **password** (Optional, Defaulted to ""): Password credentials for the ftp server.

18.7.13 tools.replace_in_file()

```
def replace_in_file(file_path, search, replace, strict=True, encoding=None)
```

This function is useful for a simple "patch" or modification of source files. A typical use would be to augment some library existing *CMakeLists.txt* in the `source()` method of a *conanfile.py*, so it uses Conan dependencies without forking or modifying the original project:

```
from conans import tools

def source(self):
    # get the sources from somewhere
    tools.replace_in_file("hello/CMakeLists.txt", "PROJECT(MyHello)",
        '''PROJECT(MyHello)
           include(${CMAKE_BINARY_DIR}/conanbuildinfo.cmake)
           conan_basic_setup()''')
```

Parameters:

- **file_path** (Required): File path of the file to perform the replace in.
- **search** (Required): String you want to be replaced.
- **replace** (Required): String to replace the searched string.
- **strict** (Optional, Defaulted to `True`): If `True`, it raises an error if the searched string is not found, so nothing is actually replaced.

- **encoding** (Optional, Defaulted to `None`): Specifies the input and output files text encoding. The `None` value has a special meaning - perform the encoding detection by checking the BOM (byte order mask), if no BOM is present tries to use: `utf-8`, `cp1252`. In case of `None`, the output file is saved to the `utf-8`

18.7.14 tools.replace_path_in_file()

```
def replace_path_in_file(file_path, search, replace, strict=True, windows_paths=None,
                         encoding=None)
```

Replace a path in a file with another string. In Windows, it will match the path even if the casing and the path separator doesn't match.

```
from conans import tools

def build(self):
    tools.replace_path_in_file("hello/somefile.cmake", "c:\Some/PATH/to\File.txt",
↪"PATTERN/file.txt")
```

Parameters:

- **file_path** (Required): File path of the file to perform the replace in.
- **search** (Required): String with the path you want to be replaced.
- **replace** (Required): String to replace the searched path.
- **strict** (Optional, Defaulted to `True`): If `True`, it raises an error if the search string is not found and nothing is actually replaced.
- **windows_paths** (Optional, Defaulted to `None`): Controls whether the casing of the path and the different directory separators are taken into account:
  - `None`: Only when Windows operating system is detected.
  - `False`: Deactivated, it will match exact patterns (like *tools.replace_in_file()*).
  - `True`: Always activated, irrespective of the detected operating system.
- **encoding** (Optional, Defaulted to `None`): Specifies the input and output files text encoding. The `None` value has a special meaning - perform the encoding detection by checking the BOM (byte order mask), if no BOM is present tries to use: `utf-8`, `cp1252`. In case of `None`, the output file is saved to the `utf-8`

18.7.15 tools.run_environment()

```
def run_environment(conanfile)
```

Context manager that sets temporary environment variables set by *RunEnvironment*.

18.7.16 tools.check_with_algorithm_sum()

```
def check_with_algorithm_sum(algorithm_name, file_path, signature)
```

Useful to check that some downloaded file or resource has a predefined hash, so integrity and security are guaranteed. Something that could be typically done in `source()` method after retrieving some file from the internet.

Parameters:

- **algorithm_name** (Required): Name of the algorithm to be checked.
- **file_path** (Required): File path of the file to be checked.
- **signature** (Required): Hash code that the file should have.

There are specific functions for common algorithms:

```
def check_sha1(file_path, signature)
def check_md5(file_path, signature)
def check_sha256(file_path, signature)
```

For example:

```
from conans import tools

tools.check_sha1("myfile.zip", "eb599ec83d383f0f25691c184f656d40384f9435")
```

Other algorithms are also possible, as long as are recognized by python `hashlib` implementation, via `hashlib.new(algorithm_name)`. The previous is equivalent to:

```
from conans import tools

tools.check_with_algorithm_sum("sha1", "myfile.zip",
                               "eb599ec83d383f0f25691c184f656d40384f9435")
```

18.7.17 tools.patch()

```
def patch(base_path=None, patch_file=None, patch_string=None, strip=0, output=None,␣
↪fuzz=False)
```

Applies a patch from a file or from a string into the given path. The patch should be in diff (unified diff) format. Use it preferably in the `build()` method.

```
from conans import tools

# from a file
def build(self):
    tools.patch(patch_file="file.patch")

# from a string:
def build(self):
    patch_content = " real patch content ..."
    tools.patch(patch_string=patch_content)
```

(continues on next page)

(continued from previous page)

```
# to apply in subfolder
def build(self):
    tools.patch(base_path=mysubfolder, patch_string=patch_content)

# from conandata
def build(self):
    tools.patch(**self.conan_data["patches"][self.version])

# from conandata, using multiple versions
def build(self):
    for patch in self.conan_data.get("patches", {}).get(self.version, []):
        tools.patch(**patch)
```

If the patch to be applied uses alternate paths that have to be stripped like this example:

```
--- old_path/text.txt\t2016-01-25 17:57:11.452848309 +0100
+++ new_path/text_new.txt\t2016-01-25 17:57:28.839869950 +0100
@@ -1 +1 @@
- old content
+ new content
```

Then, the number of folders to be stripped from the path can be specified:

```
from conans import tools

tools.patch(patch_file="file.patch", strip=1)
```

If the patch to be applied differs from the source (fuzzy) the patch will fail by default, however, you can force it using the `fuzz` option:

```
from conans import tools

tools.patch(patch_file="file.patch", fuzz=True)
```

When creating a header-only package and there is no usage for build step or build folder, the patch can be applied to the source folder:

```
from conans import tools, ConanFile

class HeaderOnly(ConanFile):
    no_copy_source = True

    def source(self):
        ...
        tools.patch(patch_file="file.patch")
```

Parameters:

- **base_path** (Optional, Defaulted to `None`): Base path where the patch should be applied.
- **patch_file** (Optional, Defaulted to `None`): Patch file that should be applied.
- **patch_string** (Optional, Defaulted to `None`): Patch string that should be applied.
- **strip** (Optional, Defaulted to `0`): Number of folders to be stripped from the path.

- **output** (Optional, Defaulted to `None`): Stream object.
- **fuzz** (Optional, Defaulted to `False`): Accept fuzzy patches.

18.7.18 tools.environment_append()

```
def environment_append(env_vars)
```

This is a context manager that allows to temporary use environment variables for a specific piece of code in your conanfile:

```
from conans import tools

def build(self):
    with tools.environment_append({"MY_VAR": "3", "CXX": "/path/to/cxx", "CPPFLAGS":␣
↪None}):
        do_something()
```

The environment variables will be overridden if the value is a string, while it will be prepended if the value is a list. Additionally, if value is `None`, the given environment variable is unset (In the previous example, `CPPFLAGS` environment variable will be unset), and in case variable wasn't set prior to the invocation, it has no effect on the given variable (`CPPFLAGS`). When the context manager block ends, the environment variables will recover their previous state.

Parameters:

- **env_vars** (Required): Dictionary object with environment variable name and its value.

18.7.19 tools.chdir()

```
def chdir(newdir)
```

This is a context manager that allows to temporary change the current directory in your conanfile:

```
from conans import tools

def build(self):
    with tools.chdir("./subdir"):
        do_something()
```

Parameters:

- **newdir** (Required): Directory path name to change the current directory.

18.7.20 tools.pythonpath()

> **Warning:** This way of reusing python code from other recipes can be improved via *Python requires*.

This tool is automatically applied in the conanfile methods unless *apply_env* is deactivated, so any `PYTHONPATH` inherited from the requirements will be automatically available.

```
def pythonpath(conanfile)
```

This is a context manager that allows to load the PYTHONPATH for dependent packages, create packages with Python code and reuse that code into your own recipes.

For example:

```
from conans import tools

def build(self):
    with tools.pythonpath(self):
        from module_name import whatever
        whatever.do_something()
```

When the *apply_env* is activated (default) the above code could be simplified as:

```
from conans import tools

def build(self):
    from module_name import whatever
    whatever.do_something()
```

For that to work, one of the dependencies of the current recipe, must have a `module_name` file or folder with a `whatever` file or object inside, and should have declared in its `package_info()`:

```
from conans import tools

def package_info(self):
    self.env_info.PYTHONPATH.append(self.package_folder)
```

Parameters:

- **conanfile** (Required): Current `ConanFile` object.

18.7.21 tools.no_op()

```
def no_op()
```

Context manager that performs nothing. Useful to condition any other context manager to get a cleaner code:

```
from conans import tools

def build(self):
    with tools.chdir("some_dir") if self.options.myoption else tools.no_op():
        # if not self.options.myoption, we are not in the "some_dir"
        pass
```

18.7.22 tools.human_size()

```
def human_size(size_bytes)
```

Will return a string from a given number of bytes, rounding it to the most appropriate unit: GB, MB, KB, etc. It is mostly used by the Conan downloads and unzip progress.

```
from conans import tools

tools.human_size(1024)
>> 1.0KB
```

Parameters:

- **size_bytes** (Required): Number of bytes.

18.7.23 tools.OSInfo and tools.SystemPackageTool

These are helpers to install system packages. Check *system_requirements()*.

18.7.24 tools.cross_building()

```
def cross_building(conanfile, self_os=None, self_arch=None, skip_x64_x86=False)
```

Evaluates operating system and architecture from the `host` machine and the `build` machine to return a boolean `True` if it is a cross building scenario. Settings from `host` machine are taken from the `conanfile.settings`, while setting from the `build` context can provide from different sources:

- if `conanfile.settings_build` is available (Conan was called with a `--profile:build`) it will use settings in that profile (read more about *Build and Host contexts*).
- otherwise, the values for the `build` context will come from (in this order of precedence): `self_os` and `self_arch` if they are given to the function, the values for `os_build` and `arch_build` from `conanfile.settings` or auto-detected.

This tool can be used to run special actions depending on its return value:

```
from conans import tools

if tools.cross_building(self):
    # Some special action
```

Parameters:

- **conanfile** (Required): Conanfile object. Use `self` in a `conanfile.py`.
- **self_os** (Optional, Defaulted to `None`): Current operating system where the build is being done.
- **self_arch** (Optional, Defaulted to `None`): Current architecture where the build is being done.
- **skip_x64_x86** (Optional, Defaulted to `False`): Do not consider building for `x86` host from `x86_64` build machine as cross building, in case of host and build machine use the same operating system. Normally, in such case build machine may execute binaries produced for the target machine, and special cross-building handling may not be needed.

18.7.25 tools.get_gnu_triplet()

```
def get_gnu_triplet(os_, arch, compiler=None)
```

Returns string with GNU like <machine>-<vendor>-<op_system> triplet.

Parameters:

- **os_** (Required): Operating system to be used to create the triplet.
- **arch** (Required): Architecture to be used to create the triplet.
- **compiler** (Optional, Defaulted to `None`): Compiler used to create the triplet (only needed for Windows).

18.7.26 tools.run_in_windows_bash()

```
def run_in_windows_bash(conanfile, bashcmd, cwd=None, subsystem=None, msys_mingw=True,␣
↪env=None, with_login=True)
```

Runs a UNIX command inside a bash shell. It requires to have "bash" in the path. Useful to build libraries using `configure` and `make` in Windows. Check *Windows subsytems* section.

You can customize the path of the bash executable using the environment variable `CONAN_BASH_PATH` or the *conan.conf* `bash_path` variable to change the default bash location.

```
from conans import tools

command = "pwd"
tools.run_in_windows_bash(self, command) # self is a conanfile instance
```

Parameters:

- **conanfile** (Required): Current `ConanFile` object.
- **bashcmd** (Required): String with the command to be run.
- **cwd** (Optional, Defaulted to `None`): Path to directory where to apply the command from.
- **subsystem** (Optional, Defaulted to `None` will autodetect the subsystem): Used to escape the command according to the specified subsystem.
- **msys_mingw** (Optional, Defaulted to `True`): If the specified subsystem is MSYS2, will start it in MinGW mode (native windows development).
- **env** (Optional, Defaulted to `None`): You can pass a dictionary with environment variable to be applied **at first place** so they will have more priority than others.
- **with_login** (Optional, Defaulted to `True`): Pass the `--login` flag to **bash** command. This might come handy when you don't want to create a fresh user session for running the command.

18.7.27 tools.get_cased_path()

```
get_cased_path(abs_path)
```

This function converts a case-insensitive absolute path to a case-sensitive one. That is, with the real cased characters. Useful when using Windows subsystems where the file system is case-sensitive.

18.7.28 tools.detected_os()

```
detected_os()
```

It returns the recognized OS name e.g "Macos", "Windows". Otherwise it will return the value from `platform.system()`.

18.7.29 tools.remove_from_path()

```
remove_from_path(command)
```

This is a context manager that allows you to remove a tool from the `PATH`. Conan will locate the executable (using *tools.which()*) and will remove from the `PATH` the directory entry that contains it. It's not necessary to specify the extension.

```
from conans import tools

with tools.remove_from_path("make"):
    self.run("some command")
```

18.7.30 tools.unix_path()

```
def unix_path(path, path_flavor=None)
```

Used to translate Windows paths to MSYS/CYGWIN Unix paths like `c/users/path/to/file`.

Parameters:

- **path** (Required): Path to be converted.
- **path_flavor** (Optional, Defaulted to `None`, will try to autodetect the subsystem): Type of Unix path to be returned. Options are `MSYS`, `MSYS2`, `CYGWIN`, `WSL` and `SFU`.

18.7.31 tools.escape_windows_cmd()

```
def escape_windows_cmd(command)
```

Useful to escape commands to be executed in a windows bash (msys2, cygwin etc).

- Adds escapes so the argument can be unpacked by `CommandLineToArgvW()`.
- Adds escapes for *cmd.exe* so the argument survives to `cmd.exe`'s substitutions.

Parameters:

- **command** (Required): Command to execute.

18.7.32 tools.sha1sum(), sha256sum(), md5sum()

```
def def md5sum(file_path)
def sha1sum(file_path)
def sha256sum(file_path)
```

Return the respective hash or checksum for a file.

```
from conans import tools

md5 = tools.md5sum("myfilepath.txt")
sha1 = tools.sha1sum("myfilepath.txt")
```

Parameters:

- **file_path** (Required): Path to the file.

18.7.33 tools.md5()

```
def md5(content)
```

Returns the MD5 hash for a string or byte object.

```
from conans import tools

md5 = tools.md5("some string, not a file path")
```

Parameters:

- **content** (Required): String or bytes to calculate its md5.

18.7.34 tools.save()

```
def save(path, content, append=False, encoding="utf-8")
```

Utility function to save files in one line. It will manage the open and close of the file and creating directories if necessary.

```
from conans import tools

tools.save("otherfile.txt", "contents of the file")
```

Parameters:

- **path** (Required): Path to the file.
- **content** (Required): Content that should be saved into the file.
- **append** (Optional, Defaulted to `False`): If `True`, it will append the content.
- **encoding** (Optional, Defaulted to `utf-8`): Specifies the output file text encoding.

18.7.35 tools.load()

```
def load(path, binary=False, encoding="auto")
```

Utility function to load files in one line. It will manage the open and close of the file, and load binary encodings. Returns the content of the file.

```
from conans import tools

content = tools.load("myfile.txt")
```

Parameters:

- **path** (Required): Path to the file.
- **binary** (Optional, Defaulted to `False`): If `True`, it reads the the file as binary code.
- **encoding** (Optional, Defaulted to `auto`): Specifies the input file text encoding. The `auto` value has a special meaning - perform the encoding detection by checking the BOM (byte order mask), if no BOM is present tries to use: `utf-8`, `cp1252`. The value is ignored in case of `binary` set to the `True`.

18.7.36 tools.mkdir(), tools.rmdir()

```
def mkdir(path)
def rmdir(path)
```

Utility functions to create/delete a directory. The existence of the specified directory is checked, so `mkdir()` will do nothing if the directory already exists and `rmdir()` will do nothing if the directory does not exists.

This makes it safe to use these functions in the `package()` method of a *conanfile.py* when `no_copy_source=True`.

```
from conans import tools

tools.mkdir("mydir") # Creates mydir if it does not already exist
tools.mkdir("mydir") # Does nothing

tools.rmdir("mydir") # Deletes mydir
tools.rmdir("mydir") # Does nothing
```

Parameters:

- **path** (Required): Path to the directory.

18.7.37 tools.which()

```
def which(filename)
```

Returns the path to a specified executable searching in the `PATH` environment variable. If not found, it returns `None`.

This tool also looks for filenames with following extensions if no extension provided:

- `.com`, `.exe`, `.bat` `.cmd` for Windows.
- `.sh` if not Windows.

```
from conans import tools

abs_path_make = tools.which("make")
```

Parameters:

- **filename** (Required): Name of the executable file. It doesn't require the extension of the executable.

18.7.38 tools.unix2dos()

```
def unix2dos(filepath)
```

Converts line breaks in a text file from Unix format (LF) to DOS format (CRLF).

```
from conans import tools

tools.unix2dos("project.dsp")
```

Parameters:

- **filepath** (Required): The file to convert.

18.7.39 tools.dos2unix()

```
def dos2unix(filepath)
```

Converts line breaks in a text file from DOS format (CRLF) to Unix format (LF).

```
from conans import tools

tools.dos2unix("dosfile.txt")
```

Parameters:

- **filepath** (Required): The file to convert.

18.7.40 tools.rename()

Available since: 1.29.0

```
def rename(src, dst)
```

Utility functions to rename a file or folder *src* to *dst* with retrying. `os.rename()` frequently raises "Access is denied" exception on windows. This function renames file or folder using robocopy to avoid the exception on windows.

```
from conans import tools
tools.rename("src_dir", "dst_dir") # renaming a folder
```

Parameters:

- **src** (Required): Path to be renamed.
- **dst** (Required): Path to be renamed to.

18.7.41 tools.touch()

```
def touch(fname, times=None)
```

Updates the timestamp (last access and last modification times) of a file. This is similar to Unix' `touch` command except that this one fails if the file does not exist.

Optionally, a tuple of two numbers can be specified, which denotes the new values for the last access and last modified times respectively.

```
from conans import tools
import time

tools.touch("myfile")                              # Sets atime and mtime to the current␣
↪time
tools.touch("myfile", (time.time(), time.time()) # Similar to above
tools.touch("myfile", (time.time(), 1))          # Modified long, long ago
```

Parameters:

- **fname** (Required): File name of the file to be touched.
- **times** (Optional, Defaulted to `None`: Tuple with 'last access' and 'last modified' times.

18.7.42 tools.relative_dirs()

```
def relative_dirs(path)
```

Recursively walks a given directory (using `os.walk()`) and returns a list of all contained file paths relative to the given directory.

```
from conans import tools

tools.relative_dirs("mydir")
```

Parameters:

- **path** (Required): Path of the directory.

18.7.43 tools.vswhere()

```
def vswhere(all_=False, prerelease=False, products=None, requires=None, version="",
            latest=False, legacy=False, property_="", nologo=True)
```

Wrapper of `vswhere` tool to look for details of Visual Studio installations. Its output is always a list with a dictionary for each installation found.

```
from conans import tools

vs_legacy_installations = tool.vswhere(legacy=True)
```

Parameters:

- **all_** (Optional, Defaulted to `False`): Finds all instances even if they are incomplete and may not launch.

- **prerelease** (Optional, Defaulted to `False`): Also searches prereleases. By default, only releases are searched.
- **products** (Optional, Defaulted to `None`): List of one or more product IDs to find. Defaults to Community, Professional, and Enterprise. Specify `["*"]` by itself to search all product instances installed.
- **requires** (Optional, Defaulted to `None`): List of one or more workload or component IDs required when finding instances. See https://docs.microsoft.com/en-us/visualstudio/install/workload-and-component-ids?view=vs-2017 listing all workload and component IDs.
- **version** (Optional, Defaulted to `""`): A version range of instances to find. Example: `"[15.0,16.0)"` will find versions 15.*.
- **latest** (Optional, Defaulted to `False`): Return only the newest version and last installed.
- **legacy** (Optional, Defaulted to `False`): Also searches Visual Studio 2015 and older products. Information is limited. This option cannot be used with either `products` or `requires` parameters.
- **property_** (Optional, Defaulted to `""`): The name of a property to return. Use delimiters `.`, `/`, or `_` to separate object and property names. Example: `"properties.nickname"` will return the "nickname" property under "properties".
- **nologo** (Optional, Defaulted to `True`): Do not show logo information.

18.7.44 tools.vs_comntools()

```
def vs_comntools(compiler_version)
```

Returns the value of the environment variable `VS<compiler_version>.0COMNTOOLS` for the compiler version indicated.

```
from conans import tools

vs_path = tools.vs_comntools("14")
```

Parameters:

- **compiler_version** (Required): String with the version number: `"14"`, `"12"`...

18.7.45 tools.vs_installation_path()

```
def vs_installation_path(version, preference=None)
```

Returns the Visual Studio installation path for the given version. It uses *tools.vswhere()* and *tools.vs_comntools()*. It will also look for the installation paths following *CONAN_VS_INSTALLATION_PREFERENCE* environment variable or the preference parameter itself. If the tool is not able to return the path it will return `None`.

```
from conans import tools

vs_path_2017 = tools.vs_installation_path("15", preference=["Community", "BuildTools",
↪"Professional", "Enterprise"])
```

Parameters:

- **version** (Required): Visual Studio version to locate. Valid version numbers are strings: `"10"`, `"11"`, `"12"`, `"13"`, `"14"`, `"15"`...

- **preference** (Optional, Defaulted to `None`): Set to value of *CONAN_VS_INSTALLATION_PREFERENCE* or defaulted to `["Enterprise", "Professional", "Community", "BuildTools"]`. If only set to one type of preference, it will return the installation path only for that Visual type and version, otherwise `None`.

18.7.46 tools.replace_prefix_in_pc_file()

```
def replace_prefix_in_pc_file(pc_file, new_prefix)
```

Replaces the `prefix` variable in a package config file *.pc* with the specified value.

```
from conans import tools

lib_b_path = self.deps_cpp_info["libB"].rootpath
tools.replace_prefix_in_pc_file("libB.pc", lib_b_path)
```

Parameters:

- **pc_file** (Required): Path to the pc file
- **new_prefix** (Required): New prefix variable value (Usually a path pointing to a package).

See also:

Check section *pkg-config and .pc files* to know more.

18.7.47 tools.collect_libs()

```
def collect_libs(conanfile, folder=None)
```

Returns a sorted list of library names from the libraries (files with extensions *.so*, *.lib*, *.a* and *.dylib*) located inside the `conanfile.cpp_info.libdirs` (by default) or the **folder** directory relative to the package folder. Useful to collect not inter-dependent libraries or with complex names like `libmylib-x86-debug-en.lib`.

```
from conans import tools

def package_info(self):
    self.cpp_info.libdirs = ["lib", "other_libdir"]  # Deafult value is 'lib'
    self.cpp_info.libs = tools.collect_libs(self)
```

For UNIX libraries staring with **lib**, like *libmath.a*, this tool will collect the library name **math**.

Parameters:

- **conanfile** (Required): A `ConanFile` object to get the `package_folder` and `cpp_info`.
- **folder** (Optional, Defaulted to `None`): String indicating the subfolder name inside `conanfile.package_folder` where the library files are.

Warning: This tool collects the libraries searching directly inside the package folder and returns them in no specific order. If libraries are inter-dependent, then `package_info()` method should order them to achieve correct linking order.

18.7.48 tools.PkgConfig()

```
class PkgConfig(library, pkg_config_executable="pkg-config", static=False, msvc_
↪syntax=False, variables=None, print_errors=True)
```

Wrapper of the `pkg-config` tool.

```
from conans import tools

with tools.environment_append({'PKG_CONFIG_PATH': tmp_dir}):
    pkg_config = PkgConfig("libastral")
    print(pkg_config.version)
    print(pkg_config.cflags)
    print(pkg_config.cflags_only_I)
    print(pkg_config.variables)
```

Parameters of the constructor:

- **library** (Required): Library (package) name, such as `libastral`.
- **pkg_config_executable** (Optional, Defaulted to `"pkg-config"`): Specify custom pkg-config executable (e.g., for cross-compilation).
- **static** (Optional, Defaulted to `False`): Output libraries suitable for static linking (adds `--static` to `pkg-config` command line).
- **msvc_syntax** (Optional, Defaulted to `False`): MSVC compatibility (adds `--msvc-syntax` to `pkg-config` command line).
- **variables** (Optional, Defaulted to `None`): Dictionary of pkg-config variables (passed as `--define-variable=VARIABLENAME=VARIABLEVALUE`).
- **print_errors** (Optional, Defaulted to `True`): Output error messages (adds –print-errors)

Properties:

PROPERTY	DESCRIPTION
.version	get version defined in the module
.cflags	get all pre-processor and compiler flags
.cflags_only_I	get -I flags
.cflags_only_other	get cflags not covered by the cflags-only-I option
.libs	get all linker flags
.libs_only_L	get -L flags
.libs_only_l	get -l flags
.libs_only_other	get other libs (e.g., -pthread)
.provides	get which packages the package provides
.requires	get which packages the package requires
.requires_private	get packages the package requires for static linking
.variables	get list of variables defined by the module

18.7.49 tools.Git()

Warning: This is an **experimental** feature subject to breaking changes in future releases.

```
class Git(folder=None, verify_ssl=True, username=None, password=None,
          force_english=True, runner=None):
```

Wrapper of the `git` tool.

Parameters of the constructor:

- **folder** (Optional, Defaulted to `None`): Specify a subfolder where the code will be cloned. If not specified it will clone in the current directory.
- **verify_ssl** (Optional, Defaulted to `True`): Verify SSL certificate of the specified **url**.
- **username** (Optional, Defaulted to `None`): When present, it will be used as the login to authenticate with the remote.
- **password** (Optional, Defaulted to `None`): When present, it will be used as the password to authenticate with the remote.
- **force_english** (Optional, Defaulted to `True`): The encoding of the tool will be forced to use `en_US.UTF-8` to ease the output parsing.
- **runner** (Optional, Defaulted to `None`): By default `subprocess.check_output` will be used to invoke the `git` tool.

Methods:

- **version**: (property) Retrieve version from the installed Git client.
- **run(command)**: Run any "git" command, e.g., `run("status")`
- **get_url_with_credentials(url)**: Returns the passed URL but containing the `username` and `password` in the URL to authenticate (only if `username` and `password` is specified)
- **clone(url, branch=None, args="", shallow=False)**: Clone a repository. Optionally you can specify a branch. Note: If you want to clone a repository and the specified **folder** already exist you have to specify a `branch`. Additional `args` may be specified (e.g. git config variables). Use `shallow` to perform a shallow clone (with *–depth 1* - only last revision is being cloned, such clones are usually done faster and take less disk space). In this case, `branch` may specify any valid git reference - e.g. branch name, tag name, sha256 of the revision, expression like *HEAD~1* or *None* (default branch, e.g. *master*).
- **checkout(element, submodule=None)**: Checkout a branch, commit or tag given by `element`. Argument `submodule` can get values in `shallow` or `recursive` to instruct what to do with submodules.
- **get_remote_url(remote_name=None)**: Returns the remote URL of the specified remote. If not `remote_name` is specified `origin` will be used.
- **get_qualified_remote_url()**: Returns the remote url (see `get_remote_url()`) but with forward slashes if it is a local folder.
- **get_revision(), get_commit()**: Gets the current commit hash.
- **get_branch()**: Gets the current branch.
- **get_tag()**: Gets the current checkout tag (**`git describe --exact-match --tags`**) and returns `None` if not in a tag.
- **excluded_files()**: Gets a list of the files and folders that would be excluded by *.gitignore* file.

- **is_local_repository()**: Returns *True* if the remote is a local folder.
- **is_pristine()**: Returns *True* if there aren't modified or uncommitted files in the working copy.
- **get_repo_root()**: Returns the root folder of the working copy.
- **get_commit_message()**: Returns the latest log message

18.7.50 tools.SVN()

Warning: This is an **experimental** feature subject to breaking changes in future releases.

```
class SVN(folder=None, verify_ssl=True, username=None, password=None,
          force_english=True, runner=None):
```

Wrapper of the `svn` tool.

Parameters of the constructor:

- **folder** (Optional, Defaulted to `None`): Specify a subfolder where the code will be cloned. If not specified it will clone in the current directory.
- **verify_ssl** (Optional, Defaulted to `True`): Verify SSL certificate of the specified **url**.
- **username** (Optional, Defaulted to `None`): When present, it will be used as the login to authenticate with the remote.
- **password** (Optional, Defaulted to `None`): When present, it will be used as the password to authenticate with the remote.
- **force_english** (Optional, Defaulted to `True`): The encoding of the tool will be forced to use `en_US.UTF-8` to ease the output parsing.
- **runner** (Optional, Defaulted to `None`): By default `subprocess.check_output` will be used to invoke the `svn` tool.

Methods:

- **version**: (property) Retrieve version from the installed SVN client.
- **run(command)**: Run any "svn" command, e.g., `run("status")`
- **get_url_with_credentials(url)**: Return the passed url but containing the `username` and `password` in the URL to authenticate (only if `username` and `password` is specified)
- **checkout(url, revision="HEAD")**: Checkout the revision number given by `revision` from the specified `url`.
- **update(revision="HEAD")**: Update working copy to revision number given by `revision`.
- **get_remote_url()**: Returns the remote url of working copy.
- **get_qualified_remote_url()**: Returns the remote url of the working copy with the peg revision appended to it.
- **get_revision()**: Gets the current revision number from the repo server.
- **get_last_changed_revision(use_wc_root=True)**: Returns the revision number corresponding to the last changed item in the working folder (`use_wc_root=False`) or in the working copy root (`use_wc_root=True`).

- **get_branch()**: Tries to deduce the branch name from the standard SVN layout. Will raise if cannot resolve it.
- **get_tag()**: Tries to deduce the tag name from the standard SVN layout and returns the current tag name. Otherwise it will return `None`.
- **excluded_files()**: Gets a list of the files and folders that are marked to be ignored.
- **is_local_repository()**: Returns *True* if the remote is a local folder.
- **is_pristine()**: Returns *True* if there aren't modified or uncommitted files in the working copy.
- **get_repo_root()**: Returns the root folder of the working copy.
- **get_revision_message()**: Returns the latest log message

Warning: SVN allows to checkout a subdirectory of the remote repository, take into account that the return value of some of these functions may depend on the root of the working copy that has been checked out.

18.7.51 tools.is_apple_os()

```
def is_apple_os(os_)
```

Returns `True` if OS is an Apple one: macOS, iOS, watchOS or tvOS.

Parameters:

- **os_** (Required): OS to perform the check. Usually this would be `self.settings.os`.

18.7.52 tools.to_apple_arch()

```
def to_apple_arch(arch)
```

Converts Conan style architecture into Apple style architecture.

Parameters:

- **arch** (Required): arch to perform the conversion. Usually this would be `self.settings.arch`.

18.7.53 tools.apple_sdk_name()

```
def apple_sdk_name(settings)
```

Returns proper SDK name suitable for OS and architecture you are building for (considering simulators). If `self.settings.os.sdk` setting is defined, it is used, otherwise the function tries to auto-detect based on `self.settings.os` and `self.settings.arch`.

Parameters:

- **settings** (Required): Conanfile settings.

18.7.54 tools.apple_deployment_target_env()

```
def apple_deployment_target_env(os_, os_version)
```

Environment variable name which controls deployment target: `MACOSX_DEPLOYMENT_TARGET`, `IOS_DEPLOYMENT_TARGET`, `WATCHOS_DEPLOYMENT_TARGET` or `TVOS_DEPLOYMENT_TARGET`.

Parameters:

- **os_** (Required): OS of the settings. Usually `self.settings.os`.
- **os_version** (Required): OS version.

18.7.55 tools.apple_deployment_target_flag()

```
def apple_deployment_target_flag(os_, os_version, os_sdk=None, os_subsystem=None,␣
↪arch=None)
```

Compiler flag name which controls deployment target. For example: `-mappletvos-version-min=9.0`

Parameters:

- **os_** (Required): OS of the settings. Usually `self.settings.os`.
- **os_version** (Required): OS version. Usually `self.settings.os.version`.
- **os_sdk** (Optional, Defaulted to `None`): OS SDK. Usually `self.settings.os.sdk`. Otherwise, check **`xcodebuild -sdk -version`**. for available SDKs.
- **os_subsystem** Optional, Defaulted to `None`): OS sybsystem. Usually `self.settings.os.subsystem`. The only subsystem supported right now is Catalyst.
- **arch** (Optional, Defaulted to `None`): Architecture of the settings. Usually `self.settings.arch`.

18.7.56 tools.XCRun()

```
class XCRun(object):

    def __init__(self, settings, sdk=None):
```

XCRun wrapper used to get information for building.

Properties:

- **sdk_path**: Obtain SDK path (a.k.a. Apple sysroot or -isysroot).
- **sdk_version**: Obtain SDK version.
- **sdk_platform_path**: Obtain SDK platform path.
- **sdk_platform_version**: Obtain SDK platform version.
- **cc**: Path to C compiler (CC).
- **cxx**: Path to C++ compiler (CXX).
- **ar**: Path to archiver (AR).
- **ranlib**: Path to archive indexer (RANLIB).
- **strip**: Path to symbol removal utility (STRIP).

18.7.57 tools.latest_vs_version_installed()

```
def latest_vs_version_installed()
```

Returns a string with the major version of latest Microsoft Visual Studio available on machine. If no Microsoft Visual Studio installed, it returns `None`.

18.7.58 tools.apple_dot_clean()

```
def apple_dot_clean(folder)
```

Remove recursively all `._` files inside `folder`, these files are created by Apple OS when the underlying filesystem cannot store metadata associated to files (they could appear when unzipping a file that has been created in Macos). This tool will remove only the `._` files that are accompanied with a file without that prefix (it will remove `._file.txt` only if `file.txt` exists).

Parameters:

- **folder** (Required): root folder to start deleting `._` files.

18.7.59 tools.Version()

```
from conans import tools

v = tools.Version("1.2.3-dev23")
assert v < "1.2.3"
```

This is a helper class to work with semantic versions, built on top of `semver.SemVer` class with loose parsing. It exposes all the version components as properties and offers total ordering through compare operators.

Build the `tools.Version` object using any valid string or any object that converts to string, the constructor will raise if the string is not a valid loose semver.

Properties:

- **major**: component `major` of semver version
- **minor**: component `minor` of semver version (defaults to `"0"`)
- **patch**: component `patch` of semver version (defaults to `"0"`)
- **prerelease**: component `prerelease` of semver version (defaults to `""`)
- **build**: component `build` of semver version (defaults to `""`). Take into account that `build` component doesn't affect precedence between versions.

18.7.60 tools.to_android_abi()

```
def to_android_abi(arch)
```

Converts Conan style architecture into Android NDK style architecture.

Parameters:

- **arch** (Required): Arch to perform the conversion. Usually this would be `self.settings.arch`.

18.7.61 tools.check_min_cppstd()

```
def check_min_cppstd(conanfile, cppstd, gnu_extensions=False)
```

Validates if the applied cppstd setting (from *compiler.cppstd* settings or deducing the default from *compiler* and *compiler.version*) is at least the value specified in the *cppstd* argument. It raises a `ConanInvalidConfiguration` when is not supported.

```
from conans import tools, ConanFile

class Recipe(ConanFile):
    ...

    def validate(self):
        tools.check_min_cppstd(self, "17")
```

- If the current cppstd does not support C++17, `check_min_cppstd` will raise an `ConanInvalidConfiguration` error.
- If `gnu_extensions` is True, it is required that the applied `cppstd` supports the gnu extensions. (e.g. gnu17), otherwise, an *ConanInvalidConfiguration* will be raised. The `gnu_extensions` is checked in any OS.
- If no compiler has been specified or the compiler is unknown, it raises a `ConanException` exception.

Parameters:

- **conanfile** (Required): ConanFile instance. Usually `self`.
- **cppstd** (Required): C++ standard version which must be supported.
- **gnu_extensions** (Optional): GNU extension is required.

18.7.62 tools.valid_min_cppstd()

```
def valid_min_cppstd(conanfile, cppstd, gnu_extensions=False)
```

Validate the current cppstd from settings or compiler, if it is supported by the required cppstd version. It returns `True` when is valid, otherwise, `False`.

```
from conans import tools, ConanFile

class Recipe(ConanFile):
    ...

    def validate(self):
```

(continues on next page)

(continued from previous page)

```
        if not tools.valid_min_cppstd(self, "17"):
            self.output.error("C++17 is required.")
```

- The `valid_min_cppstd` works exactly like `check_min_cppstd`, however, it does not raise `ConanInvalidConfiguration` error.

Parameters:

- **conanfile** (Required): ConanFile instance. Usually `self`.
- **cppstd** (Required): C++ standard version which must be supported.
- **gnu_extensions** (Optional): GNU extension is required.

18.7.63 tools.cppstd_flag():

```
def cppstd_flag(settings)
```

Returns the corresponding C++ standard flag based on the settings. For instance, it may return `-std=c++17` for `compiler.cppstd=17`, and so on.

Parameters:

- **settings** (Required): Conanfile settings. Use `self.settings`.

18.7.64 tools.msvs_toolset()

```
def msvs_toolset(conanfile)
```

Returns the corresponding Visual Studio platform toolset based on the settings of the given `conanfile`. For instance, it may return `v142` for `compiler=Visual Studio` with `compiler.version=16`. If `compiler.toolset` was set in settings, it has a priority and always returned.

Parameters:

- **conanfile** (Required): ConanFile instance. Usually `self`.

18.7.65 tools.intel_compilervars_command()

Warning: This is an **experimental** feature subject to breaking changes in future releases.

```
def intel_compilervars_command(conanfile, arch=None, compiler_version=None, force=False)
```

Returns, for given settings of the given `conanfile`, the command that should be called to load the Intel C++ environment variables for a certain Intel C++ version. It wraps the functionality of compilervars but does not execute the command, as that typically have to be done in the same command as the compilation, so the variables are loaded for the same subprocess. It will be typically used in the `build()` method, like this:

```
from conans import tools

def build(self):
```

(continues on next page)

(continued from previous page)

```
    cvars_command = tools.intel_compilervars_command(self)
    build_command = ...
    self.run("%s && configure %s" % (cvars_command, " ".join(args)))
    self.run("%s && %s %s" % (cvars, build_command, " ".join(build_args)))
```

The `cvars_command` string will contain something like `call "compilervars.bat"` for the corresponding Intel C++ version for the current settings.

This is typically not needed if using CMake, as the `cmake` generator will handle the correct Intel C++ version.

If **arch** or **compiler_version** is specified, it will ignore the settings and return the command to set the Intel C++ environment for these parameters.

Parameters:

- **conanfile** (Required): ConanFile instance. Usually `self`.
- **arch** (Optional, Defaulted to `None`): Will use `conanfile.settings.arch`.
- **compiler_version** (Optional, Defaulted to `None`): Will use `conanfile.settings.compiler.version`.
- **force** (Optional, Defaulted to `False`): Will ignore if the environment is already set for a different Intel C++ version.

18.7.66 tools.intel_compilervars_dict()

Warning: This is an **experimental** feature subject to breaking changes in future releases.

```
def intel_compilervars_dict(conanfile, arch=None, compiler_version=None, force=False,␣
↪only_diff=True)
```

Returns a dictionary with the variables set by the *tools.intel_compilervars_command()* that can be directly applied to *tools.environment_append()*.

The values of the variables `INCLUDE`, `LIB`, `LIBPATH` and `PATH` will be returned as a list. When used with *tools.environment_append()*, the previous environment values that these variables may have will be appended automatically.

```
from conans import tools

def build(self):
    env_vars = tools.intel_compilervars_dict(self.settings)
    with tools.environment_append(env_vars):
        # Do something
```

Parameters:

- Same as *tools.intel_compilervars_command()*.
- **only_diff** (Optional, Defaulted to `True`): When True, the command will return only the variables set by `intel_compilervars` and not the whole environment. If *intel_compilervars* modifies an environment variable by appending values to the old value (separated by `;`), only the new values will be returned, as a list.

18.7.67 tools.intel_compilervars()

Warning: This is an **experimental** feature subject to breaking changes in future releases.

```
def intel_compilervars(conanfile, arch=None, compiler_version=None, force=False, only_
↪diff=True)
```

This is a context manager that allows to append to the environment all the variables set by the *tools.intel_compilervars_dict()*. You can replace *tools.intel_compilervars_dict()* and use this context manager to get a cleaner way to activate the Intel C++ environment:

```
from conans import tools

def build(self):
    with tools.intel_compilervars(self.settings):
        do_something()
```

18.7.68 tools.intel_installation_path()

Warning: This is an **experimental** feature subject to breaking changes in future releases.

```
def intel_installation_path(version, arch)
```

Returns the Intel Compiler installation path for the given version and target arch. If the tool is not able to return the path it will raise a `ConanException`.

```
from conans import tools

intel_path_2020 = tools.intel_installation_path("19.1", "x86")
```

Parameters:

- **version** (Required): Intel Compiler version to locate. Valid version numbers are strings: `"15"`, `"16"`, `"17"`, `"18"`, `"19"`, `"19.1"`...
- **arch** (Required): Intel Compiler target arch. Valid archs are `"x86"` and `"x86_64"`.

18.7.69 tools.remove_files_by_mask()

```
def remove_files_by_mask(directory, pattern)
```

Removes files recursively in the given `directory` matching the `pattern`. The function removes only files, and never removes directories, even if their names match the pattern. The functions returns the array of the files removed (empty array in case no files were removed). The paths in the returned array are relative to the given `directory`.

Parameters:

- **directory** (Required): Directory to remove files inside. You may use `os.getcwd` or `self.package_folder`, for instance.

- **pattern** (Required): Pattern to check. See fnmatch documentation for more details.

18.7.70 tools.stdcpp_library():

```
def stdcpp_library(conanfile)
```

Returns the corresponding C++ standard library to link with based on the settings of the given conanfile. For instance, it may return `c++` for `compiler.libcxx=libc++`, and it may return `stdc++` for `compiler.libcxx=libstdc++` or `compiler.libcxx=libstdc++11`. Returns `None` if there is no C++ standard library need to be linked. Usually, this is required to populate `self.cpp_info.system_libs` for C++ libraries with plain C API, therefore such libraries might be safely used in pure C projects (or in general, non-C++ projects capable of using C API, such as written in Objective-C, Fortran, etc.).

Parameters:

- **conanfile** (Required): ConanFile instance. Usually `self`.

18.7.71 tools.fix_symlinks():

> **Warning:** This is an **experimental** feature subject to breaking changes in future releases.

```
def fix_symlinks(conanfile, raise_if_error=False)
```

This tool is intended to be used inside the `package()` method after all files have been copied. It takes care of symlinks:

- Converts every symlink into a relative one starting in the root of the package.
- Removes (or raises) symlinks that point to files/directories outside the package.
- Removes (or raises) broken symlinks.

Parameters:

- **conanfile** (Required): ConanFile instance. Usually `self`.
- **raise_if_error** (Optional, Defaulted to `False`): Indicates whether to raise or to remove invalid symlinks.

18.8 Configuration files

These are the most important configuration files, used to customize conan.

18.8.1 artifacts.properties

This is a file in the Conan cache that is useful to define a set of key-value pairs that will be sent together with the packages uploaded in the **`conan upload`** command. This information is sent as custom headers in the PUT request and, if the server has the capability, as matrix params.

.conan/artifacts.properties

```
custom_header1=Value1
custom_header2=45
build.name=BuildJob
```

Artifactory users can benefit from this capability to set file properties for the uploaded files. If the Artifactory version doesn't support matrix params yet (available since Artifactory 7.3.2) it will use the properties from the file that are prefixed with `artifact_property_`:

.conan/artifacts.properties

```
artifact_property_build.name=Build1
artifact_property_build.number=23
artifact_property_build.timestamp=1487676992
artifact_property_custom_multiple_var=one;two;three;four
```

Take into account that some reverse proxies will block headers that contain a period in their name, for example Nginx, as they consider it to be a security issue (you can bypass this check adding the *ignore_invalid_headers* to your Nginx configuration).

18.8.2 client.crt / client.key

Conan support client TLS certificates. Create a *client.crt* with the client certificate in the Conan home directory (default *~/.conan*) and a *client.key* with the private key.

You could also create only the `client.crt` file containing both the certificate and the private key concatenated.

Alternatively, you can define a path to those files in whichever location using the *client_cert_path* and *client_cert_key_path* configuration entries in the *conan.conf* .

18.8.3 conan.conf

The typical location of the **conan.conf** file is the directory `~/.conan/`:

```
[log]
run_to_output = True        # environment CONAN_LOG_RUN_TO_OUTPUT
run_to_file = False         # environment CONAN_LOG_RUN_TO_FILE
level = critical            # environment CONAN_LOGGING_LEVEL
# trace_file =              # environment CONAN_TRACE_FILE
print_run_commands = False  # environment CONAN_PRINT_RUN_COMMANDS

[general]
default_profile = default
compression_level = 9                 # environment CONAN_COMPRESSION_LEVEL
sysrequires_sudo = True               # environment CONAN_SYSREQUIRES_SUDO
request_timeout = 60                  # environment CONAN_REQUEST_TIMEOUT (seconds)
default_package_id_mode = semver_direct_mode # environment CONAN_DEFAULT_PACKAGE_ID_MODE
# parallel_download = 8               # experimental download binaries in parallel
# full_transitive_package_id = 0
# retry = 2                           # environment CONAN_RETRY
# retry_wait = 5                      # environment CONAN_RETRY_WAIT (seconds)
# sysrequires_mode = enabled          # environment CONAN_SYSREQUIRES_MODE (allowed␣
↪modes enabled/verify/disabled)
# vs_installation_preference = Enterprise, Professional, Community, BuildTools #␣
↪environment CONAN_VS_INSTALLATION_PREFERENCE
# verbose_traceback = False           # environment CONAN_VERBOSE_TRACEBACK
# error_on_override = False           # environment CONAN_ERROR_ON_OVERRIDE
# bash_path = ""                      # environment CONAN_BASH_PATH (only windows)
```

(continues on next page)

(continued from previous page)

```
# read_only_cache = True                # environment CONAN_READ_ONLY_CACHE
# cache_no_locks = True                 # environment CONAN_CACHE_NO_LOCKS
# user_home_short = your_path           # environment CONAN_USER_HOME_SHORT
# use_always_short_paths = False        # environment CONAN_USE_ALWAYS_SHORT_PATHS
# skip_vs_projects_upgrade = False      # environment CONAN_SKIP_VS_PROJECTS_UPGRADE
# non_interactive = False               # environment CONAN_NON_INTERACTIVE
# skip_broken_symlinks_check = False    # enviornment CONAN_SKIP_BROKEN_SYMLINKS_CHECK
# revisions_enabled = False             # environment CONAN_REVISIONS_ENABLED

# conan_make_program = make             # environment CONAN_MAKE_PROGRAM (overrides the␣
↪make program used in AutoToolsBuildEnvironment.make)
# conan_cmake_program = cmake           # environment CONAN_CMAKE_PROGRAM (overrides the␣
↪make program used in CMake.cmake_program)

# cmake_generator                       # environment CONAN_CMAKE_GENERATOR
# cmake_generator_platform              # environment CONAN_CMAKE_GENERATOR_PLATFORM
# http://www.vtk.org/Wiki/CMake_Cross_Compiling
# cmake_toolchain_file                  # environment CONAN_CMAKE_TOOLCHAIN_FILE
# cmake_system_name                     # environment CONAN_CMAKE_SYSTEM_NAME
# cmake_system_version                  # environment CONAN_CMAKE_SYSTEM_VERSION
# cmake_system_processor                # environment CONAN_CMAKE_SYSTEM_PROCESSOR
# cmake_find_root_path                  # environment CONAN_CMAKE_FIND_ROOT_PATH
# cmake_find_root_path_mode_program     # environment CONAN_CMAKE_FIND_ROOT_PATH_MODE_
↪PROGRAM
# cmake_find_root_path_mode_library     # environment CONAN_CMAKE_FIND_ROOT_PATH_MODE_
↪LIBRARY
# cmake_find_root_path_mode_include     # environment CONAN_CMAKE_FIND_ROOT_PATH_MODE_
↪INCLUDE

# msbuild_verbosity = minimal           # environment CONAN_MSBUILD_VERBOSITY

# cpu_count = 1             # environment CONAN_CPU_COUNT

# Change the default location for building test packages to a temporary folder
# which is deleted after the test.
# temp_test_folder = True               # environment CONAN_TEMP_TEST_FOLDER

# cacert_path                           # environment CONAN_CACERT_PATH
# scm_to_conandata                      # environment CONAN_SCM_TO_CONANDATA

# config_install_interval = 1h
# required_conan_version = >=1.26

# keep_python_files = False             # environment CONAN_KEEP_PYTHON_FILES

[storage]
# This is the default path, but you can write your own. It must be an absolute path or a
# path beginning with "~" (if the environment var CONAN_USER_HOME is specified, this␣
↪directory, even
# with "~/", will be relative to the conan user home, not to the system user home)
path = ./data
# download_cache = /path/to/my/cache
```

(continues on next page)

(continued from previous page)

```
[proxies]
# Empty (or missing) section will try to use system proxies.
# As documented in https://requests.readthedocs.io/en/master/user/advanced/#proxies -␣
↪but see below
# for proxies to specific hosts
# http = http://user:pass@10.10.1.10:3128/
# http = http://10.10.1.10:3128
# https = http://10.10.1.10:1080
# To specify a proxy for a specific host or hosts, use multiple lines each specifying␣
↪host = proxy-spec
# http =
#   hostname.to.be.proxied.com = http://user:pass@10.10.1.10:3128
# You can skip the proxy for the matching (fnmatch) urls (comma-separated)
# no_proxy_match = *center.conan.io*, https://myserver.*

[hooks]    # environment CONAN_HOOKS
attribute_checker

# Default settings now declared in the default profile
```

Log

The `level` variable, defaulted to 50 (critical events), declares the LOG level . If you want to show more detailed logging information, set this variable to lower values, as 10 to show debug information, or use the level names as `critical`, `error`, `warning`, `info` and `debug`. You can also adjust the environment variable `CONAN_LOGGING_LEVEL`. The level number is related to the Python Logging Levels.

The `print_run_commands`, when is 1, Conan will print the executed commands in `self.run` to the output. You can also adjust the environment variable CONAN_PRINT_RUN_COMMANDS

The `run_to_file` variable, defaulted to False, will print the output from the `self.run` executions to the path that the variable specifies. You can also adjust the environment variable `CONAN_LOG_RUN_TO_FILE`.

The `run_to_output` variable, defaulted to 1, will print to the `stdout` the output from the `self.run` executions in the conanfile. You can also adjust the environment variable `CONAN_LOG_RUN_TO_OUTPUT`.

The `trace_file` variable enable extra logging information about your conan command executions. Set it with an absolute path to a file. You can also adjust the environment variable `CONAN_TRACE_FILE`.

General

The `vs_installation_preference` variable determines the preference of usage when searching a Visual installation. The order of preference by default is Enterprise, Professional, Community and BuildTools. It can be fixed to just one type of installation like only BuildTools. You can also adjust the environment variable `CONAN_VS_INSTALLATION_PREFERENCE`.

The `verbose_traceback` variable will print the complete traceback when an error occurs in a recipe or even in the conan code base, allowing to debug the detected error.

The `error_on_override` turn the messages related to dependencies overriding into errors. When a downstream package overrides some dependency upstream, if this variable is `True` then an error will be raised; to bypass these errors those requirements should be declared explicitly with the `override` keyword.

The `bash_path` variable is used only in windows to help the *tools.run_in_windows_bash()* function to locate our Cygwin/MSYS2 bash. Set it with the bash executable path if it's not in the PATH or you want to use a different one.

The `cache_no_locks` variable is used to disable locking mechanism of local cache. This is primary used for debugging purposes, and in general it's not recommended to disable locks otherwise, as it may result in corrupted packages.

The `default_package_id_mode` changes the way package IDs are computed. By default, if not specified it will be `semver_direct_mode`, but can change to any value defined in *Using package_id() for Package Dependencies*.

The `full_transitive_package_id` changes the way package IDs are computed regarding transitive dependencies. By default, if not specified will be disabled (`0`). Read more about it in *Enabling full transitivity in package_id modes*.

The `parallel_download` configuration defines the number of threads to be used to do parallel downloads of different binaries. This happens when dependencies are installed (`conan install`, `conan create`) and when multiple binaries for the same package are retrieved via `conan download` command. This is an **experimental** feature, subject to change. It is known that the output is still not clean, and will be mangled when using multiple threads. Please report on https://github.com/conan-io/conan/issues about performance gains, and other issues. You might want to try this one in combination with the `storage.download_cache` configuration (see below.)

The `revisions_enabled` variable controls the package revisioning feature. See *Package Revisions* for more info`

The `cmake_***` variables will declare the corresponding CMake variable when you use the *cmake generator* and the *CMake build tool*.

The `msbuild_verbosity` variable is used only by *MSBuild* and *CMake* build helpers. For the *CMake* build helper, it has an effect only for `` `Visual Studio` `` generators. Variable defines verbosity level used by the `` `msbuild` `` tool, as documented on *MSDN <https://docs.microsoft.com/en-us/visualstudio/msbuild/msbuild-command-line-reference?view=vs-2017>*. By default, `` `minimal` `` verbosity level is used, matching the Visual Studio IDE behavior. Allowed values are (in ascending order): `` `quiet` ``, `` `minimal` ``, `` `normal` ``, `` `detailed` ``, `` `diagnostic` ``. You can also adjust the environment variable `CONAN_MSBUILD_VERBOSITY`.

The `conan_make_program` variable used by *CMake* and *AutotoolsBuildEnvironment* build helpers. It overrides a default `` `make` `` executable, might be useful in case you need to use a different make (e.g. BSD Make instead of GNU Make, or MinGW Make). Set it with the make executable path if it's not in the PATH or you want to use a different one.

The `conan_cmake_program` variable used only by *CMake* build helper. It overrides a default `` `cmake` `` executable, might be useful in case you need to use a CMake wrapper tool (such as scan build). Set it with the cmake executable path if it's not in the PATH or you want to use a different one.

The `cpu_count` variable set the number of cores that the *tools.cpu_count()* will return, by default the number of cores available in your machine. Conan recipes can use the `cpu_count()` tool to build the library using more than one core.

The `retry` variable allows to set up the global default value for the number of retries in all commands related to download/upload. User can override the value provided by the variable if the command provides an argument with the same name.

The `retry_wait` variable allows to set up the global default value for the time (in seconds) to wait until the next retry on failures in all commands related to download/upload. User can override the value provided by the variable if the command provides an argument with the same name.

The `sysrequires_mode` variable, defaulted to `enabled` (allowed modes `enabled/verify/disabled`) controls whether system packages should be installed into the system via `SystemPackageTool` helper, typically used in *system_requirements()*. You can also adjust the environment variable `CONAN_SYSREQUIRES_MODE`.

The `sysrequires_sudo` variable, defaulted to True, controls whether `sudo` is used for installing apt, yum, etc. system packages via `SystemPackageTool`. You can also adjust the environment variable `CONAN_SYSREQUIRES_SUDO`.

The `request_timeout` variable, defaulted to 30 seconds, controls the time after Conan will stop waiting for a response. Timeout is not a time limit on the entire response download; rather, an exception is raised if the server has not issued

a response for timeout seconds (more precisely, if no bytes have been received on the underlying socket for timeout seconds). If no timeout is specified explicitly, it do not timeout.

The `user_home_short` specify the base folder to be used with the *short paths* feature. If not specified, the packages marked as *short_paths* will be stored in the `C:\.conan` (or the current drive letter).

If the variable is set to "None" will disable the *short_paths* feature in Windows, for modern Windows that enable long paths at the system level.

Setting this variable equal to, or to a subdirectory of, the local conan cache (e.g. ~/.conan) would result in an invalid cache configuration and is therefore disallowed.

The `verbose_traceback` variable will print the complete traceback when an error occurs in a recipe or even in the conan code base, allowing to debug the detected error.

The `cacert_path` variable lets the user specify a custom path to the *cacert.pem* file to use in requests. You can also adjust this value using the environment variable `CONAN_CACERT_PATH`.

The `client_cert_path` and `client_cert_key_path` variables let the user specify a custom path to the *client.crt / client.key* files.

The `scm_to_conandata` variable tells Conan to store the resolved information of the *SCM feature* in the *conandata.yml* file instead of modifying the recipe file itself. You can also adjust this value using the environment variable `CONAN_SCM_TO_CONANDATA`.

The `skip_broken_symlinks_check` variable (defaulted to `False`) allows the existence broken symlinks while creating a package.

The `config_install_interval` variable starts a time scheduler which runs **`conan config install`** according the time interval configured. It only accepts the follow time intervals: seconds, minutes, hours, days and weeks (e.g 10s, 35m, 48h, 1d, 2w). Empty units or not listed units are not valid, they are considered an error and will be removed automatically on the next execution.

The `required_conan_version` variable validates if the current Conan client version is valid according to its version. When it's not according to the required version or its range, Conan raises an exception before running any command. It accepts SemVer format, including version range. This configuration is useful when a company wants to align the Conan client version used by all teams. This can be also specified at *recipe level* if you need adding this information just for certain recipes.

The `keep_python_files` variable will allow Python *.pyc* files to be packaged. If set to `True`, *.pyc* files will be added to the Conan package, otherwise they will be filtered.

Storage

The `storage.path` variable define the path where all the packages will be stored. By default it is *./data*, which is relative to the folder containing this *conan.conf* file, which by default is the *<userhome>/.conan* folder. It can start with "~", and that will be expanded to the current user home folder. If the environment var CONAN_USER_HOME is specified, the "~" will be replaced by the current Conan home (the folder pointed by the CONAN_USER_HOME environment variable).

On Windows:

- It is recommended to assign it to some unit, e.g. map it to X: in order to avoid hitting the 260 chars path name length limit).
- Also see the *short_paths docs* to know more about how to mitigate the limitation of 260 chars path name length limit.
- It is recommended to disable the Windows indexer or exclude the storage path to avoid problems (busy resources).

Note: If you want to change the default "conan home" (directory where `conan.conf` file is) you can adjust the environment variable `CONAN_USER_HOME`.

The `storage.download_cache` variable defines the path to a folder that can be used to cache the different file downloads from Conan servers but also from user downloads via the `tools.get()` and `tools.download()` methods that provide a checksum. Defining this variable will both configure the path and activate the download cache. If it is not defined, the download cache will not be used.

Read more about the *download cache here*.

Proxies

Warning: `no_proxy` is deprecated in favor of `no_proxy_match` since Conan 1.16.

If you leave the `[proxies]` section blank or delete the section, conan will copy the system configured proxies, but if you configured some exclusion rule it won't work:

```
[proxies]
# Empty (or missing) section will try to use system proxies.
```

You can specify http and https proxies as follows. Use the *no_proxy_match* keyword to specify a list of URLs or patterns that will skip the proxy:

```
[proxies]
# As documented in https://requests.readthedocs.io/en/master/user/advanced/#proxies
http: http://user:pass@10.10.1.10:3128/
http: http://10.10.1.10:3128
https: http://10.10.1.10:1080
http: http://10.10.2.10
    hostname1.to.be.proxied.com = http://user:pass@10.10.3.10
    hostname2.to.be.proxied.com = http://user:pass@10.10.4.10
no_proxy_match: http://url1, http://url2, https://url3*, https://*.custom_domain.*
```

Use *http=None* and/or *https=None* to disable the usage of a proxy.

To nominate a proxy for a specific scheme and host only, add *host.to.proxy=* in front of the url of the proxy (the *host.to.proxy* name must exactly match the host name that should be proxied). You can list several *host name = proxy* pairs on separate indented lines.

You can still specify a default proxy, without a host, which will be used if none of the host names match. If you do not, then the proxy is disabled for non-matching hosts.

If this fails, you might also try to set environment variables:

```
# linux/osx
$ export HTTP_PROXY="http://10.10.1.10:3128"
$ export HTTPS_PROXY="http://10.10.1.10:1080"

# with user/password
$ export HTTP_PROXY="http://user:pass@10.10.1.10:3128/"
$ export HTTPS_PROXY="http://user:pass@10.10.1.10:3128/"
```

(continues on next page)

(continued from previous page)

```
# windows (note, no quotes here)
$ set HTTP_PROXY=http://10.10.1.10:3128
$ set HTTPS_PROXY=http://10.10.1.10:1080
```

18.8.4 global.conf

Warning: This new configuration mechanism is an **experimental** feature subject to breaking changes in future releases.

The **global.conf** file is located in the Conan user home directory.

Global configuration

- `core:required_conan_version = expression` allows defining a version expression like `>=1.30`. Conan will raise an error if its current version does not satisfy the condition
- `core.package_id:msvc_visual_incompatible` allows opting-out the fallback from the new `msvc` compiler to the `Visual Studio` compiler existing binaries
- `core:default_profile` defines the default host profile ('default' by default)
- `core:default_build_profile` defines the default build profile (None by default)

Tools configurations

Tools and user configurations allows them to be defined both in the *global.conf* file and in profile files. Profile values will have priority over globally defined ones in *global.conf*, and can be defined as:

```
[settings]
...

[conf]
tools.microsoft.msbuild:verbosity=Diagnostic
tools.microsoft.msbuild:max_cpu_count=2
tools.microsoft.msbuild:vs_version = 16
tools.build:jobs=10
```

To list all possible configurations available, run **conan config list**.

```
$ conan config list
core:required_conan_version: Raise if current version does not match the defined range.
core.package_id:msvc_visual_incompatible: Allows opting-out the fallback from the new␣
↪msvc compiler to the Visual Studio compiler existing binaries
core:default_profile: Defines the default host profile ('default' by default)
core:default_build_profile: Defines the default build profile (None by default)
tools.android:ndk_path: Argument for the CMAKE_ANDROID_NDK
tools.build:skip_test: Do not execute CMake.test() and Meson.test() when enabled
tools.build:jobs: Default compile jobs number -jX Ninja, Make, /MP VS (default: max CPUs)
tools.build:sysroot: Pass the --sysroot=<tools.build:sysroot> flag if available. (None␣
```

(continues on next page)

(continued from previous page)

```
↪by default)
tools.cmake.cmaketoolchain:generator: User defined CMake generator to use instead of␣
↪default
tools.cmake.cmaketoolchain:find_package_prefer_config: Argument for the CMAKE_FIND_
↪PACKAGE_PREFER_CONFIG
tools.cmake.cmaketoolchain:toolchain_file: Use other existing file rather than conan_
↪toolchain.cmake one
tools.cmake.cmaketoolchain:user_toolchain: Inject existing user toolchains at the␣
↪beginning of conan_toolchain.cmake
tools.cmake.cmaketoolchain:system_name: Define CMAKE_SYSTEM_NAME in CMakeToolchain
tools.cmake.cmaketoolchain:system_version: Define CMAKE_SYSTEM_VERSION in CMakeToolchain
tools.cmake.cmaketoolchain:system_processor: Define CMAKE_SYSTEM_PROCESSOR in␣
↪CMakeToolchain
tools.env.virtualenv:auto_use: Automatically activate virtualenv file generation
tools.cmake.cmake_layout.build_folder_vars: Settings and Options that will produce a␣
↪different build folder and different CMake presets names
tools.files.download:retry: Number of retries in case of failure when downloading
tools.files.download:retry_wait: Seconds to wait between download attempts
tools.gnu:make_program: Indicate path to make program
tools.gnu:define_libcxx11_abi: Force definition of GLIBCXX_USE_CXX11_ABI=1 for␣
↪libstdc++11
tools.google.bazel:configs: Define Bazel config file
tools.google.bazel:bazelrc_path: Defines Bazel rc-path
tools.microsoft.msbuild:verbosity: Verbosity level for MSBuild: 'Quiet', 'Minimal',
↪'Normal', 'Detailed', 'Diagnostic'
tools.microsoft.msbuild:vs_version: Defines the IDE version when using the new msvc␣
↪compiler
tools.microsoft.msbuild:max_cpu_count: Argument for the /m when running msvc to build␣
↪parallel projects
tools.microsoft.msbuild:installation_path: VS install path, to avoid auto-detect via␣
↪vswhere, like C:/Program Files (x86)/Microsoft Visual Studio/2019/Community
tools.microsoft.msbuilddeps:exclude_code_analysis: Suppress MSBuild code analysis for␣
↪patterns
tools.microsoft.msbuildtoolchain:compile_options: Dictionary with MSBuild compiler␣
↪options
tools.intel:installation_path: Defines the Intel oneAPI installation root path
tools.intel:setvars_args: Custom arguments to be passed onto the setvars.sh|bat script␣
↪from Intel oneAPI
tools.system.package_manager:tool: Default package manager tool: 'apt-get', 'yum', 'dnf',
↪ 'brew', 'pacman', 'choco', 'zypper', 'pkg' or 'pkgutil'
tools.system.package_manager:mode: Mode for package_manager tools: 'check' or 'install'
tools.system.package_manager:sudo: Use 'sudo' when invoking the package manager tools in␣
↪Linux (False by default)
tools.system.package_manager:sudo_askpass: Use the '-A' argument if using sudo in Linux␣
↪to invoke the system package manager (False by default)
tools.apple.xcodebuild:verbosity: Verbosity level for xcodebuild: 'verbose' or 'quiet
tools.apple:enable_bitcode: (boolean) Enable/Disable Bitcode Apple Clang flags
tools.apple:enable_arc: (boolean) Enable/Disable ARC Apple Clang flags
tools.apple:enable_visibility: (boolean) Enable/Disable Visibility Apple Clang flags
tools.build:cxxflags: List of extra CXX flags used by different toolchains like␣
↪CMakeToolchain, AutotoolsToolchain and MesonToolchain
tools.build:cflags: List of extra C flags used by different toolchains like␣
```

(continues on next page)

(continued from previous page)

```
↪CMakeToolchain, AutotoolsToolchain and MesonToolchain
tools.build:defines: List of extra definition flags used by different toolchains like␣
↪CMakeToolchain and AutotoolsToolchain
tools.build:sharedlinkflags: List of extra flags used by CMakeToolchain for CMAKE_SHARED_
↪LINKER_FLAGS_INIT variable
tools.build:exelinkflags: List of extra flags used by CMakeToolchain for CMAKE_EXE_
↪LINKER_FLAGS_INIT variable
```

Configuration file template

Available since: 1.46.0

It is possible to use **jinja2** template engine for *global.conf*. When Conan loads this file, immediately parses and renders the template, which must result in a standard tools-configuration text.

```
# Using all the cores automatically
tools.build:jobs={{os.cpu_count()}}
# Using the current OS
user.myconf.system:name = {{platform.system()}}
```

Note: The Python packages passed to render the template are only `os` and `platform`.

Configuration data types

Available since: 1.46.0

All the values will be interpreted by Conan as the result of the python built-in *eval()* function:

```
# String
tools.microsoft.msbuild:verbosity=Diagnostic
# Boolean
tools.system.package_manager:sudo=True
# Integer
tools.microsoft.msbuild:max_cpu_count=2
# List of values
user.myconf.build:ldflags=["--flag1", "--flag2"]
# Dictionary
tools.microsoft.msbuildtoolchain:compile_options={"ExceptionHandling": "Async"}
```

Configuration data operators

Available since: 1.46.0

It's also possible to use some extra operators when you're composing tool configurations in your *global.conf* or any of your profiles:

- `+=` == `append`: appends values at the end of the existing value (only for lists).
- `=+` == `prepend`: puts values at the beginning of the existing value (only for lists).
- `=!` == `unset`: gets rid of any configuration value.

Listing 68: *myprofile*

```
[settings]
...

[conf]
# Define the value => ["-f1"]
user.myconf.build:flags=["-f1"]

# Append the value ["-f2"] => ["-f1", "-f2"]
user.myconf.build:flags+=["-f2"]

# Prepend the value ["-f0"] => ["-f0", "-f1", "-f2"]
user.myconf.build:flags=+["-f0"]

# Unset the value
user.myconf.build:flags=!
```

Configuration in your profiles

Let's see a little bit more complex example trying different configurations coming from the *global.conf* and a simple profile:

Listing 69: *global.conf*

```
# Defining several lists
user.myconf.build:ldflags=["--flag1 value1"]
user.myconf.build:cflags=["--flag1 value1"]
```

Listing 70: *myprofile*

```
[settings]
...

[conf]
# Appending values into the existing list
user.myconf.build:ldflags+=["--flag2 value2"]

# Unsetting the existing value (it'd be like we define it as an empty value)
user.myconf.build:cflags=!

# Prepending values into the existing list
user.myconf.build:ldflags=+["--prefix prefix-value"]
```

Running, for instance, **conan install . -pr myprofile**, the configuration output will be something like:

```
...
Configuration:
[settings]
[options]
[build_requires]
[env]
```

(continues on next page)

(continued from previous page)

```
[conf]
user.myconf.build:cflags=!
user.myconf.build:ldflags=['--prefix prefix-value', '--flag1 value1', '--flag2 value2']
...
```

Configuration in your recipes

From Conan 1.46, the user interface to manage the configurations in your recipes has been improved. The `self.conf_info` object has the following methods available:

- `get(name, default=None, check_type=None)`: gets the value for the given configuration name. Besides that you can pass `check_type` to check the Python type matches with the value type returned, e.g., `check_type=list`. If the configuration does not exist, `default` will be returned instead. Notice that this `default` value won't be affected by the `check_type=list` param.
- `pop(name, default=None)`: removes (if exists) the configuration name given. If the configuration does not exist, `default` will be returned instead.
- `define(name, value)`: sets `value` for the given configuration name. If it already exists, the configuration will be overwritten with the new value.
- `append(name, value)`: (only available for `list`) appends `value` into the existing list for the given configuration name. If the list does not exist yet, it'll be created with the value given by default. `value` can be a list or a single value.
- `prepend(name, value)`: (only available for `list`) prepends `value` into the existing list for the given configuration name. If the list does not exist yet, it'll be created with the value given by default. `value` can be a list or a single value.
- `update(name, value)`: (only available for `dict`) updates the existing dictionary with `value` for the given configuration name. If the dict does not exist yet, it'll be created with the value given by default. `value` must be another dictionary.
- `remove(name, value)`: (only available for `dict` and `list`) removes `value` from the existing value for the given configuration name.
- `unset(name)`: removes any existing value for the given configuration name. It's behaving like using `define(name, None)`.

This example illustrates all of these methods:

```
import os
from conans import ConanFile

class Pkg(ConanFile):
    name = "pkg"

    def package_info(self):
        # Setting values
        self.conf_info.define("tools.microsoft.msbuild:verbosity", "Diagnostic")
        self.conf_info.define("tools.system.package_manager:sudo", True)
        self.conf_info.define("tools.microsoft.msbuild:max_cpu_count", 2)
        self.conf_info.define("user.myconf.build:ldflags", ["--flag1", "--flag2"])
        self.conf_info.define("tools.microsoft.msbuildtoolchain:compile_options", {
↪"ExceptionHandling": "Async"})
```

(continues on next page)

(continued from previous page)

```
        # Getting values
        self.conf_info.get("tools.microsoft.msbuild:verbosity")  # == "Diagnostic"
        # Getting default values from configurations that don't exist yet
        self.conf_info.get("user.myotherconf.build:cxxflags", default=["--flag3"])  # ==␣
↪["--flag3"]
        # Getting values and ensuring the gotten type is the passed one otherwise an␣
↪exception will be raised
        self.conf_info.get("tools.system.package_manager:sudo", check_type=bool)  # ==␣
↪True
        self.conf_info.get("tools.system.package_manager:sudo", check_type=int)  # ERROR!
↪ It raises a ConanException
        # Modifying configuration list-like values
        self.conf_info.append("user.myconf.build:ldflags", "--flag3")  # == ["--flag1",
↪"--flag2", "--flag3"]
        self.conf_info.prepend("user.myconf.build:ldflags", "--flag0")  # == ["--flag0",
↪"--flag1", "--flag2", "--flag3"]
        # Modifying configuration dict-like values
        self.conf_info.update("tools.microsoft.msbuildtoolchain:compile_options", {
↪"ExpandAttributedSource": "false"})
        # Unset any value
        self.conf_info.unset("tools.microsoft.msbuildtoolchain:compile_options")
        # Remove
        self.conf_info.remove("user.myconf.build:ldflags", "--flag1")  # == ["--flag0",
↪"--flag2", "--flag3"]
        # Removing completely the configuration
        self.conf_info.pop("tools.system.package_manager:sudo")
```

Important: Legacy configuration methods to set/get values like `self.conf_info["xxxxx"] = "yyyyy"` and `v = self.conf_info["xxxxx"]` are deprecated since Conan 1.46 version. Use `self.conf_info.define("xxxxx", "yyyyy")` and `v = self.conf_info.get("xxxxx")` instead like the example above.

Configuration from tool_requires

From Conan 1.37, it is possible to define configuration in packages that are `tool_requires`. For example, assuming there is a package that bundles the AndroidNDK, it could define the location of such NDK to the `tools.android:ndk_path` configuration as:

```
import os
from conans import ConanFile

class Pkg(ConanFile):
    name = "android_ndk"

    def package_info(self):
        self.conf_info.define("tools.android:ndk_path", os.path.join(self.package_folder,
↪ "ndk"))
```

Note that this only propagates from the immediate, direct `tool_requires` of a recipe.

18.8.5 conandata.yml

This YAML file can be used to declare specific information to be used inside the recipe. This file is specific to each recipe *conanfile.py* and it should be placed next to it. The file is automatically exported with the recipe (no need to add it to *exports* attribute) and its content is loaded into the *conan_data* attribute of the recipe.

This file can be used, for example, to declare a list of sources links and checksums for the recipe or a list patches to apply to them, but you can use it to store any data you want to extract from the recipe. For example:

```
sources:
  "1.70.0":
    url: "https://boostorg.jfrog.io/artifactory/main/release/1.70.0/source/boost_1_70_0.
↪tar.bz2"
    sha256: "430ae8354789de4fd19ee52f3b1f739e1fba576f0aded0897c3c2bc00fb38778"
  "1.71.0":
    url: "https://boostorg.jfrog.io/artifactory/main/release/1.70.0/source/boost_1_71_0.
↪tar.bz2"
    sha256: "d73a8da01e8bf8c7eda40b4c84915071a8c8a0df4a6734537ddde4a8580524ee"
patches:
  "1.70.0":
    patches: "0001-beast-fix-moved-from-executor.patch,bcp_namespace_issues.patch"
  "1.71.0":
    patches: "bcp_namespace_issues.patch,boost_build_qcc_fix_debug_build_parameter.patch"
requirements:
  - "foo/1.0"
  - "bar/1.0"
```

Usages in a *conanfile.py*:

```
def source(self):
    tools.get(**self.conan_data["sources"][self.version])
    for patch in self.conan_data["patches"][self.version]:
        tools.patch(**patch)

def requirements(self):
    for req in self.conan_data["requirements"]:
        self.requires(req)
```

Warning: Use always quotes around versions numbers, otherwise YAML parser could interpret those values as integers or floats and lead to unexpected effects when comparing them against the recipe version inside the recipes.

Note: The first level entry key `.conan` is reserved for Conan usage.

18.8.6 profiles/default

This is the typical *~/.conan/profiles/default* file:

```
[tool_requires]
[settings]
    os=Macos
    arch=x86_64
    compiler=apple-clang
    compiler.version=8.1
    compiler.libcxx=libc++
    build_type=Release
[options]
[env]
```

The settings defaults are the setting values used whenever you issue a **conan install** command over a *conanfile* in one of your projects. The initial values for these default settings are auto-detected the first time you run a **conan** command.

You can override the default settings using the **-s** parameter in **conan install** and **conan info** commands but when you specify a profile, **conan install --profile gcc48**, the default profile won't be applied, unless you specify it with an `include()` statement:

Listing 71: my_clang_profile

```
include(default)

[settings]
compiler=clang
compiler.version=3.5
compiler.libcxx=libstdc++11

[env]
CC=/usr/bin/clang
CXX=/usr/bin/clang++
```

Tip: Default profile can be overridden using the environment variable `CONAN_DEFAULT_PROFILE_PATH`.

See also:

Check the section *Profiles* to read more about this feature.

18.8.7 Editable layout files

This file contain information consumed by *editable packages*. It is an *.ini* file listing the directories that Conan should use for the packages that are opened in editable mode. Before parsing this file Conan runs Jinja2 template engine with the `settings`, `options` and `reference` objects, so you can add *any* logic to this files:

```
# Affects to all packages but cool/version@user/dev
[includedirs]
src/include

# using placeholders from conan settings and options
```

(continues on next page)

(continued from previous page)

```
[libdirs]
build/{{settings.build_type}}/{{settings.arch}}

[bindirs]
{% if options.shared %}
build/{{settings.build_type}}/shared
{% else %}
build/{{settings.build_type}}/static
{% endif %}

# Affects only to cool/version@user/dev
[cool/version@user/dev:includedirs]
src/core/include
src/cmp_a/include

# The source_folder, build_folder are useful for workspaces
[source_folder]
src

[build_folder]
build/{{settings.build_type}}/{{settings.arch}}
```

The specific sections using a package reference will have higher priority than the general ones.

This file can live in the conan cache, in the `.conan/layouts` folder, or in a user folder, like inside the source repo.

If there exists a `.conan/layouts/default` layout file in the cache and no layout file is specified in the **conan editable add <path> <reference>** command, that file will be used.

The `[source_folder]` and `[build_folder]` are useful for workspaces. For example, when using `cmake` workspace-generator, it will locate the `CMakeLists.txt` of each package in editable mode in the `[source_folder]` and it will use the `[build_folder]` as the base folder for the build temporary files.

It is possible to define out-of-source builds for workspaces, using relative paths and the `reference` argument. The following could be used to locate the build artifacts of an editable package in a sibling `build/<package-name>` folder:

```
[build_folder]
../build/{{reference.name}}/{{settings.build_type}}

[includedirs]
src

[libdirs]
../build/{{reference.name}}/{{settings.build_type}}/lib
```

See also:

Check the section *Packages in editable mode* and *Workspaces* to learn more about this file.

18.8.8 settings.yml

The input `settings` for packages in Conan are predefined in `~/.conan/settings.yml` file, so only a few like `os` or `compiler` are possible. These are the **default** values, but it is possible to customize them, see *Customizing settings*.

```
# Only for cross building, 'os_build/arch_build' is the system that runs Conan
os_build: [Windows, WindowsStore, Linux, Macos, FreeBSD, SunOS, AIX, VxWorks]
arch_build: [x86, x86_64, ppc32be, ppc32, ppc64le, ppc64, armv5el, armv5hf, armv6, armv7,
 ↪ armv7hf, armv7s, armv7k, armv8, armv8_32, armv8.3, sparc, sparcv9, mips, mips64, avr,␣
 ↪s390, s390x, sh4le, e2k-v2, e2k-v3, e2k-v4, e2k-v5, e2k-v6, e2k-v7]

# Only for building cross compilation tools, 'os_target/arch_target' is the system for
# which the tools generate code
os_target: [Windows, Linux, Macos, Android, iOS, watchOS, tvOS, FreeBSD, SunOS, AIX,␣
 ↪Arduino, Neutrino]
arch_target: [x86, x86_64, ppc32be, ppc32, ppc64le, ppc64, armv5el, armv5hf, armv6,␣
 ↪armv7, armv7hf, armv7s, armv7k, armv8, armv8_32, armv8.3, sparc, sparcv9, mips, mips64,
 ↪ avr, s390, s390x, asm.js, wasm, sh4le, e2k-v2, e2k-v3, e2k-v4, e2k-v5, e2k-v6, e2k-v7,
 ↪ xtensalx6, xtensalx106]

# Rest of the settings are "host" settings:
# - For native building/cross building: Where the library/program will run.
# - For building cross compilation tools: Where the cross compiler will run.
os:
    Windows:
        subsystem: [None, cygwin, msys, msys2, wsl]
    WindowsStore:
        version: ["8.1", "10.0"]
    WindowsCE:
        platform: ANY
        version: ["5.0", "6.0", "7.0", "8.0"]
    Linux:
    Macos:
        version: [None, "10.6", "10.7", "10.8", "10.9", "10.10", "10.11", "10.12", "10.13
 ↪", "10.14", "10.15", "11.0", "12.0", "13.0"]
        sdk: [None, "macosx"]
        sdk_version: [None, "10.13", "10.14", "10.15", "11.0", "11.1", "11.3", "12.0",
 ↪"12.1"]
        subsystem: [None, catalyst]
    Android:
        api_level: ANY
    iOS:
        version: ["7.0", "7.1", "8.0", "8.1", "8.2", "8.3", "9.0", "9.1", "9.2", "9.3",
 ↪"10.0", "10.1", "10.2", "10.3",
                  "11.0", "11.1", "11.2", "11.3", "11.4", "12.0", "12.1", "12.2", "12.3",
 ↪"12.4",
                  "13.0", "13.1", "13.2", "13.3", "13.4", "13.5", "13.6", "13.7",
                  "14.0", "14.1", "14.2", "14.3", "14.4", "14.5", "14.6", "14.7", "14.8",
 ↪"15.0", "15.1"]
        sdk: [None, "iphoneos", "iphonesimulator"]
        sdk_version: [None, "11.3", "11.4", "12.0", "12.1", "12.2", "12.4",
                      "13.0", "13.1", "13.2", "13.4", "13.5", "13.6", "13.7",
                      "14.0", "14.1", "14.2", "14.3", "14.4", "14.5", "15.0", "15.2"]
```

(continues on next page)

(continued from previous page)

```
    watchOS:
        version: ["4.0", "4.1", "4.2", "4.3", "5.0", "5.1", "5.2", "5.3", "6.0", "6.1",
↪"6.2",
                 "7.0", "7.1", "7.2", "7.3", "7.4", "7.5", "7.6", "8.0", "8.1"]
        sdk: [None, "watchos", "watchsimulator"]
        sdk_version: [None, "4.3", "5.0", "5.1", "5.2", "5.3", "6.0", "6.1", "6.2",
                      "7.0", "7.1", "7.2", "7.4", "8.0", "8.0.1", "8.3"]
    tvOS:
        version: ["11.0", "11.1", "11.2", "11.3", "11.4", "12.0", "12.1", "12.2", "12.3",
↪ "12.4",
                 "13.0", "13.2", "13.3", "13.4", "14.0", "14.2", "14.3", "14.4", "14.5",
↪"14.6", "14.7",
                 "15.0", "15.1"]
        sdk: [None, "appletvos", "appletvsimulator"]
        sdk_version: [None, "11.3", "11.4", "12.0", "12.1", "12.2", "12.4",
                      "13.0", "13.1", "13.2", "13.4", "14.0", "14.2", "14.3", "14.5", "15.0
↪", "15.2"]
    FreeBSD:
    SunOS:
    AIX:
    Arduino:
        board: ANY
    Emscripten:
    Neutrino:
        version: ["6.4", "6.5", "6.6", "7.0", "7.1"]
    baremetal:
    VxWorks:
        version: ["7"]
arch: [x86, x86_64, ppc32be, ppc32, ppc64le, ppc64, armv4, armv4i, armv5el, armv5hf,␣
↪armv6, armv7, armv7hf, armv7s, armv7k, armv8, armv8_32, armv8.3, sparc, sparcv9, mips,␣
↪mips64, avr, s390, s390x, asm.js, wasm, sh4le, e2k-v2, e2k-v3, e2k-v4, e2k-v5, e2k-v6,␣
↪e2k-v7, xtensalx6, xtensalx106]
compiler:
    sun-cc:
        version: ["5.10", "5.11", "5.12", "5.13", "5.14", "5.15"]
        threads: [None, posix]
        libcxx: [libCstd, libstdcxx, libstlport, libstdc++]
    gcc: &gcc
        version: ["4.1", "4.4", "4.5", "4.6", "4.7", "4.8", "4.9",
                "5", "5.1", "5.2", "5.3", "5.4", "5.5",
                "6", "6.1", "6.2", "6.3", "6.4", "6.5",
                "7", "7.1", "7.2", "7.3", "7.4", "7.5",
                "8", "8.1", "8.2", "8.3", "8.4",
                "9", "9.1", "9.2", "9.3",
                "10", "10.1", "10.2", "10.3",
                "11", "11.1", "11.2",
                "12"]
        libcxx: [libstdc++, libstdc++11]
        threads: [None, posix, win32]  # Windows MinGW
        exception: [None, dwarf2, sjlj, seh]  # Windows MinGW
        cppstd: [None, 98, gnu98, 11, gnu11, 14, gnu14, 17, gnu17, 20, gnu20, 23, gnu23]
    Visual Studio: &visual_studio
```

(continues on next page)

(continued from previous page)

```
        runtime: [MD, MT, MTd, MDd]
        version: ["8", "9", "10", "11", "12", "14", "15", "16", "17"]
        toolset: [None, v90, v100, v110, v110_xp, v120, v120_xp,
                  v140, v140_xp, v140_clang_c2, LLVM-vs2012, LLVM-vs2012_xp,
                  LLVM-vs2013, LLVM-vs2013_xp, LLVM-vs2014, LLVM-vs2014_xp,
                  LLVM-vs2017, LLVM-vs2017_xp, v141, v141_xp, v141_clang_c2, v142,
                  llvm, ClangCL, v143]
        cppstd: [None, 14, 17, 20, 23]
    msvc:
        version: [190, 191, 192, 193]
        update: [None, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
        runtime: [static, dynamic]
        runtime_type: [Debug, Release]
        cppstd: [14, 17, 20, 23]
    clang:
        version: ["3.3", "3.4", "3.5", "3.6", "3.7", "3.8", "3.9", "4.0",
                  "5.0", "6.0", "7.0", "7.1",
                  "8", "9", "10", "11", "12", "13", "14"]
        libcxx: [None, libstdc++, libstdc++11, libc++, c++_shared, c++_static]
        cppstd: [None, 98, gnu98, 11, gnu11, 14, gnu14, 17, gnu17, 20, gnu20, 23, gnu23]
        runtime: [None, MD, MT, MTd, MDd]
    apple-clang: &apple_clang
        version: ["5.0", "5.1", "6.0", "6.1", "7.0", "7.3", "8.0", "8.1", "9.0", "9.1",
↪"10.0", "11.0", "12.0", "13.0"]
        libcxx: [libstdc++, libc++]
        cppstd: [None, 98, gnu98, 11, gnu11, 14, gnu14, 17, gnu17, 20, gnu20]
    intel:
        version: ["11", "12", "13", "14", "15", "16", "17", "18", "19", "19.1"]
        update: [None, ANY]
        base:
            gcc:
                <<: *gcc
                threads: [None]
                exception: [None]
            Visual Studio:
                <<: *visual_studio
            apple-clang:
                <<: *apple_clang
    intel-cc:
        version: ["2021.1", "2021.2", "2021.3"]
        update: [None, ANY]
        mode: ["icx", "classic", "dpcpp"]
        libcxx: [None, libstdc++, libstdc++11, libc++]
        cppstd: [None, 98, gnu98, 03, gnu03, 11, gnu11, 14, gnu14, 17, gnu17, 20, gnu20,␣
↪23, gnu23]
        runtime: [None, static, dynamic]
        runtime_type: [None, Debug, Release]
    qcc:
        version: ["4.4", "5.4", "8.3"]
        libcxx: [cxx, gpp, cpp, cpp-ne, accp, acpp-ne, ecpp, ecpp-ne]
        cppstd: [None, 98, gnu98, 11, gnu11, 14, gnu14, 17, gnu17]
    mcst-lcc:
```

(continues on next page)

(continued from previous page)

```
        version: ["1.19", "1.20", "1.21", "1.22", "1.23", "1.24", "1.25"]
        base:
            gcc:
                <<: *gcc
                threads: [None]
                exceptions: [None]

build_type: [None, Debug, Release, RelWithDebInfo, MinSizeRel]


cppstd: [None, 98, gnu98, 11, gnu11, 14, gnu14, 17, gnu17, 20, gnu20, 23, gnu23]  #␣
↪Deprecated, use compiler.cppstd
```

As you can see, the possible values `settings` can take are restricted in the same file. This is done to ensure matching naming and spelling as well as defining a common settings model among users and the OSS community. If a setting is allowed to be set to any value, you can use `ANY`. If a setting is allowed to be set to any value or it can also be unset, you can use `[None, ANY]`.

However, this configuration file can be modified to any needs, including new settings or subsettings and their values. If you want to distribute a unified *settings.yml* file you can use the *conan config install command*.

Note: The *settings.yml* file is not perfect nor definitive and surely incomplete. Please share any suggestion in the Conan issue tracker with any missing settings and values that could make sense for other users.

To force the creation of the *settings.yml* the command `conan config init` is available.

Operating systems

`baremetal` operating system (introduced in Conan 1.43) is a convention meaning that the binaries run directly on the hardware, without a operating system or equivalent layer. This is to differentiate to the `None` value, which is associated to the "this value is not defined" semantics. The `baremetal` is a common name convention for embedded microprocessors and microcontrollers code. It is expected that users might customize the space inside the `baremetal` setting with further subsettings to specify their specific hardware platforms, boards, families, etc. At the moment (Conan 1.43) the `os=baremetal` value is still not used by Conan builtin toolchains and helpers, but it is expected that they can evolve and start using it.

Compilers

Some notes about different compilers:

msvc

The new `msvc` compiler is a new, **experimental** one, that is intended to deprecate the `Visual Studio` one in Conan 2.0:

- It uses the compiler version, that is 190 (19.0), 191 (19.1), etc, instead of the Visual Studio IDE (15, 16, etc).
- It is only used by the new build integrations in *conan.tools.cmake* and *conan.tools.microsoft*, but not the previous ones.

- At the moment it implements a `compatible_packages` fallback to Visual Studio compiled packages, that is, previous existing binaries compiled with `settings.compiler="Visual Studio"` can be used for the `msvc` compiler if no binaries exist for it yet. This behavior can be opted-out with `core.package_id:msvc_visual_incompatible` *global.conf* configuration.

When using the `msvc` compiler, the Visual Studio toolset version (the actual `vcvars` activation and `MSBuild` location) will be defined by the default provide of that compiler version:

- `msvc` compiler version '190': Visual Studio 14 2015
- `msvc` compiler version '191': Visual Studio 15 2017
- `msvc` compiler version '192': Visual Studio 16 2019

This can be configured in your profiles with the `tools.microsoft.msbuild:vs_version` configuration:

```
[settings]
compiler=msvc
compiler.version=190

[conf]
tools.microsoft.msbuild:vs_version = 16
```

In this case, the `vcvars` will activate the Visual Studio 16 installation, but the `190` compiler version will still be used because the necessary `toolset=v140` will be set.

The settings define the last digit `update: [None, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9]`, which by default is `None`, means that Conan assumes binary compatibility for the compiler patches, which works in general for the Microsoft compilers. For cases where finer control is desired, you can just add the `update` part to your profiles:

```
[settings]
compiler=msvc
compiler.version=191
compiler.version.update=3
```

This will be equivalent to the full version `1913 (19.13)`. If even further details is desired, you could even add your own digits to the `update` subsetting in `settings.yml`.

clang

The release 13.0.0 will be released officially on September 21, 2021. However, Conan 1.40 will support it in settings.yml before the final release. It will be considered as **experimental** in case of incompatibility until the release.

intel-cc

Available since: 1.41.0

This compiler is a new, **experimental** one, aimed to handle the new Intel oneAPI DPC++/C++/Classic compilers. Instead of having *n* different compilers, you have 3 different **modes** of working:

- `icx` for Intel oneAPI C++.
- `dpcpp` for Intel oneAPI DPC++.
- `classic` for Intel C++ Classic ones.

Besides that, Intel releases some versions with revisions numbers so the `update` field it's supposed to be any possible minor number for the Intel compiler version used, e.g, `compiler.version=2021.1` and `compiler.update=311` mean Intel version is `2021.1.311`.

For more information, you can check the *IntelCC section*.

Architectures

Here you can find a brief explanation of each of the architectures defined as `arch`, `arch_build` and `arch_target` settings.

- **x86**: The popular 32 bit x86 architecture.
- **x86_64**: The popular 64 bit x64 architecture.
- **ppc64le**: The PowerPC 64 bit Big Endian architecture.
- **ppc32**: The PowerPC 32 bit architecture.
- **ppc64le**: The PowerPC 64 bit Little Endian architecture.
- **ppc64**: The PowerPC 64 bit Big Endian architecture.
- **armv5el**: The ARM 32 bit version 5 architecture, soft-float.
- **armv5hf**: The ARM 32 bit version 5 architecture, hard-float.
- **armv6**: The ARM 32 bit version 6 architecture.
- **armv7**: The ARM 32 bit version 7 architecture.
- **armv7hf**: The ARM 32 bit version 7 hard-float architecture.
- **armv7s**: The ARM 32 bit version 7 *swift* architecture mostly used in Apple's A6 and A6X chips on iPhone 5, iPhone 5C and iPad 4.
- **armv7k**: The ARM 32 bit version 7 *k* architecture mostly used in Apple's WatchOS.
- **armv8**: The ARM 64 bit and 32 bit compatible version 8 architecture. It covers only the `aarch64` instruction set.
- **armv8_32**: The ARM 32 bit version 8 architecture. It covers only the `aarch32` instruction set (a.k.a. `ILP32`).
- **armv8.3**: The ARM 64 bit and 32 bit compatible version 8.3 architecture. Also known as `arm64e`, it is used on the A12 chipset added in the latest iPhone models (XS/XS Max/XR).
- **sparc**: The SPARC (Scalable Processor Architecture) originally developed by Sun Microsystems.
- **sparcv9**: The SPARC version 9 architecture.
- **mips**: The 32 bit MIPS (Microprocessor without Interlocked Pipelined Stages) developed by MIPS Technologies (formerly MIPS Computer Systems).
- **mips64**: The 64 bit MIPS (Microprocessor without Interlocked Pipelined Stages) developed by MIPS Technologies (formerly MIPS Computer Systems).
- **avr**: The 8 bit AVR microcontroller architecture developed by Atmel (Microchip Technology).
- **s390**: The 32 bit address Enterprise Systems Architecture 390 from IBM.
- **s390x**: The 64 bit address Enterprise Systems Architecture 390 from IBM.
- **asm.js**: The subset of JavaScript that can be used as low-level target for compilers, not really a processor architecture, it's produced by Emscripten. Conan treats it as an architecture to align with build systems design (e.g. GNU auto tools and CMake).

- **wasm**: The Web Assembly, not really a processor architecture, but byte-code format for Web, it's produced by Emscripten. Conan treats it as an architecture to align with build systems design (e.g. GNU auto tools and CMake).
- **sh4le**: The Hitachi SH-4 SuperH architecture.
- **e2k-v2**: The Elbrus 2000 v2 512 bit VLIW (Very Long Instruction Word) architecture (Elbrus 2CM, Elbrus 2C+ CPUs) originally developed by MCST (Moscow Center of SPARC Technologies).
- **e2k-v3**: The Elbrus 2000 v3 512 bit VLIW (Very Long Instruction Word) architecture (Elbrus 2S, aka Elbrus 4C, CPU) originally developed by MCST (Moscow Center of SPARC Technologies).
- **e2k-v4**: The Elbrus 2000 v4 512 bit VLIW (Very Long Instruction Word) architecture (Elbrus 8C, Elbrus 8C1, Elbrus 1C+ and Elbrus 1CK CPUs) originally developed by MCST (Moscow Center of SPARC Technologies).
- **e2k-v5**: The Elbrus 2000 v5 512 bit VLIW (Very Long Instruction Word) architecture (Elbrus 8C2 ,aka Elbrus 8CB, CPU) originally developed by MCST (Moscow Center of SPARC Technologies).
- **e2k-v6**: The Elbrus 2000 v6 512 bit VLIW (Very Long Instruction Word) architecture (Elbrus 2C3, Elbrus 12C and Elbrus 16C CPUs) originally developed by MCST (Moscow Center of SPARC Technologies).
- **e2k-v7**: The Elbrus 2000 v7 512 bit VLIW (Very Long Instruction Word) architecture (Elbrus 32C CPU) originally developed by MCST (Moscow Center of SPARC Technologies).

C++ standard libraries (aka compiler.libcxx)

`compiler.libcxx` sub-setting defines C++ standard libraries implementation to be used. The sub-setting applies only to certain compilers, e.g. it applies to *clang*, *apple-clang* and *gcc*, but doesn't apply to *Visual Studio*.

- **libstdc++** (gcc, clang, apple-clang, sun-cc): The GNU C++ Library. NOTE that this implicitly defines **_GLIBCXX_USE_CXX11_ABI=0** to use old ABI. See *How to manage the GCC >= 5 ABI* for the additional details. Might be a wise choice for old systems, such as CentOS 6. On Linux systems, you may need to install libstdc++-dev (package name could be different in various distros) in order to use the standard library. NOTE that on Apple systems usage of **libstdc++** has been deprecated.
- **libstdc++11** (gcc, clang, apple-clang): The GNU C++ Library. NOTE that this implicitly defines **_GLIBCXX_USE_CXX11_ABI=1** to use new ABI. See *How to manage the GCC >= 5 ABI* for the additional details. Might be a wise choice for newer systems, such as Ubuntu 20. On Linux systems, you may need to install libstdc++-dev (package name could be different in various distros) in order to use the standard library. NOTE that on Apple systems usage of **libstdc++** has been deprecated.
- **libc++** (clang, apple-clang): LLVM libc++. On Linux systems, you may need to install libc++-dev (package name could be different in various distros) in order to use the standard library.
- **c++_shared** (clang, Android only): use LLVM libc++ as a shared library. Refer to the C++ Library Support for the additiona details.
- **c++_static** (clang, Android only): use LLVM libc++ as a static library. Refer to the C++ Library Support for the additiona details.
- **libCstd** (sun-cc): Rogue Wave's stdlib. See Comparing C++ Standard Libraries libCstd, libstlport, and libstdcxx.
- **libstlport** (sun-cc): STLport. See Comparing C++ Standard Libraries libCstd, libstlport, and libstdcxx.
- **libstdcxx** (sun-cc): Apache C++ Standard Library. See Comparing C++ Standard Libraries libCstd, libstlport, and libstdcxx.
- **gpp** (qcc): GNU C++ lib. See QCC documentation.
- **cpp** (qcc): Dinkum C++ lib. See QCC documentation.
- **cpp-ne** (qcc): Dinkum C++ lib (no exceptions). See QCC documentation.

- **acpp** (qcc): Dinkum Abridged C++ lib. See QCC documentation.
- **acpp-ne** (qcc): Dinkum Abridged C++ lib (no exceptions). See QCC documentation.
- **ecpp** (qcc): Embedded Dinkum C++ lib. See QCC documentation.
- **ecpp-ne** (qcc): Embedded Dinkum C++ lib (no exceptions). See QCC documentation.
- **cxx** (qcc): LLVM C++. See QCC documentation.

18.9 Environment variables

These are the environment variables used to customize Conan.

> Most of them can be set in the *conan.conf* configuration file (inside your `<userhome>/.conan` folder). However, this environment variables will take precedence over the *conan.conf* configuration.

18.9.1 CMAKE RELATED VARIABLES

There are some Conan environment variables that will set the equivalent CMake variable using the *cmake generator* and the *CMake build tool*:

Variable	CMake set variable
CONAN_CMAKE_TOOLCHAIN_FILE	CMAKE_TOOLCHAIN_FILE
CONAN_CMAKE_SYSTEM_NAME	CMAKE_SYSTEM_NAME
CONAN_CMAKE_SYSTEM_VERSION	CMAKE_SYSTEM_VERSION
CONAN_CMAKE_SYSTEM_PROCESSOR	CMAKE_SYSTEM_PROCESSOR
CONAN_CMAKE_SYSROOT	CMAKE_SYSROOT
CONAN_CMAKE_FIND_ROOT_PATH	CMAKE_FIND_ROOT_PATH
CONAN_CMAKE_FIND_ROOT_PATH_MODE_PROGRA	CMAKE_FIND_ROOT_PATH_MODE_PROGRAM
CONAN_CMAKE_FIND_ROOT_PATH_MODE_LIBRARY	CMAKE_FIND_ROOT_PATH_MODE_LIBRARY
CONAN_CMAKE_FIND_ROOT_PATH_MODE_INCLUDE	CMAKE_FIND_ROOT_PATH_MODE_INCLUDE
CONAN_CMAKE_GENERATOR_PLATFORM	CMAKE_GENERATOR_PLATFORM
CONAN_CMAKE_ANDROID_NDK	CMAKE_ANDROID_NDK

See also:

See CMake cross building wiki

18.9.2 CONAN_BASH_PATH

Defaulted to: Not defined

Used only in windows to help the *tools.run_in_windows_bash()* function to locate our Cygwin/MSYS2 bash. Set it with the bash executable path if it's not in the `PATH` or you want to use a different one.

18.9.3 CONAN_CACHE_NO_LOCKS

Defaulted to: `False/0`

Set it to `True/1` to disable locking mechanism of local cache. Set it to `False/0` to enable locking mechanism of local cache. Use it with caution, and only for debugging purposes. Disabling locks may easily lead to corrupted packages. Not recommended for production environments, and in general should be used for conan development and contributions only.

18.9.4 CONAN_CMAKE_GENERATOR

Conan `CMake` helper class is just a convenience to help to translate Conan settings and options into CMake parameters, but you can easily do it yourself, or adapt it.

For some compiler configurations, as `gcc` it will use by default the `Unix Makefiles` CMake generator. Note that this is not a package settings, building it with makefiles or other build system, as Ninja, should lead to the same binary if using appropriately the same underlying compiler settings. So it doesn't make sense to provide a setting or option for this.

So it can be set with the environment variable `CONAN_CMAKE_GENERATOR`. Just set its value to your desired CMake generator (as `Ninja`).

18.9.5 CONAN_CMAKE_GENERATOR_PLATFORM

Defines generator platform to be used by particular CMake generator (see *CMAKE_GENERATOR_PLATFORM documentation <https://cmake.org/cmake/help/latest/variable/CMAKE_GENERATOR_PLATFORM.html>*). Resulting value is passed to the `cmake` command line (`-A` argument) by the Conan `CMake` helper class during the configuration step. Passing `None` causes auto-detection, which currently only happens for the `Visual Studio 16 2019` generator. The detection is according to the following table:

settings.arch	generator platform
x86	Win32
x86_64	x64
armv7	ARM
armv8	ARM64
other	(none)

For any other generators besides the `Visual Studio 16 2019` generator, detection results in no generator platform applied (and no `-A` argument passed to the CMake command line).

18.9.6 CLICOLOR

Defaulted to: Not defined

Set it to `0` to disable console output colors, overriding tty detection. Set it to any value other than `0` to enable console output colors if a tty is detected. If this is left undefined, Conan will use the `CONAN_COLOR_DISPLAY` logic to determine whether colors should be enabled.

18.9.7 CLICOLOR_FORCE

Defaulted to: Not defined

Set it to any value other than `0` to force the generation of console output colors, overriding tty detection and `CLICOLOR`.

18.9.8 NO_COLOR

Defaulted to: Not defined

Set it to any value to force disable console output colors, overriding tty detection and any other color output controls.

18.9.9 CONAN_COLOR_DARK

Defaulted to: `False/0`

Set it to `True/1` to use dark colors in the terminal output, instead of light ones. Useful for terminal or consoles with light colors as white, so text is rendered in Blue, Black, Magenta, instead of Yellow, Cyan, White.

18.9.10 CONAN_COLOR_DISPLAY

Defaulted to: Not defined

By default if undefined Conan output will use color if a tty is detected.

Set it to `False/0` to remove console output colors. Set it to `True/1` to force console output colors.

18.9.11 CONAN_COMPRESSION_LEVEL

Defaulted to: `9`

Conan uses *.tgz* compression for archives before uploading them to remotes. The default compression level is good and fast enough for most cases, but users with huge packages might want to change it and set `CONAN_COMPRESSION_LEVEL` environment variable to a lower number, which is able to get slightly bigger archives but much better compression speed.

18.9.12 CONAN_CPU_COUNT

Defaulted to: Number of available cores in your machine.

Set the number of cores that the *tools.cpu_count()* will return. Conan recipes can use the `cpu_count()` tool to build the library using more than one core.

18.9.13 CONAN_DEFAULT_PROFILE_PATH

Defaulted to: Not defined

This variable can be used to define a path to an existing profile file that Conan will use as default. If relative, the path will be resolved from the profiles folder.

18.9.14 CONAN_NON_INTERACTIVE

Defaulted to: `False/0`

This environment variable, if set to `True/1`, will prevent interactive prompts. Invocations of Conan commands where an interactive prompt would otherwise appear, will fail instead.

This variable can also be set in `conan.conf` as `non_interactive = True` in the `[general]` section.

18.9.15 CONAN_ENV_XXXX_YYYY

You can override the default settings (located in your `~/.conan/profiles/default` directory) with environment variables.

The `XXXX` is the setting name upper-case, and the `YYYY` (optional) is the sub-setting name.

Examples:

- Override the default compiler:

```
CONAN_ENV_COMPILER = "Visual Studio"
```

- Override the default compiler version:

```
CONAN_ENV_COMPILER_VERSION = "14"
```

- Override the architecture:

```
CONAN_ENV_ARCH = "x86"
```

18.9.16 CONAN_LOG_RUN_TO_FILE

Defaulted to: `0`

If set to 1 will log every `self.run("{Some command}")` command output in a file called `conan_run.log`. That file will be located in the current execution directory, so if we call `self.run` in the conanfile.py's build method, the file will be located in the build folder.

In case we execute `self.run` in our `source()` method, the `conan_run.log` will be created in the source directory, but then conan will copy it to the `build` folder following the regular execution flow. So the `conan_run.log` will contain all the logs from your conanfile.py command executions.

The file can be included in the Conan package (for debugging purposes) using the `package` method.

```
def package(self):
    self.copy(pattern="conan_run.log", dst="", keep_path=False)
```

18.9.17 CONAN_LOG_RUN_TO_OUTPUT

Defaulted to: 1

If set to `0` Conan won't print the command output to the stdout. Can be used with `CONAN_LOG_RUN_TO_FILE` set to `1` to log only to file and not printing the output.

18.9.18 CONAN_LOGGING_LEVEL

Defaulted to: `critical`

By default Conan logging level is only set for critical events. If you want to show more detailed logging information, set this variable according to Python Logging Levels or, use a logging level name:

logging level name	logging level id
critical	50
error	40
warning/warn	30
info	20
debug	10

Both names and IDs are acceptable by environment variable, or using the conan.conf file.

18.9.19 CONAN_LOGIN_USERNAME, CONAN_LOGIN_USERNAME_{REMOTE_NAME}

Defaulted to: Not defined

You can define the username for the authentication process using environment variables. Conan will use a variable **CONAN_LOGIN_USERNAME_{REMOTE_NAME}**, if the variable is not declared Conan will use the variable **CONAN_LOGIN_USERNAME**, if the variable is not declared either, Conan will request to the user to input a username.

These variables are useful for unattended executions like CI servers or automated tasks.

If the remote name contains "-" you have to replace it with "_" in the variable name:

For example: For a remote named "conancenter":

```
SET CONAN_LOGIN_USERNAME_CONANCENTER=MyUser
```

See also:

See the *conan user* command documentation for more information about login to remotes

18.9.20 CONAN_LOGIN_ENCRYPTION_KEY

Defaulted to: Not defined

This variable is used to obfuscate the credential token when it is stored in the database after a successful **`conan user`** command. The encryption algorithm is a basic Vigenere cypher which is **not ok for security at all**.

This variable, however, is useful for shared CI servers where the stored value can be compromised: assign a random generated string to this value for each of the builds and configure your server to expire tokens, this will make the value stored in the database harder to crack.

18.9.21 CONAN_MAKE_PROGRAM

Defaulted to: Not defined

Specify an alternative `make` program to use with:

- The build helper *AutoToolsBuildEnvironment*. Will invoke the specified executable in the *make* method.
- The build helper *build helper CMake*. By adjusting the CMake variable CMAKE_MAKE_PROGRAM.

For example:

```
CONAN_MAKE_PROGRAM="/path/to/mingw32-make"

# Or only the exe name if it is in the path

CONAN_MAKE_PROGRAM="mingw32-make"
```

18.9.22 CONAN_CMAKE_PROGRAM

Defaulted to: Not defined

Specify an alternative `cmake` program to use with *CMake* build helper.

For example:

```
CONAN_CMAKE_PROGRAM="scan-build cmake"
```

18.9.23 CONAN_MSBUILD_VERBOSITY

Defaulted to: Not defined

Specify `` `MSBuild` `` verbosity level to use with:

- The build helper *CMake*.
- The build helper *MSBuild*.

For list of allowed values and their meaning, check out the MSBuild documentation.

18.9.24 CONAN_PASSWORD, CONAN_PASSWORD_{REMOTE_NAME}

Defaulted to: Not defined

You can define the authentication password using environment variables. Conan will use a variable **CONAN_PASSWORD_{REMOTE_NAME}**, if the variable is not declared Conan will use the variable **CONAN_PASSWORD**, if the variable is not declared either, Conan will request to the user to input a password.

These variables are useful for unattended executions like CI servers or automated tasks.

The remote name is transformed to all uppercase. If the remote name contains "-", you have to replace it with "_" in the variable name.

For example, for a remote named "conancenter":

```
SET CONAN_PASSWORD_CONANCENTER=Mypassword
```

See also:

See the *conan user* command documentation for more information about login to remotes

18.9.25 CONAN_HOOKS

Defaulted to: Not defined

Can be set to a comma separated list with the names of the hooks that will be executed when running a Conan command.

18.9.26 CONAN_PRINT_RUN_COMMANDS

Defaulted to: `0`

If set to 1, every `self.run("{Some command}")` call will log the executed command {Some command} to the output.

For example: In the *conanfile.py* file:

```
self.run("cd %s && %s ./configure" % (self.ZIP_FOLDER_NAME, env_line))
```

Will print to the output (stout and/or file):

```
----Running------
> cd zlib-1.2.9 && env LIBS="" LDFLAGS=" -m64   $LDFLAGS" CFLAGS="-mstackrealign -fPIC
↪$CFLAGS -m64  -s -DNDEBUG  " CPPFLAGS="$CPPFLAGS -m64  -s -DNDEBUG  " C_INCLUDE_PATH=
↪$C_INCLUDE_PATH: CPLUS_INCLUDE_PATH=$CPLUS_INCLUDE_PATH: ./configure
-----------------
...
```

18.9.27 CONAN_READ_ONLY_CACHE

Defaulted to: Not defined

This environment variable if defined, will make the Conan cache read-only. This could prevent developers to accidentally edit some header of their dependencies while navigating code in their IDEs.

This variable can also be set in `conan.conf` as `read_only_cache = True` in the `[general]` section.

The packages are made read-only in two points: when a package is built from sources, and when a package is retrieved from a remote repository.

The packages are not modified for upload, so users should take that into consideration before uploading packages, as they will be read-only and that could have other side-effects.

> **Warning:** It is not recommended to upload packages directly from developers machines with read-only mode as it could lead to inconsistencies. For better reproducibility we recommend that packages are created and uploaded by CI machines.

18.9.28 CONAN_RUN_TESTS

Defaulted to: Not defined (True/False if defined)

This environment variable (if defined) can be used in `conanfile.py` to enable/disable the tests for a library or application.

It can be used as a convention variable and it's specially useful if a library has unit tests and you are doing *cross building*, the target binary can't be executed in current host machine building the package.

It can be defined in your profile files at `~/.conan/profiles`

```
...
[env]
CONAN_RUN_TESTS=False
```

or declared in command line when invoking **`conan install`** to reduce the variable scope for conan execution

```
$ conan install . -e CONAN_RUN_TESTS=0
```

See how to retrieve the value with *tools.get_env()* and check a use case with *a header only with unit tests recipe* while cross building.

This variable is evaluated inside the build helper call to `test()` and will not run the tests if set to `False`.

```
from conans import ConanFile, CMake, tools

class HelloConan(ConanFile):
    name = "hello"
    version = "0.1"

    def build(self):
        cmake = CMake(self)
        cmake.configure()
        cmake.build()
        cmake.test()
```

18.9.29 CONAN_SKIP_VS_PROJECTS_UPGRADE

Defaulted to: `False/0`

When set to `True/1`, the *tools.build_sln_command()*, the *tools.msvc_build_command()* and the *MSBuild()* build helper, will not call `devenv` command to upgrade the `sln` project, irrespective of the `upgrade_project` parameter value.

18.9.30 CONAN_SYSREQUIRES_MODE

Defaulted to: Not defined (allowed values `enabled/verify/disabled`)

This environment variable controls whether system packages should be installed into the system via `SystemPackageTool` helper, typically used in *system_requirements()*.

See values behavior:

- `enabled`: Default value and any call to install method of `SystemPackageTool` helper should modify the system packages.

- `verify`: Display a report of system packages to be installed and abort with exception. Useful if you don't want to allow Conan to modify your system but you want to get a report of packages to be installed.
- `disabled`: Display a report of system packages that should be installed but continue the Conan execution and doesn't install any package in your system. Useful if you want to keep manual control of these dependencies, for example in your development environment.

18.9.31 CONAN_SYSREQUIRES_SUDO

Defaulted to: `True/1`

This environment variable controls whether `sudo` is used for installing apt, yum, etc. system packages via `SystemPackageTool` helper, typically used in `system_requirements()`. By default when the environment variable does not exist, "True" is assumed, and `sudo` is automatically prefixed in front of package management commands. If you set this to "False" or "0" `sudo` will not be prefixed in front of the commands, however installation or updates of some packages may fail due to a lack of privilege, depending on the user account Conan is running under.

18.9.32 CONAN_TEMP_TEST_FOLDER

Defaulted to: `False/0`

Activating this variable will make build folder of *test_package* to be created in the temporary folder of your machine.

18.9.33 CONAN_TRACE_FILE

Defaulted to: Not defined

If you want extra logging information about your Conan command executions, you can enable it by setting the `CONAN_TRACE_FILE` environment variable. Set it with an absolute path to a file.

```
export CONAN_TRACE_FILE=/tmp/conan_trace.log
```

When the Conan command is executed, some traces will be appended to the specified file. Each line contains a JSON object. The `_action` field contains the action type, like `COMMAND` for command executions, `EXCEPTION` for errors and `REST_API_CALL` for HTTP calls to a remote.

The logger will append the traces until the `CONAN_TRACE_FILE` variable is unset or pointed to a different file.

See also:

Read more here: *How to log and debug a conan execution*

18.9.34 CONAN_USERNAME, CONAN_CHANNEL

> **Warning:** Environment variables `CONAN_USERNAME` and `CONAN_CHANNEL` are deprecated and will be removed in Conan 2.0. Don't use them to populate the value of `self.user` and `self.channel`.

These environment variables will be checked when using `self.user` or `self.channel` in package recipes in user space, where the user and channel have not been assigned yet (they are assigned when exported in the local cache). More about these variables in the *attributes reference*.

18.9.35 CONAN_USER_HOME

Defaulted to: Not defined

Allows defining a custom base directory for Conan cache directory. Can be useful for concurrent builds under different users in CI, to retrieve and store per-project specific dependencies (useful for deployment, for example). Conan will generate the folder `.conan` under the custom base path.

See also:

Read more about it in *Conan local cache: concurrency, Continuous Integration, isolation*

18.9.36 CONAN_USER_HOME_SHORT

Defaulted to: Not defined

Specify the base folder to be used with the *short paths* feature. When not specified, the packages marked as *short_paths* will be stored in the `C:\.conan` (or the current drive letter).

If set to `None`, it will disable the *short_paths* feature in Windows for modern Windows that enable long paths at the system level.

Setting this variable equal to, or to a subdirectory of, the local conan cache (e.g. ~/.conan) would result in an invalid cache configuration and is therefore disallowed.

18.9.37 CONAN_USE_ALWAYS_SHORT_PATHS

Defaulted to: Not defined

If defined to `True` or `1`, every package will be stored in the *short paths directory* resolved by Conan after evaluating `CONAN_USER_HOME_SHORT` variable (see above). This variable, therefore, overrides the value defined in recipes for the attribute *short paths*.

If the variable is not defined or it evaluates to `False` then every recipe will be stored according to the value of its `short_paths` attribute. So, `CONAN_USE_ALWAYS_SHORT_PATHS` can force every recipe to use short paths, but it won't work to force the opposite behavior.

18.9.38 CONAN_VERBOSE_TRACEBACK

Defaulted to: `0`

When an error is raised in a recipe or even in the Conan code base, if set to `1` it will show the complete traceback to ease the debugging.

18.9.39 CONAN_ERROR_ON_OVERRIDE

Defaulted to: `False`

When a consumer overrides one transitive requirement without using explicitly the keyword `override` Conan will raise an error if this environmente variable is set to `True`.

This variable can also be set in the **conan.conf** file under the section `[general]`.

18.9.40 CONAN_VS_INSTALLATION_PREFERENCE

Defaulted to: `Enterprise, Professional, Community, BuildTools`

This environment variables defines the order of preference when searching for a Visual installation product. This would affect every tool that uses `tools.vs_installation_path()` and will search in the order indicated.

For example:

```
set CONAN_VS_INSTALLATION_PREFERENCE=Enterprise, Professional, Community, BuildTools
```

It can also be used to fix the type of installation you want to use indicating just one product type:

```
set CONAN_VS_INSTALLATION_PREFERENCE=BuildTools
```

18.9.41 CONAN_CACERT_PATH

Defaulted to: Not defined

Specify an alternative path to a *cacert.pem* file to be used for requests. This variable overrides the value defined in the *conan.conf* as `cacert_path = <path/to/cacert.pem>` under the section `[general]`.

18.9.42 CONAN_DEFAULT_PACKAGE_ID_MODE

Defaulted to: semver_direct_mode

It changes the way package IDs are computed, but can change to any value defined in *Using package_id() for Package Dependencies*.

18.9.43 CONAN_SKIP_BROKEN_SYMLINKS_CHECK

Defaulted to: `False/0`

When set to `True/1`, Conan will allow the existence broken symlinks while creating a package.

18.9.44 CONAN_PYLINT_WERR

Defaulted to: Not defined

This environment variable changes the PyLint behavior from *warning* level to *error*. Therefore, any inconsistency found in the recipe will break the process during linter analysis.

18.9.45 CONAN_KEEP_PYTHON_FILES

Defaulted to: `False`

This environment variable will allow Python *.pyc* files to be packaged. If not set as `True/1`, all the generated *.pyc* files will be filtered when packaging.

18.10 Hooks

Warning: This is an **experimental** feature subject to breaking changes in future releases.

The Conan hooks are Python functions that are intended to extend the Conan functionalities and let users customize the client behavior at determined execution points. Check the *hooks section in extending Conan* to see some examples of how to use them and already available ones providing useful functionality.

18.10.1 Hook interface

Here you can see a complete example of all the hook functions available and the different parameters for each of them depending on the context:

```
def pre_export(output, conanfile, conanfile_path, reference, **kwargs):
    assert conanfile
    output.info("conanfile_path={}".format(conanfile_path))
    output.info("reference={}".format(reference.full_str()))

def post_export(output, conanfile, conanfile_path, reference, **kwargs):
    assert conanfile
    output.info("conanfile_path={}".format(conanfile_path))
    output.info("reference={}".format(reference.full_str()))

def pre_source(output, conanfile, conanfile_path, **kwargs):
    assert conanfile
    output.info("conanfile_path={}".format(conanfile_path))
    if conanfile.in_local_cache:
        output.info("reference={}".format(kwargs["reference"].full_str()))

def post_source(output, conanfile, conanfile_path, **kwargs):
    assert conanfile
    output.info("conanfile_path={}".format(conanfile_path))
    if conanfile.in_local_cache:
        output.info("reference={}".format(kwargs["reference"].full_str()))

def pre_build(output, conanfile, **kwargs):
    assert conanfile
    if conanfile.in_local_cache:
        output.info("reference={}".format(kwargs["reference"].full_str()))
        output.info("package_id={}".format(kwargs["package_id"]))
    else:
        output.info("conanfile_path={}".format(kwargs["conanfile_path"]))

def post_build(output, conanfile, **kwargs):
    assert conanfile
    if conanfile.in_local_cache:
        output.info("reference={}".format(kwargs["reference"].full_str()))
        output.info("package_id={}".format(kwargs["package_id"]))
    else:
        output.info("conanfile_path={}".format(kwargs["conanfile_path"]))
```

(continues on next page)

(continued from previous page)

```
def pre_package(output, conanfile, conanfile_path, **kwargs):
    assert conanfile
    output.info("conanfile_path={}".format(conanfile_path))
    if conanfile.in_local_cache:
        output.info("reference={}".format(kwargs["reference"].full_str()))
        output.info("package_id={}".format(kwargs["package_id"]))

def post_package(output, conanfile, conanfile_path, **kwargs):
    assert conanfile
    output.info("conanfile_path={}".format(conanfile_path))
    if conanfile.in_local_cache:
        output.info("reference={}".format(kwargs["reference"].full_str()))
        output.info("package_id={}".format(kwargs["package_id"]))

def pre_upload(output, conanfile_path, reference, remote, **kwargs):
    output.info("conanfile_path={}".format(conanfile_path))
    output.info("reference={}".format(reference.full_str()))
    output.info("remote.name={}".format(remote.name))

def post_upload(output, conanfile_path, reference, remote, **kwargs):
    output.info("conanfile_path={}".format(conanfile_path))
    output.info("reference={}".format(reference.full_str()))
    output.info("remote.name={}".format(remote.name))

def pre_upload_recipe(output, conanfile_path, reference, remote, **kwargs):
    output.info("conanfile_path={}".format(conanfile_path))
    output.info("reference={}".format(reference.full_str()))
    output.info("remote.name={}".format(remote.name))

def post_upload_recipe(output, conanfile_path, reference, remote, **kwargs):
    output.info("conanfile_path={}".format(conanfile_path))
    output.info("reference={}".format(reference.full_str()))
    output.info("remote.name={}".format(remote.name))

def pre_upload_package(output, conanfile_path, reference, package_id, remote, **kwargs):
    output.info("conanfile_path={}".format(conanfile_path))
    output.info("reference={}".format(reference.full_str()))
    output.info("package_id={}".format(package_id))
    output.info("remote.name={}".format(remote.name))

def post_upload_package(output, conanfile_path, reference, package_id, remote, **kwargs):
    output.info("conanfile_path={}".format(conanfile_path))
    output.info("reference={}".format(reference.full_str()))
    output.info("package_id={}".format(package_id))
    output.info("remote.name={}".format(remote.name))

def pre_download(output, reference, remote, **kwargs):
    output.info("reference={}".format(reference.full_str()))
    output.info("remote.name={}".format(remote.name))

def post_download(output, conanfile_path, reference, remote, **kwargs):
```

(continues on next page)

(continued from previous page)

```
    output.info("conanfile_path={}".format(conanfile_path))
    output.info("reference={}".format(reference.full_str()))
    output.info("remote.name={}".format(remote.name))

def pre_download_recipe(output, reference, remote, **kwargs):
    output.info("reference={}".format(reference.full_str()))
    output.info("remote.name={}".format(remote.name))

def post_download_recipe(output, conanfile_path, reference, remote, **kwargs):
    output.info("conanfile_path={}".format(conanfile_path))
    output.info("reference={}".format(reference.full_str()))
    output.info("remote.name={}".format(remote.name))

def pre_download_package(output, conanfile, conanfile_path, reference, package_id,␣
↪remote, **kwargs):
    output.info("conanfile.name={}".format(conanfile.name))
    output.info("conanfile_path={}".format(conanfile_path))
    output.info("reference={}".format(reference.full_str()))
    output.info("package_id={}".format(package_id))
    output.info("remote.name={}".format(remote.name))

def post_download_package(output, conanfile, conanfile_path, reference, package_id,␣
↪remote, **kwargs):
    output.info("conanfile.name={}".format(conanfile.name))
    output.info("conanfile_path={}".format(conanfile_path))
    output.info("reference={}".format(reference.full_str()))
    output.info("package_id={}".format(package_id))
    output.info("remote.name={}".format(remote.name))

def pre_package_info(output, conanfile, reference, **kwargs):
    output.info("reference={}".format(reference.full_str()))
    output.info("conanfile.cpp_info.defines={}".format(conanfile.cpp_info.defines))

def post_package_info(output, conanfile, reference, **kwargs):
    output.info("reference={}".format(reference.full_str()))
    output.info("conanfile.cpp_info.defines={}".format(conanfile.cpp_info.defines))
```

Functions of the hooks are intended to be self-descriptive regarding to the execution of them. For example, the `pre_package()` function is called just before the `package()` method of the recipe is executed.

For download/upload functions, the `pre_download()`/`pre_upload()` function is executed first in an **conan download**/**conan upload** command. Then **pre** and **post** `download_recipe()`/`upload_recipe()` and its subsequent **pre**/**post** `download_package()`/`upload_package()` if that is the case. Finally the general `post_download()`/`post_upload()` function is called to wrap up the whole execution.

Important: **Pre** and **post** `download_recipe()`/`download_package()` are also executed when installing new recipes/packages from remotes using **conan create** or **conan install**.

18.10.2 Function parameters

Here you can find the description for each parameter:

- **output**: *Output object* to print formatted messages during execution with the name of the hook and the function executed, e.g., `[HOOK - complete_hook] post_download_package(): This is the remote name: default`.
- **conanfile**: It is a regular `ConanFile` object loaded from the recipe that received the Conan command. It has its normal attributes and dynamic objects such as `build_folder`, `package_folder`...
- **conanfile_path**: Path to the *conanfile.py* file whether it is in local cache or in user space.
- **reference**: Named tuple with attributes `name`, `version`, `user`, and `channel`. Its representation will be a reference like: `box2d/2.1.0@user/channel`
- **package_id**: String with the computed package ID.
- **remote**: Named tuple with attributes `name`, `url` and `verify_ssl`.

Hook function*	Parameters				
	conanfile	conanfile_path	reference	package_id	remote
export()	Yes	pre/post	Yes	No	No
source()	Yes	Yes	cache	No	No
build()	Yes	user space	cache	cache	No
package()	Yes	pre/post	cache	Yes	No
upload() upload_recipe(upload_package	No	Yes	Yes	Yes	Yes
download() download_recip	Yes	post	Yes	Yes	Yes
download_packa	Yes	Yes	Yes	Yes	Yes
package_info()	Yes	No	Yes	No	No

*Hook functions are indicated without `pre` and `post` prefixes for simplicity.

Table legend:

- **Yes**: Availability in `pre` and `post` functions in any context.
- **No**: Not available.
- **pre / post**: Availability in both `pre` and `post` functions with **different values**. e.g. `conanfile_path` pointing to user space in `pre` and to local cache in `post`.
- **post**: Only available in `post` function.
- **cache**: Only available when the context of the command executed is the local cache. e.g. **`conan create`**, **`conan install`**...

- **user space**: Only available when the context of the command executed is the user space. e.g. **conan build**

Note: Path to the different folders of the Conan execution flow may be accessible as usual through the `conanfile` object. See *source_folder* to learn more.

Some of this parameters does not appear in the signature of the function as they may not be always available (Mostly depending on the recipe living in the local cache or in user space). However, they can be checked with the `kwargs` parameter.

Important: Hook functions should have a `**kwargs` parameter to keep compatibility of new parameters that may be introduced in future versions of Conan.

18.11 CONAN_V2_MODE

If defined in the environment, this variable will raise errors whenever a *Conan 2.0* deprecated feature is used. It is a good mechanism to check the recipes future Conan 2.0 "compliance". Activating it should not change behavior in any way, just raise error for deprecated things, but if it works, the same result should be achieved. The number of deprecated features will increase in future Conan 1.X releases, it could be a good practice to have it activated in some nightly job or the like, to report on current status of your recipes.

So, if you are ready to experiment add the variable `CONAN_V2_MODE` to your environment and, please, report your feedback about it.

The following is a current known list of features that will change in Conan 2.0 and might start raising errors if CONAN_V2_MODE is activated:

18.11.1 Changes related to the default configuration

- First level setting *cppstd* is removed.
- Revisions are enabled by default (adds `revisions_enabled=1` to *conan.conf*).
- No hooks activated by default.
- SCM data will be stored into *conandata.yml*.
- GCC >= 5 autodetected profile will use `libstdc++11`.
- Directory `<cache>/python` is not added to Python `sys.path`.

18.11.2 Changes in recipes

These changes could break existing recipes:

- Forbid access to `self.cpp_info` in `conanfile::package_id()` method.
- Deprecate `conanfile::config()` method.
- Deprecate old `python_requires` syntax.
- Forbid access to `self.info` in `conanfile.package()`.
- `default_options` are required to be a dictionary.

- Raise if setting `cppstd` appears in the recipe.
- Forbid `self.settings` and `self.options` in `conanfile::source()` method.
- Deprecate `tools.msvc_build_command`.
- Deprecate `tools.build_sln_command`.
- Deprecate `cpp_info.cppflags` (use `cxxflags` instead).
- Deprecate environment variables `CONAN_USERNAME` and `CONAN_CHANNEL`.
- `PYTHONPATH` is not added automatically to the environment before running consumer functions.
- Attribute `self.version` is ensured to be a string in all the functions and scenarios.
- Access to member `name` in `deps_cpp_info` objects is forbidden, use `get_name(<generator>)` with the name of the generator.

18.11.3 Changes in profiles

Could break existing profiles:

- Deprecate `scopes` section in profiles.

18.11.4 Other changes

- Package name used by the `pkg_config` generator uses the same rules as any other generator. Previously, if it was not explicit, it was using lowercase `cpp_info.name` when it was different from the package name.
- If `build_type` or `compiler` are not defined when using build helpers Conan will raise an error.
- New compiler detection algorithm is used (e.g. when running `conan profile new <name> --detect`). Previously, `<compiler> --version` was parsed to detect the compiler and its version. Now, using `CONAN_V2_MODE`, Conan will try to detect the compiler and its version via compiler's built-in macro definitions.

Note: More changes will be added, some of them could be reverted and the behavior may change without further noticing. If you are using `CONAN_V2_MODE`, **thanks!** We really appreciate your feedback about the future of Conan.

CHAPTER

NINETEEN

VIDEOS AND LINKS

- Conan in Practice: Interactive Exercises of common commands
- NDC TechTown 2019: Using Conan in a real-world complex project by Kristian Jerpetjøn.
- Meeting Embedded 2018: "Continuous Integration of C/C++ for embedded and IoT with Jenkins, Docker and Conan" by Diego Rodriguez-Losada and Daniel Manzaneque.
- CppCon 2018: "Git, CMake, Conan - How to ship and reuse our C++ projects" by Mateusz Pusz.
- JFrog swampUP 2018: "Managing dependencies and toolchains with Conan and Artifactory" by Tobias Hieta
- JFrog swampUP 2018: "Cross building… It's almost too easy!" by Théo Delrieu.
- JFrog Conan Playlist: "Conan - The C/C++ Package Manager"
- FOSDEM 2018: "Packaging C/C++ libraries with Conan" by Théo Delrieu.

  Includes AndroidNDK package and cross build to Android
- CppCon 2016: "Introduction to Conan C/C++ Package Manager" by Diego Rodriguez-Losada.
- CppCon 2017: "Faster Delivery of Large C/C++ Projects with Conan Package Manager and Efficient Continuous Integration" by Diego Rodriguez-Losada.
- "Conan.io C++ Package Manager demo with SFML" by Charl Botha
- CppRussia 2019: "ABI compatibility is not a MAJOR problem" by Javier Garcia Sogo
- CppCon 2019: "Building happiness in your life" by Steve Robinson

Do you have a video, tutorial, blog post that could be useful for other users and would like to share? Please tell us about it or directly send a PR to our docs: https://github.com/conan-io/docs, and we will link it here.

CHAPTER

TWENTY

FAQ

See also:

There is a great community behind Conan with users helping each other in Cpplang Slack. Please join us in the `#conan` channel!

20.1 General

20.1.1 Is Conan CMake based, or is CMake a requirement?

No. It isn't. Conan is build-system agnostic. Package creators could very well use cmake to create their packages, but you will only need it if you want to build packages from source, or if there are no available precompiled packages for your system/settings. We use CMake extensively in our examples and documentation, but only because it is very convenient and most C/C++ devs are familiar with it.

20.1.2 Is build-system XXXXX supported?

Yes. It is. Conan makes no assumption about the build system. It just wraps any build commands specified by the package creators. There are already some helper methods in code to ease the use of CMake, but similar functions can be very easily added for your favorite build system. Please check out the alternatives explained in *generator packages*

20.1.3 Is my compiler, version, architecture, or setting supported?

Yes. Conan is very general, and does not restrict any configuration at all. However, Conan comes with some compilers, versions, architectures, ..., etc. pre-configured in the `~/.conan/settings.yml` file, and you can get an error if using settings not present in that file. Go to *invalid settings* to learn more about it, or see the section *Customizing settings*.

20.1.4 Does it run offline?

Yes. It runs offline very well. Package recipes and binary packages are stored in your machine, per user, and so you can start new projects that depend on the same libraries without any Internet connection at all. Packages can be fully created, tested and consumed locally, without needing to upload them anywhere.

20.1.5 Is it possible to install 2 different versions of the same library?

Yes. You can install as many different versions of the same library as you need, and easily switch among them in the same project, or have different projects use different versions simultaneously, and without having to install/uninstall or re-build any of them.

Package binaries are stored per user in (e.g.) `~/.conan/data/Boost/1.59/user/stable/package/{sha_0, sha_1, sha_2...}` with a different SHA signature for every different configuration (debug, release, 32-bit, 64-bit, compiler...). Packages are managed per user, but additionally differentiated by version and channel, and also by their configuration. So large packages, like Boost, don't have to be compiled or downloaded for every project.

20.1.6 Can I run multiple Conan isolated instances (virtual environments) on the same machine?

Yes, Conan supports the concept of virtual environments; so it manages all the information (packages, remotes, user credentials, ..., etc.) in different, isolated environments. Check *virtual environments* for more details.

20.1.7 Can I run the conan_server or Artifactory behind a firewall (on-premises)?

Yes. Conan does not require a connection to conan.io site or any other external service at all for its operation. You can install packages from the ConanCenter repository if you want, test them, and only after approval, upload them to your on-premises server and forget about the original repository. Or you can just get the package recipes, re-build from source on your premises, and then upload the packages to your server.

20.1.8 Can I connect to Conan remote servers through a corporate proxy?

Yes, it can be configured in your **~/.conan/conan.conf** configuration file or with some environment variables. Check *proxy configuration* for more details.

20.1.9 Can I create packages for third-party libraries?

Of course, as long as their license allows it.

20.1.10 Can I upload closed source libraries to ConanCenter?

No. ConanCenter (https://conan.io/center/) is for Open Source packages only. Binaries in ConanCenter are created by our build service from recipes in https://github.com/conan-io/conan-center-index. Read how to contribute to ConanCenter in https://github.com/conan-io/conan-center-index/wiki

20.1.11 Do I always need to specify how to build the package from source?

No. But it is highly recommended. If you want, you can just directly start with the binaries, build elsewhere, and upload them directly. Maybe your `build()` step can download pre-compiled binaries from another source and unzip them, instead of actually compiling from sources. You can also use the *conan export-pkg* command to create packages from existing binaries.

20.1.12 Does Conan use semantic versioning (semver) for dependencies?

It uses a convention by which package dependencies follow semver by default; thus it intelligently avoids recompilation/repackaging if you update upstream minor versions, but will correctly do so if you update major versions upstream. This behavior can be easily configured and changed in the `package_id()` method of your conanfile, and any versioning scheme you desire is supported.

20.2 Using Conan

20.2.1 How to package header-only libraries?

Packaging header-only libraries is similar to other packages. Be sure to start by reading and understanding the *packaging getting started guide*. The main difference is that a package recipe is typically much simpler. There are different approaches depending on if you want Conan to run the library unit tests while creating the package or not. Full details are described *in this how-to guide*.

20.2.2 When to use settings or options?

While creating a package, you may want to add different configurations and variants of the package. There are two main inputs that define packages: settings and options.

Settings are a project-wide configuration, something that typically affects the whole project that is being built. For example, the operating system or the architecture would be naturally the same for all packages in a dependency graph, linking a Linux library for a Windows app, or mixing architectures is impossible.

Settings cannot be defaulted in a package recipe. A recipe for a given library cannot say that its default is `os=Windows`. The `os` will be given by the environment in which that recipe is processed. It is a mandatory input.

Settings are configurable. You can edit, add, remove settings or subsettings in your *settings.yml* file. See *the settings.yml reference*.

On the other hand, **options** are a package-specific configuration. Static or shared library are not settings that apply to all packages. Some can be header only libraries while others packages can be just data, or package executables. Packages can contain a mixture of different artifacts. `shared` is a common option, but packages can define and use any options they want.

Options are defined in the package recipe, including their supported values, while other can be defaulted by the package recipe itself. A package for a library can well define that by default it will be a static library (a typical default). If not specified other. the package will be static.

There are some exceptions to the above. For example, settings can be defined per-package using the command line:

```
$ conan install . -s mypkg:compiler=gcc -s compiler=clang ..
```

This will use `gcc` for "mypkg" and `clang` for the rest of the dependencies (extremely rare case).

There are situations whereby many packages use the same option, thereby allowing you to set its value once using patterns, like:

```
$ conan install . -o *:shared=True
```

20.2.3 Can Conan use git repositories as package servers?

Or put it with other words, can a conan recipe define requirements something like `requires="git://github.com/someuser/somerepo.git#sometag"`?

No, it is not possible. There are several technical reasons for this, mainly around the dependency resolution algorithm, but also about performance:

- Conan manages dependency versions conflicts. These can be efficiently handled from the abstract reference quickly, while a git repo reference would require cloning contents even before deciding.
- The version overriding mechanism from downstream consumers to resolve conflicts cannot be implemented either with git repos, as both the name and the version of the package is not defined.
- Conan support version-ranges, like depending on `boost/[>1.60 <1.70]`. This is basically impossible to implement in git repos.
- Conan has an "update" concept, that allows to query servers for latest modifications, latest versions, or even latest revisions, which would not work at all with git repos either.
- Binary management is one of the biggest advantages of Conan. Obviously, it is not possible to manage binaries for this case either.

In summary, whatever could be done would be an extremely limited solution, very likely inefficient and much slower, with a lot of corner cases and rough edges around those said limitations. It would require a big development effort, and the compounded complexity it would induce in the codebase is a liability that will slow down future development, maintenance and support efforts.

Besides the impossibility on the technical side, there are also other reasons like well known best practices around package management and modern devops in other languages that show evidence that even if this approach looks like convenient, it should be discouraged in practice:

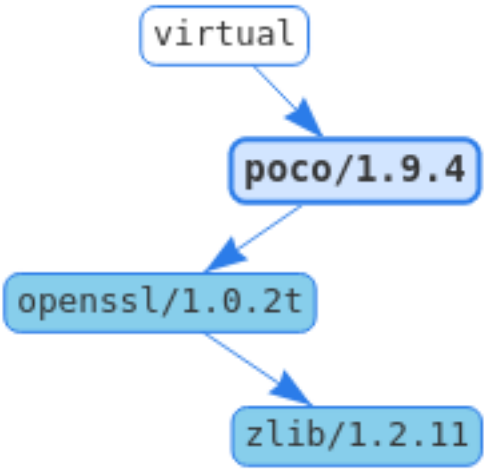
- Packages should be fully relocatable to a different location. Users should be able to retrieve their dependencies and upload a copy to their own private server, and fully disconnect from the external world. This is critical for robust and secure production environments, and avoid problems that other ecosystems like NPM have had in the past. As a consequence, all recipes dependencies should not be coupled to any location, and be abstract as conan "requires" are.
- Other languages, like Java (which would be the closest one regarding enterprise-ness), never provided this feature. Languages like golang, that based its dependency management on this feature, has also evolved away from it and towards abstract "module" concepts that can be hosted in different servers

So there are no plans to support this approach, and the client-server architecture will continue to be the proposed solution. There are several alternatives for the servers from different vendors, for public open source packages ConanCenter is the recommended one, and for private packages, the free Artifactory CE is a simple and powerful solution.

20.2.4 How to obtain the dependents of a given package?

The search model for Conan in commands such as **`conan install`** and **`conan info`** is done from the downstream or "consumer" package as the starting node of the dependency graph and upstream.

```
$ conan info poco/1.9.4@
```



The inverse model (from upstream to downstream) is not simple to obtain for Conan packages. This is because the dependency graph is not unique, it changes for every configuration. The graph can be different for different operating systems or just by changing some package options. So you cannot query which packages are dependent on `my_lib/0.1@user/channel`, but which packages are dependent on `my_lib/0.1@user/channel:63da998e3642b50bee33` binary package. Also, the response can contain many different binary packages for the same recipe, like `my_dependent/0.1@user/channel:packageID1... ID2... my_dependent/0.1@user/channel:packageIDN`. That is the reason why **`conan info`** and **`conan install`** need a profile (default profile or one given with `--profile`) or installation files `conanbuildinfo.txt` to look for settings and options.

In order to show the inverse graph model, the bottom node is needed to build the graph upstream and an additional node too to get the inverse list. This is usually done to get the build order in case a package is updated. For example, if we want to know the build order of the Poco dependency graph in case OpenSSL is changed we could type:

```
$ conan info poco/1.9.4@ -bo openssl/1.0.2t
WARN: Usage of `--build-order` argument is deprecated and can return wrong results. Use␣
↪`conan lock build-order ...` instead.
[openssl/1.0.2t], [poco/1.9.4]
```

If OpenSSL is changed, we would need to rebuild it (of course) and rebuild Poco.

20.2.5 Packages got outdated when uploading an unchanged recipe from a different machine

Usually this is caused due to different line endings in Windows and Linux/macOS. Normally this happens when Windows uploads it with CRLF while Linux/macOS do it with only LF. Conan does not change the line endings to not interfere with user. We suggest always using LF line endings. If this issue is caused by git, it could be solved with **`git config --system core.autocrlf input`**.

The *outdated* status is computed from the recipe hash, comparing the hash of the recipe used to create a binary package and the current recipe. The recipe hash is the hash of all the files included in the *conanmanifest.txt* file (you can inspect this file in your cache with **`conan get <ref> conanmanifest.txt`**). The first value in the manifest file is a timestamp and is not taken into account to compute the hash. Checking and comparing the contents of the different *conanmanifest.txt* files in the different machines can give an idea of what is changing.

If you want to make the solution self-contained, you can add a *.git/config* file in your project that sets the `core.autocrlf` property (for the whole repo), or if you need a per-file configuration, you could use the *.gitattributes* file to set the `text eol=lf` for every file you want.

20.2.6 Is there any recommendation regarding which <user> or <channel> to use in a reference?

A Conan reference is defined by the following template: `<library-name>/<library-version>@<user>/<channel>`

The `<user>` term in a Conan reference is basically a namespace to avoid collisions of libraries with the same name and version in the local cache and in the same remote. This field is usually populated with the author's name of the package recipe (which could be different from the author of the library itself) or with the name of the organization creating it. Here are some examples from Conan Center:

```
OpenSSL/1.1.1@conan/stable
CLI11/1.6.1@cliutils/stable
CTRE/2.1@ctre/stable
Expat/2.2.5@pix4d/stable
FakeIt/2.0.5@gasuketsu/stable
Poco/1.9.0@pocoproject/stable
c-blosc/v1.14.4@francescalted/stable
```

In the case of the `<channel>` term, normally OSS package creators use `testing` when developing a recipe (e.g. it compiles only in few configurations) and `stable` when the recipe is ready enough to be used (e.g. it is built and tested in a wide range of configurations).

It is strongly recommended that packages are considered immutable. Once a package has been created with a user/channel, it shouldn't be changed. Instead, a new package with a new user/channel should be created.

20.2.7 What does "outdated from recipe" mean exactly?

In some output or commands there are references to "outdated" or "outdated from recipe". For example, there is a flag **`--outdated`** in **`conan search`** and **`conan remove`** to filter by outdated packages.

When packages are created, Conan stores some metadata of the package such as the settings, the final resolution of the dependencies... and it also saves the recipe hash of the recipe contents they were generated with. This way Conan is able to know the real relation between a recipe and a package.

Basically outdated packages appear when you modify a recipe and export and/or upload it, without re-building binary packages with it. This information is displayed in yellow with:

```
$ conan search pkg/0.1@user/channel --table=file.html
# open file.html
# It will show outdated binaries in yellow.
```

This information is important to know if the packages are up to date with the recipe or even if the packages are still "accessible" from the recipe. That means: if the recipe has completely removed an option (it could be a setting or a requirement) but there are old packages that were generated previously with that option, those packages will be impossible to install as their package ID are calculated from the recipe file (and that option does not exist anymore).

When using "revisions" (it is opt-in in Conan 1.X, but it will be always enabled in Conan 2.0), this should never happen, as doing any change to a recipe or source should create a new revision that will contain its own binaries.

20.2.8 How to configure the remotes priority order

The lookup remote order is defined by the command **conan remote**:

```
$ conan remote list
conancenter: https://center.conan.io [Verify SSL: True]
myremote: https://MyTeamServerIP:8081/artifactory/api/conan/myremote [Verify SSL: True]
```

As you can see, the remote `conancenter` is listed on index **0**, which means it has the highest priority when searching or installing a package, followed by `myremote`, on index **1**. To update the index order, the argument `--insert` can be added to the command **conan remote update**:

```
$ conan remote update myremote https://MyTeamServerIP:8081/artifactory/api/conan/
↪myremote --insert
$ conan remote list
myremote: https://MyTeamServerIP:8081/artifactory/api/conan/myremote [Verify SSL: True]
conancenter: https://center.conan.io [Verify SSL: True]
```

The `--insert` argument means *index 0*, the highest priority, thus the `myremote` remote will be updated as the first remote to be used.

It's also possible to define a specific index when adding a remote to the list:

```
$ conan remote add otherremote https://MyCompanyOtherIP:8081/artifactory/api/conan/
↪otherremote --insert 1
$ conan remote list
myremote: https://MyTeamServerIP:8081/artifactory/api/conan/myremote [Verify SSL: True]
otherremote: https://MyCompanyOtherIP:8081/artifactory/api/conan/otherremote [Verify␣
↪SSL: True]
conancenter: https://center.conan.io [Verify SSL: True]
```

The `otherremote` remote needs to be added after `myremote`, so we need to set the remote index as **1**.

20.3 Troubleshooting

20.3.1 ERROR: The recipe is constraining settings

When you install or create a package you might have error like the following one:

```
ERROR: The recipe wtl/10.0.9163 is constraining settings. Invalid setting 'Linux' is not␣
↪a valid 'settings.os' value.
Possible values are ['Windows']
Read "http://docs.conan.io/en/latest/faq/troubleshooting.html#error-the-recipe-is-
↪contraining-settings"
```

This means that your target operating system is not supported by the recipe.

20.3.2 ERROR: Missing prebuilt package

When installing packages (with **conan install** or **conan create**) it is possible that you get an error like the following one:

```
 WARN: Can't find a 'czmq/4.2.0' package for the specified settings, options and␣
↪dependencies:
- Settings: arch=x86_64, build_type=Release, compiler=Visual Studio, compiler.runtime=MD,
↪ compiler.version=16, os=Windows
- Options: shared=False, with_libcurl=True, with_libuuid=True, with_lz4=True,␣
↪libcurl:shared=False, ...
- Dependencies: openssl/1.1.1d, zeromq/4.3.2, libcurl/7.67.0, lz4/1.9.2
- Requirements: libcurl/7.Y.Z, lz4/1.Y.Z, openssl/1.Y.Z, zeromq/4.Y.Z
- Package ID: 7a4079899e0893ca670df1f682b4606abe79ee5b

ERROR: Missing prebuilt package for 'czmq/4.2.0@'
Use 'conan search czmq/4.2.0@ --table=table.html -r=remote' and open the table.html file␣
↪to see available packages
Or try to build locally from sources with '--build=czmq'

More Info at 'https://docs.conan.io/en/latest/faq/troubleshooting.html#error-missing-
↪prebuilt-package'
```

This means that the package recipe `czmq/4.2.0@` exists, but for some reason there is no precompiled package for your current settings. Maybe the package creator didn't build and shared pre-built packages at all and only uploaded the package recipe, or they are only providing packages for some platforms or compilers. E.g. the package creator built packages from the recipe for Visual Studio 14 and 15, but you are using Visual Studio 16. Also you may want to check your *package ID mode* as it may have an influence on the packages available for it.

By default, Conan doesn't build packages from sources. There are several possibilities to overcome this error:

- You can try to build the package for your settings from sources, indicating some build policy as argument, like **--build czmq** or **--build missing**. If the package recipe and the source code work for your settings you will have your binaries built locally and ready for use.
- If building from sources fails, you might want to fork the original recipe, improve it until it supports your configuration, and then use it. Most likely contributing back to the original package creator is the way to go. But you can also upload your modified recipe and pre-built binaries under your own username too.

20.3.3 ERROR: Invalid setting

It might happen sometimes, when you specify a setting not present in the defaults that you receive a message like this:

```
$ conan install . -s compiler.version=4.19 ...

ERROR: Invalid setting '4.19' is not a valid 'settings.compiler.version' value.
Possible values are ['4.4', '4.5', '4.6', '4.7', '4.8', '4.9', '5.1', '5.2', '5.3', '5.4
↪', '6.1', '6.2']
Read "http://docs.conan.io/en/latest/faq/troubleshooting.html#error-invalid-setting"
```

This doesn't mean that such architecture is not supported by conan, it is just that it is not present in the actual defaults settings. You can find in your user home folder `~/.conan/settings.yml` a settings file that you can modify, edit, add any setting or any value, with any nesting if necessary. See *Customizing settings*.

As long as your team or users have the same settings (you can share with them the file), everything will work. The *settings.yml* file is just a mechanism so users agree on a common spelling for typical settings. Also, if you think that some settings would be useful for many other conan users, please submit it as an issue or a pull request, so it is included in future releases.

It is possible that some build helper, like `CMake` will not understand the new added settings, don't use them or even fail. Such helpers as `CMake` are simple utilities to translate from conan settings to the respective build system syntax and command line arguments, so they can be extended or replaced with your own one that would handle your own private settings.

20.3.4 ERROR: Setting value not defined

When you install or create a package, it is possible to see an error like this:

```
ERROR: hello/0.1@user/testing: 'settings.arch' value not defined
```

This means that the recipe defined `settings = "os", "arch", ...` but a value for the `arch` setting was not provided either in a profile or in the command line. Make sure to specify a value for it in your profile, or in the command line:

```
$ conan install . -s arch=x86 ...
```

If you are building a pure C library with gcc/clang, you might encounter an error like this:

```
ERROR: hello/0.1@user/testing: 'settings.compiler.libcxx' value not defined
```

Indeed, for building a C library, it is not necessary to define a C++ standard library. And if you provide a value, you might end with multiple packages for exactly the same binary. What has to be done is to remove such subsetting in your recipe:

```
def configure(self):
    del self.settings.compiler.libcxx
```

20.3.5 ERROR: Failed to create process

When conan is installed via pip/PyPI, and python is installed in a path with spaces (like many times in Windows "C:/Program Files…"), conan can fail to launch. This is a known python issue, and can't be fixed from conan. The current workarounds would be:

- Install python in a path without spaces
- Use virtualenvs. Short guide:

```
$ pip install virtualenvwrapper-win # virtualenvwrapper if not Windows
$ mkvirtualenv conan
(conan) $ pip install conan
(conan) $ conan --help
```

Then, when you will be using conan, for example in a new shell, you have to activate the virtualenv:

```
$ workon conan
(conan) $ conan --help
```

Virtualenvs are very convenient, not only for this workaround, but to keep your system clean and to avoid unwanted interaction between different tools and python projects.

20.3.6 ERROR: Failed to remove folder (Windows)

It is possible that operating conan, some random exceptions (some with complete tracebacks) are produced, related to the impossibility to remove one folder. Two things can happen:

- The user has some file or folder open (in a file editor, in the terminal), so it cannot be removed, and the process fails. Make sure to close files, specially if you are opening or inspecting the local conan cache.
- In Windows, the Search Indexer might be opening and locking the files, producing random, difficult to reproduce and annoying errors. Please **disable the Windows Search Indexer for the conan local storage folder**

20.3.7 ERROR: Error while initializing Options

When installing a Conan package and the follow error occurs:

```
ERROR: conanfile.py: Error while initializing options. Please define your default_
↪options as list or multiline string
```

Probably your Conan version is outdated. The error is related to *default_options* be used as dictionary and only can be handled by Conan >= 1.8. To fix this error, update Conan to 1.8 or higher.

20.3.8 ERROR: Error while starting Conan Server with multiple workers

When running `gunicorn` to start `conan_server` in an empty environment:

```
$ gunicorn -b 0.0.0.0:9300 -w 4 -t 300 conans.server.server_launcher:app

    ********************************************
    *                                          *
    *      ERROR: STORAGE MIGRATION NEEDED!    *
```

(continues on next page)

(continued from previous page)

```
    *                                              *
    ************************************************
    A migration of your storage is needed, please backup first the storage directory and␣
↪run:

    $ conan_server --migrate
```

Conan Server will try to create *~/.conan_server/data*, *~/.conan_server/server.conf* and *~/.conan_server/version.txt* at first time. However, as multiple workers are running at same time, it could result in a conflict. To fix this error, you should run:

```
$ conan_server --migrate
```

This command must be executed before to start the workers. It will not migrate anything, but it will populate the conan_server folder. The original discussion about this error is here.

20.3.9 ERROR: Requested a package but found case incompatible

When installing a package which is already installed, but using a different case, will result on the follow error:

```
$ conan install poco/1.10.1@

    [...]
    ERROR: Failed requirement 'openssl/1.0.2t' from 'poco/1.10.1@'
    ERROR: Requested 'openssl/1.0.2t', but found case incompatible recipe with name
↪'OpenSSL' in the cache. Case insensitive filesystem can not manage this
    Remove existing recipe 'OpenSSL' and try again.
```

You can find and use recipes with upper and lower case names (we encourage lowercase variants), but some OSs like Windows are case insensitive by default, they cannot store at the same time both variants in the Conan cache. To solve this problem you need to remove existing upper case variant `OpenSSL`:

```
$ conan remove "OpenSSL/*"
```

20.3.10 ERROR: Incompatible requirements obtained in different evaluations of 'requirements'

When two different packages require the same package as a dependency, but with different versions, will result in the following error:

```
$ cat conanfile.txt

[requires]
baz/1.0.0
foobar/1.0.0

$ conan install conanfile.txt

[...]
WARN: foobar/1.0.0: requirement foo/1.3.0 overridden by baz/1.0.0 to foo/1.0.0
```

(continues on next page)

(continued from previous page)

```
ERROR: baz/1.0.0: Incompatible requirements obtained in different evaluations of
↪'requirements'
Previous requirements: [foo/1.0.0]
New requirements: [foo/1.3.0]
```

As we can see in the following situation: the `conanfile.txt` requires 2 packages (`baz/1.0.0` and `foobar/1.0.0`) which both require the package named `foo`. However, `baz` requires `foo/1.0.0`, but `foobar` requires `foo/1.3.0`. As the required versions are different, it's considered a conflict and Conan will not solve it.

To solve this kind of collision, you have to choose a version for `foo` and add it to the `conanfile.txt` as an explicit requirement:

```
[requires]
foo/1.3.0
baz/1.0.0
foobar/1.0.0
```

Here we choose `foo/1.3.0` because is newer. Now we can proceed:

```
$ conan install conanfile.txt

    [...]
    WARN: baz/1.0.0: requirement foo/1.0.0 overridden by foobar/1.0.0 to foo/1.3.0
```

Conan still warns us about the conflict, but as we have *Dependencies overriding* the `foo` version, it's no longer an error.

20.3.11 ERROR: HTTPSConnectionPool(host='conan.bintray.com', port=443)

The `conan.bintray.com` has been deprecated and you have to update to `https://center.conan.io` now. For more information, please, read Old Bintray remote EOL.

CHAPTER

TWENTYONE

GLOSSARY

binary package
Output binary usually obtained with a *conan create* command applying settings and options as input. Usually, there are N binary packages inside one Conan package, one for each set of settings and options. Every binary package is identified by a package_id.

build helper
A build helper is a Python script that translates Conan settings to the specific settings of a build tool. For example, in the case of CMake, the build helper sets the CMake flag for the generator from Conan settings like the compiler, operating system, and architecture. Conan provides integration for several build tools such as *CMake*, *Autotools*, *MSBuild* or *Meson*. You can also integrate your preferred build system in Conan if it is not available by default.

build system
Tools used to automate the process of building binaries from sources. Some examples are Make, Autotools, SCons, CMake, Premake, Ninja or Meson. Conan has integrations with some of these build systems using *generators* and *build helpers*.

conanfile
Can refer to either *conanfile.txt* or *conanfile.py* depending on what's the context it is used in.

conanfile.py
The file that defines a Conan recipe that is typically used to create packages, but can be used also to consume packages only (see conanfile.txt). Inside of this recipe, it is defined (among other things) how to download the package's source code, how to build the binaries from those sources, how to package the binaries and information for future consumers on how to consume the package.

conanfile.txt
It is a simplified version of the *conanfile.py* used only for consuming packages. It defines a list of packages to be consumed by a project and can also define the *generators* for the build system we are using, and if we want to *import* files from the dependencies, as shared libraries, executables or assets.

cross compiler
A cross compiler is a compiler capable of creating an executable intended to run in a platform different from the one in which the compiler is running.

dependency graph
A directed graph representing dependencies of several Conan packages towards each other. The relations between the packages are declared with the *requirements* in the recipes. A dependency graph in Conan depends on the input profile applied because the requirements can be *conditioned* to a specific configuration.

editable package
A *package* that resides in the user workspace, but is consumed as if it was in the cache. This mode is useful when you are developing the packages, and the projects that consume them at the same time.

generator
A generator provides the information of dependencies calculated by Conan in a suitable format that is usually

injected in a build system. They normally provide a file that can be included or passed as input to the specific build system to help it to find the packages declared in the recipe. There are other generators that are not intended to be used with the build system. e.g. *"deploy"*, *"YouCompleteMe"*.

hook
: *Conan Hooks* are Python scripts containing functions that will be executed before and after a particular task performed by the Conan client. Those tasks could be Conan commands, recipe interactions such as exporting or packaging, or interactions with the remotes. For example, you could have a hook that checks that the recipe includes attributes like license, url and description.

library
: A library is a collection of code and resources to be reused by other programs.

local cache
: A folder in which Conan stores the package cache and some configuration files such as the *conan.conf* or *settings.yml*. By default, this file will be located in the user home folder **~/.conan/** but it's configurable with the environment variable `CONAN_USER_HOME`. In some scenarios like CI environments or when using per-project management and storage changing the default conan cache location *could be useful*.

lockfile
: Files that store the information with the exact versions, revisions, options, and configuration of a dependency graph. They are intended to make the building process reproducible even if the dependency definitions in conanfile recipes are not fully deterministic.

options
: *Options* are declared in the recipes, it is similar to the *setting* concept but it is something that can be defaulted by the recipe creator, like if a library is static or shared. Options are specific to each package (there is not a yml file like the *settings.yml* file), and each package creator can define their options "header_only" for example. The most common example is the "shared" option, with possibles values *True/False* and typically defaulted to *False*.

package
: A Conan package is a collection of files that include the recipe and the N binary packages generated for different configurations and settings. It can contain binary files such as libraries, headers or tools to be reused by the consumer of the package.

package ID
: The package id is a hash of the settings options and requirements used to identify the binary packages. Applying different profiles to the *conan create* command, it will generate different package IDs. e.g: Windows, x86, shared...

package reference
: A package reference is the combination of the recipe reference and the package ID. It adopts the form of `name/version@user/channel:package_id_hash`.

package revision
: A unique ID using the checksum of the package (all files stored in a binary package). See the *revisions mechanism* page.

profile
: *A profile* is the set of different settings, options, environment variables and tool requirements used when working with packages. The settings define the operating system, architecture, compiler, build type, and C++ standard. Options define, among other things, if dependencies are linked in shared or static mode or other compile options.

recipe
: Python script defined in a *conanfile.py* that specifies how the package is built from sources, what the final binary artifacts are, the package dependencies, etc.

recipe reference
: A recipe reference is the combination of the package name, version, and two optional fields named user and chan-

nel that could be useful to identify a forked recipe from the community with changes specific to your company. It adopts the form of `name/version@user/channel`.

recipe revision
A unique ID using the latest VCS hash or a checksum of the *conanfile.py* with the exported files if any. See the *revisions mechanism* page.

remote
The binary repository that hosts Conan packages inside a server.

requirement
Packages on which another package depends on. They are represented by a conan reference: *lib/1.0@*

revision
It is the *mechanism* to implicitly version the changes done in a recipe or package without bumping the actual reference or package version.

semantic versioning
Versioning system with versions in the form of `MAJOR.MINOR.PATCH` where `PATCH` version changes when you make backward-compatible bug fixes, `MINOR` version changes when you add functionality in a backward-compatible manner, and `MAJOR` version changes when you make incompatible API changes. Conan uses semantic versioning by default but this behavior can be *easily configured and changed* in the `package_id()` method of your conanfile, and any versioning scheme you desire is supported.

settings
A set of keys and values, like *os*, *compiler* and *build_type* that are declared at the *~/.conan/settings.yml* file.

shared library
A library that is loaded at runtime into the target application.

static library
A library that is copied at compile time to the target application.

system packages
System packages are packages that are typically installed system-wide via system package management tools such as apt, yum, pkg, pkgutil, brew or pacman. It is possible to install *system-wide packages methods* from Conan adding a `system_requirements()` method to the conanfile.

tool requirement
Requirements that are only needed when you need to build a package (that declares the *tool requirement*) from sources, but if the binary package already exists, the build-require is not retrieved.

toolchain
A toolchain is the set of tools usually intended for compiling, debugging and profiling applications.

transitive dependency
A dependency that is induced by the dependency that the program references directly. Imagine that your project uses the **Poco** library that needs the **OpenSSL** library, and **OpenSSL** is calling to the zlib library. In this case, **OpenSSL** and **zlib** would be transitive dependencies.

workspace
Conan workspaces allow us to have more than one package in user folders and have them directly use other packages from user folders without needing to put them in the local cache. Furthermore, they enable incremental builds on large projects containing multiple packages.

CHAPTER

TWENTYTWO

CHANGELOG

Check https://github.com/conan-io/conan for issues and more details about development, contributors, etc.

Important: Conan 1.49 shouldn't break any existing 1.0 recipe or command line invocation. If it does, please submit a report on GitHub. Read more about the *Conan stability commitment*.

22.1 1.49.0 (02-Jun-2022)

- Feature: Add *install_substitutes* to system package manager tools to be able to install sets of packages that are equivalent with different names for different distros. #11367 . Docs here
- Feature: Do not automatically fix the shared libraries to add the rpath in Apple and add an external tool *tools.apple.fix_apple_shared_install_name* to do it optionally in recipes for packages that do not set the correct *LC_ID_DYLIB*. #11365 . Docs here
- Feature: Allow pyyaml 6.0 dependency. #11363
- Feature: Removed Python 2.7 support, as a result of an unsolvable security vulnerability in pyjwt. #11357 . Docs here
- Feature: The `conanfile.txt` file now accepts a `[layout]` that can be filled with 3 predefined layouts: `cmake_layout`, `vs_layout` and `bazel_layout`. #11348 . Docs here
- Feature: Remove the parameter `copy_symlink_folders` of the `conan.tool.files.copy` function and now, any symlink file pointing to a folder will be treated as a regular file. #11330 . Docs here
- Feature: Tools *can_run* validates if it is possible to run a and application build for a non-native architecture. #11321 . Docs here
- Feature: Add *CMAKE_SYSROOT* support for *CMakeToolchain*. #11317 . Docs here
- Feature: Add *–sysroot* support for *AutotoolsToolchain* and remove support for *cpp_info.sysroot* in *AutotoolsDeps*. #11317 . Docs here
- Feature: Add *tools.build:sysroot* conf. #11317 . Docs here
- Feature: Improved *cmake_layout* and *CMakePresets.json* feature so you can manage different configurations using the same *CMakeUserPresets.json* not only for multi-config (Debug/Release) but for any set of settings specified in a new conf *tools.cmake.cmake_layout:build_folder_vars* that accepts a list of settings to use. e.g *tools.cmake.cmake_layout:build_folder_vars=["settings.compiler", "options.shared"]* #11308 . Docs here
- Feature: Adds GCC 9.4 in the list of compilers supported in the settings file. #11296
- Feature: Raise an error when running CMake if CMAKE_BUILD_TYPE is not defined and the generator is not multi-config. #11294 . Docs here

- Feature: Implement a `check_min_vs()` checker that will work for both `Visual Studio` and `msvc` to allow migration from 1.X to 2.0 #11292 . Docs here
- Feature: More flexibility in Autotools tools to override arguments and avoid all default arguments for *make*, *autoreconf* and *configure*. #11284 . Docs here
- Feature: Add components support in XcodeDeps. #11233 . Docs here
- Feature: Define new `tools.cmake.cmaketoolchain:toolset_arch` to define VS toolset x64 or x86 architecture #11147 . Docs here
- Feature: Add new *xtensalx7* option for the *arch_target* and *arch* settings, allowing targeting Espressif's ESP32-S2 and ESP32-S3 microcontrollers. #11143
- Fix: Use *interface_library* with *shared_library* on Windows in BazelDeps. #11355
- Fix: BazelDeps generator cannot find a lib when it's named with the basename of the lib file. #11343
- Fix: Avoid empty paths in run environments PATH, LD_LIBRARY_PATH, DYLD_LIBRARY_PATH env-vars. #11298
- Fix: Use *DESTDIR* argument in *make install* step instead of using the *–prefix* in configure. #11284 . Docs here
- Fix: Add *-DCMAKE_BUILD_TYPE* to markdown generator instructions for CMake single config. #11220
- Fix: Fixing `--require-override` over conditional `requirements()` method. #11209
- Fix: Placing quote marks around echo statement in *save_sh* function. #11123
- Bugfix: Force `conan_server` to use `pyjwt>=2.4.0` to solve a known vulnerability. #11350
- Bugfix: Fix case where CMakeDeps generator may use the wrong dependency name for transitive dependencies. #11307
- Bugfix: Link `cpp_info.objects` first in `CMakeDeps` generator, as they can have dependencies to `system_libs` that need to be after them in some platforms to correctly link. #11272
- Bugfix: Update cmake_layout generators folder to honor os path format. #11252
- Bugfix: Catching *KeyError* if *USERNAME* is not set as env variable on Windows. #11223
- Bugfix: Add Rocky Linux to with_yum. #11212

22.2 1.48.1 (18-May-2022)

- Fix: Add *-DCMAKE_BUILD_TYPE* to markdown generator instructions for CMake single config. #11234
- Bugfix: Fix case where CMakeDeps assumes a module dependency when transitive dependencies do not define *cmake_find_mode* and fallback to a config one. #11240
- Bugfix: Fixed broken `apple-clang` 13.0 compatibility. #11231

22.3 1.48.0 (03-May-2022)

- Feature: Do not recommend to set *PKG_CONFIG_PATH* in markdown generator any more as it is already set by the AutotoolsToolchain. #11120
- Feature: The *cmake_layout* declares *folders.generators = "build/generators"* that is common for different configurations, enabling *CMakePresets.json* to support different *configurePresets* and *buildPresets* for single-config and multi-config generators. #11117 . Docs here
- Feature: The *CMakeToolchain* generator will create (if missing) a *CMakeUserPresets.json* if the *CMakeLists.txt* file is found in the root folder of the project. That file will include automatically the *CMakePresets.json* file contained at *build/generators* folder to allow CMake and IDEs to locate automatically the presets generated by Conan. #11117 . Docs here
- Feature: The *CMAKE_BUILD_TYPE* is not adjusted in the *conan_toolchain.cmake* anymore, the configuration is moved to the *CMakePresets.json* preparing Conan to support multiple "single-config" configurations in one *CMakePresets.json* file #11112 . Docs here
- Feature: The *CMakePresets.json* (file generated when specified the *CMakeToolchain* generator) is appended with a new *buildPresets* after a **`conan install`** with a different *build_type* when using a multiconfiguration generator like *Visual Studio*. #11103 . Docs here
- Feature: Removed *PlatformToolset* and added *conantoolchain.props* from **.vcxproj* file. #11101
- Feature: Append the folder where the Conan generators were installed to *PKG_CONFIG_PATH* in AutotoolsToolchain. #11063 . Docs here
- Feature: Adding new `tools.env.virtualenv:powershell` conf to opt-in to generate and use powershell scripts instead of .bat ones. #11061 . Docs here
- Feature: Added new configuration fields: *tools.apple:enable_bitcode*, *tools.apple:enable_arc* and *tools.apple:enable_visibility*. #10985 . Docs here
- Feature: Added new mechanism to inject common Xcode flags in *CMakeToolchain* generator if enabled Bitcode, ARC, or Visibility configurations. #10985 . Docs here
- Feature: New templates for autotools exe and lib. #10978 . Docs here
- Feature: Change *build_script_folder* argument from the configure to the *Autotools* build helper constructor. #10978 . Docs here
- Feature: Replaced *add_definitions* by *add_compile_definitions* in *conan.tools.cmake.** package. #10974
- Feature: Added *cxxflags*, *cflags*, and *ldflags* attributes to *MSBuildToolchain*. #10972 . Docs here
- Feature: Added mechanism to inject extra flags to *MSBuildToolchain* via *[conf]*. #10972 . Docs here
- Fix: Allow `Version(self.settings.compiler.version)` to work for `new` tools.scm.Version``. #11119
- Fix: Make shared libraries build with Autotools relocatable in Macos by patching the install name (LC_ID_DYLIB) and setting to *@rpath/libname.dylib*. #11114 . Docs here
- Fix: using CMAKE_PROJECT_INCLUDE instead of presets to define variables that don't work in toolchains #11098
- Fix: Fix quote escaping for defines in pkg_config generator. #11073
- Fix: Fix quote escaping for defines in PkgConfigDeps generator. #11073
- Fix: Quote *add_compile_definitions* correctly in CMakeToolchain. #11057
- Fix: Escape quotes in definitions in CMakeToochain. #11057

- Fix: Renamed *self.base_source_folder* to *self.export_source_folder*. That variable was introduced to reference the folder where the *export_sources* are. Currently, they are copied to the *source* folder but might be changed in the future to avoid copying them, so *self.export_source_folder* will always point to the folder containing the *exports_sources*. #11055 . Docs here
- Fix: Ensure correct order for libraries in AutotoolsDeps. #11054
- Fix: Escape quotes in XCodeDeps generator. #11039
- Fix: The *CMakeDeps* generator now set *INTERFACE_LINK_DIRECTORIES* necessary when using auto link *'#pragma comment(lib, "foo")' ` when the required library sets the property `cmake_set_interface_link_directories*. #10984 . Docs here
- Fix: Renamed variables from the *CMakeToolchain* context in blocks to be all lowercase. e.g: *CMAKE_OSX_ARCHITECTURES* to *cmake_osx_architectures*. #10981
- Bugfix: Avoid `BazelDeps` to find a library when a directory with the same name exists. #11090
- Bugfix: The *binaryDir* field at *CMakePresets.json* is not set if the *conanfile.build_folder* is not available, avoiding a *null* value breaking the specification. #11088
- Bugfix: Fixed unziping while using *tools.get* or *tools.unzip* with the *strip_root=True* in a *tgz* file with hardlinks inside. #11074
- Bugfix: The method *get_commit* from the new *conan.tools.scm.Git* was capturing a wrong commit, for example, ignoring commits in subfolders when checking the parent folder. #11015
- Bugfix: The *json* generator was showing "None" in the *version* field of the dependencies when the *layout()* method was used. #10960
- Bugfix: The config *default_python_requires_id_mode=unrelated_mode* raised an error, it has been fixed. #10959
- Bugfix: The CMakeToolchain now declares *CACHE BOOL* variables when a bool is stored in a variable: *toolchain.variables["FOO"] = True*. #10941

22.4 1.47.0 (31-Mar-2022)

- Feature: Renamed *tools.build:cppflags* to *tools.build:defines* and removed *tools.build:ldflags* (now it's the union between *tools.build:sharedlinkflags* and *tools.build:exelinkflags* in mostly all the cases). #10928 . Docs here
- Feature: Adding cross-compilation for Android in MesonToolchain. #10908 . Docs here
- Feature: Add extra flags via [conf] into XcodeToolchain. #10906 . Docs here
- Feature: Preliminar support for CMakePresets.json. #10903 . Docs here
- Feature: Added *wrap_mode=nofallback* project-option into *MesonToolchain* as default value. #10884
- Feature: Added basic *rmdir* tool at *conan.tools.files*. #10874 . Docs here
- Feature: Backport of 2.0 compatibility() recipe method. #10868
- Feature: Add detection in meson toolchain for cross conditions and requirement of needs_exe_wrapper. Users may specify the exe wrapper in their meson.build now. #10846
- Feature: Allow *tested_reference_str* to be *None*. #10834
- Feature: New templates for the **`conan new`** command with bazel examples: *bazel_exe* and *bazel_lib*. #10812 . Docs here
- Feature: Add some support for `msvc` compiler older versions. #10808

- Feature: Added mechanism to inject extra flags via *[conf]* into several toolchains like *AutotoolsToolchain*, *MesonToolchain* and *CMakeToolchain*. #10800 . Docs here
- Feature: Support selecting the target to build for XcodeBuild helper. #10799 . Docs here
- Feature: Support for Xcode 13.3, iOS 15.4, watchOS 8.5 and tvOS 15.4. #10797
- Feature: New modern "conan new –template=msbuild_lib|exe" for modern MSBuild VS integration. #10760 . Docs here
- Feature: Added a checker for Conan 2.x deprecated *from conans* imports in *pylint_plugin*. #10746 . Docs here
- Feature: New *conan.tool.microsoft.unix_path* to convert paths to any subsystem when using *conanfile.win_bash*. #10743 . Docs here
- Feature: New `from conan.errors import XXX` new namespace to be ready for 2.0. #10734 . Docs here
- Feature: Allow specifying *default_options = {"pkg/*:option": "value"}* or *default_options = {"pkg*:option": "value"}* instead of *default_options = {"pkg:option": "value"}* to make compatible recipes with 2.0. #10731 . Docs here
- Feature: Add Clang 15 to default settings. #10558
- Feature: When the layout is declared, the cwd() in the source() method will follow the declared *self.folders.source* but the *exports_sources* will still be copied to the base source folder. #10091 . Docs here
- Fix: Add support for `clang` to `msvc_runtime_flag()`. It requires defining `compiler.runtime = static/dynamic` definition, same as modern `msvc` compiler setting. #10898 . Docs here
- Fix: Generate the correct *–target* triple when building for Apple catalyst. #10880
- Fix: The "conan.tools.files.patch" will (by default) look for the patch files in the *self.base_source_folder* instead of *self.source_folder* but will apply them with *self.source_folder* as the base folder. #10875 . Docs here
- Fix: Setting *CMAKE_SYSTEM_PROCESSOR* for Apple cross-compiling including M1. #10856 . Docs here
- Fix: Do not overwrite values for *CMAKE_SYSTEM_NAME*, *CMAKE_SYSTEM_VERSION* and *CMAKE_SYSTEM_PROCESSOR* set from the [conf] or the user_toolchain. #10856 . Docs here
- Fix: Fix architecture translation from Conan syntax to build system in CMakeToolchain. #10856 . Docs here
- Fix: Several improvements of the Bazel integration (*Bazel*, *BazelToolchain*, *BazelDeps*), new functional tests in all platforms. #10812 . Docs here
- Fix: *Conf.get()* always returns *default* value if internal *conf_value.value* is *None*, i.e., it was unset. #10800 . Docs here
- Fix: Set *-DCMAKE_MAKE_PROGRAM* when the generator is for MinGW and the conf *tools.gnu:make_program* is set. #10770
- Fix: Remove unused `clion_layout`. #10760 . Docs here
- Fix: AutotoolsToolchain adjust the runtime flag of *msvc* (*MTd*, *MT*, *MDd* or *MD*) to *CFLAGS* and *CXXFLAGS* when using the *msvc* as *settings.compiler*. #10755 . Docs here
- Fix: Removed *subsystem_path* from the *conan.tool.microsoft* namespace, superseded by *unix_path*. #10743 . Docs here
- Fix: Show an error message at **`conan install`** if the `validate()` method raises `ConanInvalidConfiguration`. #10725
- BugFix: The *find_dependency* called internally in *CMakeDeps* generator to locate transitive dependencies now use the *MODULE/NO_MODULE* following the *cmake_find_mode* property when declared in the dependency. #10917
- Bugfix: Fix virtualrunenv wrong `build` scope. #10904

- BugFix: Make `self.folders.root` apply at package `conan install .` too. #10842
- Bugfix: Fix call to undefined function for markdown generator when components add *system_libs*. #10783
- Bugfix: solve `build_policy=never` bug when `conan export-pkg --force` and there already exists a package in the cache. #10738
- Bugfix: Add transitive dependencies to generated Bazel BUILD files. #10686

22.5 1.46.2 (18-Mar-2022)

- Bugfix: Fix deprecated imports checker line number. #10822
- Bugfix: Specifying compiler.version=13 for apple-clang raised a CMake error when using the old cmake generator. #10820

22.6 1.46.1 (17-Mar-2022)

- Feature: Added a checker for Conan 2.x deprecated from conans imports in pylint_plugin. #10811
- Feature: Add apple-clang 13 major version to settings. #10807
- Feature: Make apple-clang 13 version package-id compatible with 13.0. #10807
- Feature: Autodetect only major version for apple-clang profile starting in version 13. #10807
- Feature: Add clang 15 version to settings. #10807
- Bugfix: Fix call to undefined function for markdown generator when components add system_libs. #10810

22.7 1.46.0 (07-Mar-2022)

- Feature: Configuration field *tools.cmake.cmaketoolchain:user_toolchain* defined as list-like object #10729 . Docs here
- Feature: Prepare Conan for search remote repos with mix of 1.X and 2.0 binaries. Conan 1.X will not list binaries (`conan search <ref>`) stored in remote repos that were created with Conan 2.0. #10692
- Feature: Adding jinja rendering of global.conf config file. #10687 . Docs here
- Feature: Improve markdown generator instructions. #10673 . Docs here
- Feature: The *CMakeToolchain* and *AutotoolsToolchain* take into account the *cpp.package* info to set the output directories for libraries, executables, and so on when running *cmake.install* or *make install*. #10672 . Docs here
- Feature: Added *basic_layout*, removed *meson_layout* and added argument *src_folder* to *cmake_layout`as a shortcut for adjusting `conanfile.folders.source*. #10659 . Docs here
- Feature: Adding `self.base_source_folder` for `exports_sources` explicit layouts. #10654 . Docs here
- Feature: Adding `root` to layout model to allow conanfile.py in subfolders. #10654 . Docs here
- Feature: Added new property *component_version* for *PkgConfigDeps* and legacy *PkgConfig*. #10633 . Docs here
- Feature: Changed *.pc* file description field for components in *PkgConfigDeps*. #10633 . Docs here
- Feature: Add *sdk_version* setting for *Macos*, *iOS*, *watchOS* and *tvOS*. #10608 . Docs here

- Feature: Add new *XcodeBuild* build helper. #10608 . Docs here
- Feature: Add new *XcodeToolchain* helper. #10608 . Docs here
- Feature: Add `compiler.version` 12 for GCC in settings. #10595
- Feature: Introduce new `conan.tools.scm.Git` helper, for direct use in `export()` method to capture git url and commit, and to be used in `source()` method to clone and checkout a git repo. #10594 . Docs here
- Feature: New `from conan.tools.files import update_conandata()` helper to add data to `conandata.yml` in the `export()` method. #10586 . Docs here
- Feature: Add CMake variables, cli arguments and native build system arguments to new `conan.tools.cmake.CMake` helper. #10573 . Docs here
- Feature: Adding more functionality to *ConfDefinition* and *Conf*, something similar to *ProfileEnvironment* and *Environment* ones. #10537 . Docs here
- Feature: Port *conan.tools.Version* to Conan 1.X. #10536 . Docs here
- Feature: Port *conan.tools.build.cross_building* to Conan 1.X. #10536 . Docs here
- Feature: New *copy* tool at *conan.tools.files* namespace that will replace the *self.copy* in Conan 2.0. #10530 . Docs here
- Feature: Add `recipe_folder` to pylint plugin. #10527
- Fix: Pin Markupsafe==2.0.1 for py27 and upgrade Jinja2>3 for py3, after Markupsafe latest 2.1 release broke Jinja2 2.11. #10710
- Fix: Fixed templates for **`conan new`** with *–template cmake_lib* and *–template cmake_exe* to include Conan 2.0 compatible syntax. #10706
- Fix: Moved new tool *cross_building* from *conan.tool.cross_building* to *conan.tool.build* to match the location in develop2. #10706
- Fix: Don't compose folders in Xcode generator using *dep.package_folder*, now *cpp_info.bindirs*, *cpp_info.includedirs*, etc. are absolute. #10694
- Fix: When the *layout()* method is declared, the *self.package_folder* in the recipe is now available even when doing a *conan install . `, pointing to the specified output folder (*-of`) or the path of the conanfile if not specified. #10655
- Fix: Fix creation path of deactivate scripts. #10653
- Fix: Add support for *[tool_requires]* section in conanfile.txt. #10642
- Fix: When `layout()` is defined, the `exports` will not be considered sources anymore, but only the `exports_sources`. The `exports` are used exclusively by the recipe, but not as package source. #10625
- Fix: Make the `source()` method run inside the `self.source_folder`, in the same way `build()` runs in `self.build_folder`. But only for recipes that define the `layout()` method, to not break unless using `layout()`. #10612 . Docs here
- Fix: Remove the `--source-folder` new argument to `install` and `editable`, not necessary at the moment. #10590 . Docs here
- Fix: Fix conf *True* and *False* handling in *tools.system.package_manage* helpers. #10583
- Fix: Change the `CMakeToolchain` message to use `CMAKE_CURRENT_LIST_FILE`. #10552
- Fix: BazelDeps was generating invalid bazel files cause the static_library paths were absolute and not relative. #10484
- Fix: BazelDeps was crashing when generating packages without libs cause the context was not checking the array size. #10484

- Fix: Fix Premake test failing on Linux because the Premake executable isn't found. #10250
- Bugfix: Fix `MesonToolchain` extra quotes in `cpp_std` #10707
- Bugfix: Add missing `system_libs` management in `AutotoolsDeps`. #10681
- Bugfix: *GnuDepsFlags* attributes like *frameworks* and *frameworkdirs* are only available for Apple OS. #10675
- Bugfix: Remove tmp folders created in Conan while checking the output of a command and detecting the compiler. #10663
- Bugfix: Fix `conan_server` circular import do to `conan.tools` namespace. #10635
- bugfix: Fix `meson_layout()` issue with shared folders. #10600
- Bugfix: Fix `SystemPackageTool` when `mode=verify`, it was still installing packages. #10596
- Bugfix: `self.build_folder` not being computed even if `layout()` method is defined in local **`conan install`**. Close https://github.com/conan-io/conan/issues/10566 #10567
- Bugfix: Fix conan_manifest.txt parse error when the filename has ":" in it. #10492
- Bugfix: Use meson toolchain file from install folder. #8965

22.8 1.45.0 (02-Feb-2022)

- Feature: Add system.package_manager tools to *conan config list*. #10469 . Docs here
- Feature: Use system package manager helpers *from conan.tools.system.package_manager*. #10467 . Docs here
- Feature: Add `[tool_requires]` section to profiles. #10462
- Feature: Add `meson_lib` and `meson_exe`, **`conan new`** templates. #10460 . Docs here
- Feature: Add *is_msvc_static_runtime* method to *conan.tools.microsoft.visual* to identify when using *msvc* with static runtime. #10437 . Docs here
- Feature: Improve support for Visual Studio in `AutotoolsToolchain`. #10429
- Feature: Make *pkg-config* tooling accessible under *conan.tools.gnu.PkgConfig* and *conan.tools.gnu.PkgConfigDeps*. #10415
- Feature: Use *.bazel* suffix for generated Bazel files. #10391 . Docs here
- Feature: New tools in *conan.tools.system* for invoking system package managers in recipes. #10380 . Docs here
- Feature: Testing the expected PC files created when the component name matches with the root package one using either *pkg_config* or *PkgConfigDeps* generators. #10344 . Docs here
- Feature: Better definition of clang compiler in Windows in *CMakeToolchain*. #10333
- Feature: Add VxWorks to OSs in default settings.yml. #10315 . Docs here
- Feature: Add *is_msvc* to validate if *settings.compiler* is *Visual Studio* and *msvc* compilers. #10310 . Docs here
- Feature: *os.sdk* field is mandatory for CMakeToolchain and OS in *('Macos', 'iOS', 'watchOS', 'tvOS')*. #10300
- Feature: Adding `--source-folder` and `--output-folder` to `conan editable` and **`conan install`** to work with `layout()`. #10274 . Docs here
- Feature: Adding clang 14 to `settings.yml`. Needed for emsdk package in Conan Center Index. #10269
- Feature: *PkgConfigDeps* shows *WARN* messages if there are duplicated *pkg_config_name* and/or *pkg_config_aliases*. #10263 . Docs here
- Feature: Improvements in `MesonToolchain`, including some cross-building functionality. #10174

- Feature: Update content created by the markdown generator. #9758 . Docs here
- Fix: Remove auto-detection of VS 2022 as `msvc` compiler, detect it as `Visual Studio` version `17`. #10457 . Docs here
- Fix: Do not report warning for duplicated component names in CMakeDeps. #10456
- Fix: Let legacy *Meson* build helper use other backends apart from *ninja*. #10447
- Fix: *msvc_runtime_flag* returns empty string instead of *None*. #10424 . Docs here
- Fix: Parsing a url with query args in `conan config install` results in a bad filename that could fail. #10423
- Fix: The argument *patch_file* from *tools.files.patch* is now relative to *conanfile.source_folder`by default, unless an absolute path to another location is provided, for example, to a path in the `conanfile.build_folder*. #10408 . Docs here
- Fix: Use install folder for Bazel dependency paths. #10391 . Docs here
- Fix: Enforce *CMP0091* policy to NEW in *CMakeToolchain*. #10390
- Fix: Add quotes around *conan_message* output variable so it is not modified. #10388
- Fix: Add `pathlib` as hidden-import to pyinstaller.py, so it is bundled with the installer. #10386
- Fix: Move imports of pre-defined layouts to their build-system domain. #10385 . Docs here
- Fix: Allow `cmake` generator checks for Visual Studio 2022. #10361
- Fix: Do not generate transitive *.props* for *MSbuildDeps* tool-requires. #10350
- Fix: Manage spaces in `[buildenv]` profile definition. #10343
- Fix: Add `-debug` to *LDFLAGS* in `AutotoolsToolchain` when necessary. #10339
- Fix: Fix extra *}* characters in cppstd info message. #10337
- Fix: Fix quotes in generated environment deactivation scripts. #10325
- Fix: Fix the `CMakeToolchain` generated code, so it doesnt fail for `-Werror --warn-unitilized`. #10292
- Fix: Fix spaces in settings.yml to prevent the YAML linter from complaining. #10230
- Fix: Convert *NewCppInfo* folders to absolute. #10207
- Fix: Improved CMakeToolchain robustness regarding `find_file`, `find_path` and `find_program` commands allowing better cross-build scenarios and better differentiation of the right context where to get, for example, executables (build vs host). #10186 . Docs here
- Bugfix: Fix `BazelDeps` using absolute `glob` paths instead of relative. #10478
- BugFix: Avoid BazelDeps exception when depending on a package without libs. Close https://github.com/conan-io/conan/issues/10471 #10472
- Bugfix: Fix `AttributeError: 'PackageEditableLayout' object has no attribute 'package_lock'` that happened when sing `package_revision_mode` with `editable` packages (and lockfiles). #10416
- Bugfix: Visual Studio 2022 auto-detected profile was incomplete. #10322
- Bugfix: Fix the caching of *ConanFile.dependencies* at *validate()* time. #10307
- Bugfix: Avoid `package_id` errors when using `compatible_packages` of repeated references (which can happen if using `private` dependencies). #10266

22.9 1.44.1 (13-Jan-2022)

- Bugfix: The *CMakeDeps* generator now uses the property *cmake_build_modules* declared in components of the required packages not only in the root cpp_info. #10326
- Bugfix: Adding missing hidden-imports to pyinstaller. Close https://github.com/conan-io/conan/issues/10318 #10320
- Bugfix: Make *pkg_config* generator listen to root *cpp_info* properties. #10312

22.10 1.44.0 (29-Dec-2021)

- Feature: Add *<PackageName>_LIBRARIES*, *<PackageName>_INCLUDE_DIRS*, *<PackageName>_INCLUDE_DIR*, *<PackageName>_DEFINITIONS* and *<PackageName>_VERSION_STRING* variables in CMakeDeps. #10227
- Feature: Adding a new Block to the CMakeToolchain now doesn't require inheriting *CMakeToolchainBlock*. #10213 . Docs here
- Feature: Add build_modules and build_modules_paths to JsonGenerator. #10203 . Docs here
- Feature: The *CMakeToolchain* is now prepared to apply several user toolchains. #10178 . Docs here
- Feature: In the conanfile.py of the test_package, the reference being tested is always available at *self.tested_reference_str*. #10171 . Docs here
- Feature: Introduced a new *test_type* value *explicit* so a user can declare explicitly the *requires* or *build_requires* manually (using *self.tested_reference_str*), it won't be automatically injected as a require. In Conan 2.0 the *test_type* attribute will be ignored, the behavior will be always explicit, so declaring *test_type="explicit"* will make the test recipe compatible with Conan 2.0. #10171 . Docs here
- Feature: Introduced *tool_requires* attribute to provide a compatible way to migrate to Conan 2.0, where the current concept of *build_requires* has been renamed to *tool_requires*. #10168 . Docs here
- Feature: Upgrade Conan python jinja requirement to v3.x. #10159
- Feature: Provided several *conan.tools.files* functions to manage symlinks: Transform absolute to relative symlinks, remove broken symlinks, remove external symlinks and get the symlinks in a folder. These tools will help migrate to Conan 2.0 where the package files won't be automatically cleaned from broken absolute symlinks or external symlinks. #10154 . Docs here
- Feature: Remove legacy folder setters in `conanfile.xxxx_folder = yyy` only used for testing. #10153
- Feature: Add `Git.version` property to check the current git version, aligned with `SVN.version`. #10114 . Docs here
- Feature: Add build_requires support in BazelDeps generators. #9876
- Fix: Fix variable names set by *CMakeDeps* modules. #10227
- Fix: Call to *find_dependency* in module mode to find transitive dependencies. #10227
- Fix: remove rpath from .pc files generated by *pkg_config* & *PkgConfigDeps* generators. #10192 . Docs here
- Fix: Deleted CMake warning for already existing targets. #10150
- Bugfix: Fix passing component's linkflags in CMakeDepes generator #10205
- Bugfix: *AutotoolsToolchain* was not passing the *compiler* to *get_gnu_triplet* function. #10141

22.11 1.43.4 (18-Feb-2022)

- Fix: Limit markupsafe python dependency to <2.1. #10616

22.12 1.43.3 (13-Jan-2022)

- Bugfix: The CMakeDeps generator now uses the property cmake_build_modules declared in components of the required packages not only in the root cpp_info. #10331
- Bugfix: Make pkg_config generator listen to root cpp_info properties. #10323

22.13 1.43.2 (21-Dec-2021)

- Fix: Remove `generator` argument from `cpp_info.set_property()` method. #10214 . Docs here
- Fix: Do not convert to `cmake_build_modules` property the legacy `cpp_info.build_modules`. #10208
- Bugfix: Compiler *msvc* was not working for CMake legacy generators. #10195

22.14 1.43.1 (17-Dec-2021)

- Bugfix: Making aggregate_components non-destructive, which was causing errors in generators with components. #10183
- Bugfix: Fix the definition of D_GLIBCXX_USE_CXX11_ABI in gcc-like compilers for `CMakeToolchain` and `AutotoolsToolchain`. Define it only to `D_GLIBCXX_USE_CXX11_ABI=0` for new compilers, assuming that the default is alread 1. #10165

22.15 1.43.0 (03-Dec-2021)

- Feature: Remove *cmake_target_namespace* and *cmake_module_target_namespace* properties. #10099 . Docs here
- Feature: Allow *CMakeDeps* to set *cmake_target_name* property as an absolute target. #10099 . Docs here
- Feature: Add warning in *CMakeDeps* generated CMake files when target names collide. #10099 . Docs here
- Feature: Legacy cmake generators (*cmake_find_package*, *cmake_find_package_multi*) don't listen to new *set_properties* model anymore. #10098 . Docs here
- Feature: *pkg_config_name* is used as the main name for a package/component and it will be used as the name for the **.pc* file. #10084 . Docs here
- Feature: Added new property`pkg_config_aliases` which admits a list of strings to define different aliases for any package/component. #10084 . Docs here
- Feature: Define `os=baremetal` in `settings.yml` to represent platforms without OS "bare metal". #10067 . Docs here
- Feature: Modern `tools.gnu.PkgConfig` to supersede legacy `tools.PkgConfig`. Includes management of *PKG_CONFIG_PATH* and mapping to a `cpp_info` structure #10065 . Docs here
- Feature: Add `test_requires()` for 2.0 migration of force_host_context. #10027 . Docs here

- Feature: Added C++23 support for Visual Studio and GCC 11.2 one. #10021
- Feature: Enable `from conan import ConanFile` to prepare for future namespace. #10016 . Docs here
- Feature: Add backend support to MesonToolchain. #9990 . Docs here
- Fix: Fix *<PackageName>_FIND_COMPONENTS* CMake generated variable to the correct value associated with the filename, not the package name. #10098 . Docs here
- Fix: Updated the ConanException in installer.py to improve the error message handling. #10089 . Docs here
- Fix: Simplify `parallel` definition in new `conf`, leaving only `tools.build:jobs`. Use max number of CPUs by default to build in parallel. #10068 . Docs here
- Fix: Fix the `msvc` version model, which is comparison broken with the "main" 19.3 version < 19.22. #10057 . Docs here
- Fix: Fix the *EnvVar.save_ps1()* method. #10049
- Fix: CMakeToolchain will not crash if build_type not defined. #9984
- Fix: Avoid raising an exception for `conan info --paths` when there are editables in the graph. #9944
- Fix: Fix wrong parameter name in CMakeDeps find_library function. #9932
- Fix: Improve error message when unzipping Conan *.tgz* artifacts. #9925
- Fix: Respect error code 6 in some situations. #9905
- Bugfix: Fix parallel package downloading. While downloading conan locks incorrect package ref. #10038
- Bugfix: Add import for `CMakeToolchainBlock` custom Blocks in `CMakeToolchain`. #10026 . Docs here
- Bugfix: Option *–require-override* is not working for *conanfile.txt*. #10013
- Bugfix: Fix unescaped double-quotes for defines in `Premake` generator. #10008
- Bugfix: Use new *tools.cross_building* in MesonToolchain . #9992
- Bugfix: CMakeDeps generated **-data.cmake* was not including properly the set of link flags. #9980
- Bugfix: CMakeDeps was not populating *INTERFACE_LINK_OPTIONS* to each target. #9980
- Bugfix: *PkgConfigDeps* was not adding correctly the *Requires* for all the package dependencies. #9945
- Bugfix: `user_toolchain` properly included and path quoted in `CMakeToolchain`. #9916
- Bugfix: Solved an issue with the *conan config install* whereby a cryptic error was raised when a user tried to install a directory that previously was a file and vice-versa. #9908
- Bugfix: Missing framework for Xcode generator with no compiler setting. #9896

22.16 1.42.2 (22-Nov-2021)

- Bugfix: Legacy *cmake_multi* generator is not affected by *set_property*. #10062

22.17 1.42.1 (08-Nov-2021)

- Fix: Fix XcodeDeps architecture name translation from Conan to Apple identifiers. #9955
- Bugfix: Fix XcodeDeps bad xcconfig generation when using dash-case-named packages #9955
- Bugfix: Avoid exception if `msvc` compiler not defined in settings.yml file. #9954
- Bugfix: legacy *cmake* generator is not affected by *set_property*. #9952

22.18 1.42.0 (29-Oct-2021)

- Feature: Remove *sdk* condition in _.xcconfig_ files generated by _XcodeDeps_. #9887
- Feature: Add new Macos version 12.0 (Monterey). #9886
- Feature: Add *CMAKE_POSITION_INDEPENDENT_CODE* in *CMakeToolchain* if it's set in the conanfile independently from the value of the shared option. #9868
- Feature: Generate aggregated deactivation file for aggregated environments. #9862 . Docs here
- Feature: adding preprocessor definitions to `MSBuildTolchain` ResourceCompile. #9843
- Feature: Support *cmake_module_target_name* and *cmake_module_file_name* properties in _cmake_find_package_ generator. #9832 . Docs here
- Feature: Define new `conf["tools.microsoft.msbuild:install_path"]` for the MSBuild toolchain and remove generation of intel_vars files by `VCVars`. #9827 . Docs here
- Feature: New Conan _XcodeDeps_ multi-config generator for _Xcode_. #9807 . Docs here
- Feature: Refactor *conan.tools.gnu.pkgconfigdeps* module. #9806
- Feature: Decoupled `Environment` as an abstract environment representation (do not depend on conanfile), and `EnvVars`, as the specialization for a given conanfile, settings/settings_build, and `win_bash`. #9755 . Docs here
- Feature: Enabled *patch_string* when using *apply_conandata_patches*. #9740 . Docs here
- Feature: Add property "cmake_target_aliases" to create some CMake alias targets using CMakeDeps. #8533
- Fix: Do not fall back for the property "filename" to target properties, only to legacy "names" definition. #9894
- Fix: Fix build settings conditional logic to work correctly with Xcode IDE. #9892
- Fix: Show a more specific error message on 'conan search' when a package version is not valid #9891
- Fix: Removed the new layout *folders.package* property as it was not clear the value/usage. #9884 . Docs here
- Fix: The "conan package" method now raises an exception when declaring the layout() method. This is part of the migration to Conan 2.0 where the local method "conan package" is removed. #9884 . Docs here
- Fix: Moved experimental LayoutPackager to a *conan.tools.files.AutoPackager* with a new interface. Removed also the *conanfile.patterns* attribute that now is configured directly in the AutoPackager. #9882 . Docs here
- Fix: When the *layout()* method is declared inside the conanfile.py of a *test-package*, Conan won't generate temporary folders at build/_HASH_ anymore. The build folder will follow the structure declared at the layout() method. #9882 . Docs here
- Fix: Fix frameworks aggregation in XcodeDeps. #9881
- Fix: Remove python `requests` version from `User-Agent` header, it can be problematic sometimes and adds no value. #9861

- Fix: Change `MesonToolchain` to define `preprocessor_definitions` as `[constant]`, because Meson 0.60 now raises errors on it defined in `[built-in options]`. #9858
- Fix: Rename `ConanFileInterface.new_cpp_info` to `ConanFileInterface.cpp_info` (being it a `NewCppInfo` object). The *ConanFileInterface* object is only used when accessing *self.dependencies* in a conanfile, so we can make sure that the new develop generators use the final *cpp_info* name. #9800 . Docs here
- Fix: Renamed `group` argument in `Environment` classes to `scope`, that can get "build" and "run" values. #9755 . Docs here
- Fix: New `win_bash` behavior failed for dual profile, the new computation of subsystem paths completely depend now on defined settings, but not on `platform.system()` or auto-detection. #9755 . Docs here
- Bugfix: Avoid lockfile raising because of incorrect options when using `compatible_packages`. Close https://github.com/conan-io/conan/issues/9591 #9890
- Bugfix: Avoid VCVars defining `-vcvars` argument for VS<=2015. Close https://github.com/conan-io/conan/issues/9888 #9889
- Bugfix: Respect the previous value of `sys.dont_write_bytecode`. #9865
- Bugfix: **`conan export-pkg`** now raises an error (ConanInvalidConfiguration exception) if the `validate()` method raise that error and results in an invalid *package_id*. #9818
- Bugfix: Respect the `build_policy="never"` even for `--build=missing` cli argument. #9817
- Bugfix: Avoid crash in `CMakeToolchain` for custom generator when compiler is not defined. #9801
- Bugfix: Do not force new authentication when a token is expired, it might not be needed. #9781
- Bugfix: The CMakeToolchain generator always sets fPIC enabled due to a typo where the actual parameter is not templated but instead hard-coded as *ON*. #9752
- Bugfix: The path to a patch file in a *conandata.yml* is also relative to the base source folder when the patches are not associated with a version (list of patches). #9740 . Docs here

22.19 1.41.0 (06-Oct-2021)

- Feature: Added *IntelCC* as public generator. #9747 . Docs here
- feature: Prepare `conan.tools.files download, get, ftp_download` for adoption. #9715 . Docs here
- Feature: Support multiple toolchains in one recipe. #9688 . Docs here
- Feature: `MSBuildDeps` generator learned how to handle `build_requires` to use executables from them. #9686 . Docs here
- Feature: `PkgConfig` helper now honors *PKG_CONFIG* environment variable. #9627
- Feature: Make new environment generators multi-config (Release/Debug and arch). #9543 . Docs here
- Feature: Environment `activate` scripts will generate `deactivate` scripts by default. #9539
- Feature: Support remote archives other than zip in `conan config install`. #9530
- Feature: New "compiler" *intel-cc* with different modes (icx, dpcpp, classic) for Intel oneAPI. #9522 . Docs here
- Feature: Add Intel oneAPI support for *CMakeToolChain*, *MSBuildToolchain* and *VCVars*. #9522 . Docs here
- Feature: Add objects attribute to the cpp_info so that we can add object files to the linker without having to add those mixed with linker flags. #9520 . Docs here
- Feature: New setttings: _xtensalx6_ and _xtensalx106_ for ESP32/ESP8266 platforms. #7977

- Fix: New environment deactivate scripts fail in Windows because env-var different casing. #9741
- Fix: Avoid "overcrowding" Windows environment variables with *LD_LIBRARY_PATH* and *DYLD_LIBRARY_PATH*, not used in Windows. #9678
- Fix: Command **`conan package`** now works for new `layout()` with generators folder. #9674
- Fix: Don't just print "ERROR: True" on unresolvable conflict. #9624
- Fix: Fix migrations settings comparison. #9615
- Fix: Removed unused `future` dependency from Python `requirements.txt`. #9563
- Fix: Make automatic call of `VirtualBuildEnv` and `VirtualRunEnv` independent. #9543 . Docs here
- Bugfix: `generator_folder` was using the `cwd` instead of a relative folder to the conanfile, producing layout errors. #9668
- Bugfix: Support components with symbolic imports. #9667
- Bugfix: Fix `conan.tools.files.patch` bug with `strip` argument. #9653
- Bugfix: Add support for Xcode 13/Apple clang 13.0 #9642
- Bugfix: Fixed bad *Requires* declaration in **.pc* files. #9635
- Bugfix: Fixed bug whereby when a conflict of requirements happens, in some special situations, just a message *ERROR: True* was printed in the terminal, making it hard to guess what was going on. #9633
- BugFix: Fixed a bug where using the new *layout()* the *exports* went to the *conanfile.source_folder* instead of the base source folder in the cache and only the *exports_sources* should go there. #9536

22.20 1.40.4 (05-Oct-2021)

- Fix: Check current _cacert.pem_ file when updating Conan and only migrate if the user has not modified the file. If the local file is modified then create a new cacert file but don't overwrite current. #9734
- Fix: Update Conan debian package to fix ssl certificates problem. #9723

22.21 1.40.3 (30-Sept-2021)

- Bugfix: Added root certificate for Let's encrypt. #9697

22.22 1.40.2 (21-Sept-2021)

- Bugfix: Add support for Xcode 13/Apple clang 13.0 #9643

22.23 1.40.1 (14-Sept-2021)

- Feature: Change default *cmake_layout()* source folder from 'src' to '.' #9596 . Docs here
- Feature: Recovered *base_path* argument for *conan.tools.files.patch* and *conan.tools.files.apply_conandata_patches* to be able to specify a relative folder from the *conanfile.source_folder* directory (that follows the layout() method). #9593 . Docs here
- Fix: Allow user definition of *CMAKE_XXX_INIT* variables in user toolchains when using *CMakeToolchain*. #9576
- Fix: Upgrade minimum `requests>=2.25` in requirements.txt to make it compatible with latest upgrade of `urllib3` to 1.26.6 #9562
- Bugfix: Aggregate *[conf]* from *build_requires* earlier so it is available for generators declared as *generators*. attribute. Close https://github.com/conan-io/conan/issues/9571 #9573
- Bugfix: The qmake generator now assigns *QMAKE_LFLAGS_SHLIB* and *QMAKE_LFLAGS_APP* variables instead of the incorrect *QMAKE_LFLAGS* following the official docs. #9568 . Docs here

22.24 1.40.0 (06-Sept-2021)

- Feature: Update **`conan new`** modern templates `--template=cmake_lib` and `--template=cmake_exe`. #9516 . Docs here
- Feature: Introduced a new cpp_info property *cmake_target_namespace* to declare the target namespace for the *CMakeDeps* generator. This feature allows declaring a global target with a different namespace like *Foo::Bar*. #9513 . Docs here
- Feature: Detect Visual Studio 2022 as *msvc*. #9504
- Feature: Add Clang 13 support. #9502 . Docs here
- Feature: Testing support for Windows CMake + Clang (independent LLVM, not VS) + Ninja/MinGW builds, and CMake + Clang (Visual Studio 16 internal LLVM 11 via ClangCL toolset). #9477
- Feature: Provide new `[conf] core:default_build_profile` to enable the usage of the build profile as default, and to allow definition of the host profile default in new `[conf] core:default_profile`. #9468 . Docs here
- Feature: **CMakeToolchain** new member *find_builddirs* defaulted to *True* to add the *cpp_info.builddirs* from the requirements to the *CMAKE_PREFIX_PATH/CMAKE_MODULE_PATH*. That would allow finding the config files packaged and to be able to *include()* them from the consumer *CMakeLists.txt*. #9455 . Docs here
- Feature: **CMakeDeps**. Added a new property *cmake_find_mode* with possible values to *config`(default), `module*, *both* or *none* to control the files to be generated from a package itself. The *none* replaces the current *skip_deps_file* property. #9455 . Docs here
- Feature: **CMakeDeps**: Added two new properties *cmake_module_file_name* and *cmake_module_target_name*, analog to *cmake_file_name* and *cmake_target_name*, but to configure the name of *FindXXX.cmake* file and the target declared inside. #9455 . Docs here
- Feature: Remove *conan-center* (https://conan.bintray.com) default remote. #9401 . Docs here
- Feature: Implement round trip new profile `[buildenv]` section, necessary for lockfiles, and specially stdout printing. #9320
- Feature: Allow `-o &:option=value` wildcard for consumer, too, same it was done for settings in 1.39 #9316 . Docs here

- Fix: Adding management of private dependencies, via new `visible` trait compatible with 2.0 for new `CMakeDeps` and `MSBuildDeps`. #9517
- Fix: Remove unused `deprecation` pip dependency #9478
- Fix: Upgrade `distro` dependency to allow 1.6.0 #9462
- Fix: Make **`conan remove`** accept package reference syntax. #9459 . Docs here
- Fix: Fixed old CMake build helper to cross-build to iOS when two profiles are specified. #9437
- Fix: Fix **`conan export`** typo in help message. #9408 . Docs here
- Fix: Relax python six dependency to allow 1.16 #9407
- Fix: Bump urllib3 version to 1.26.6 #9405
- Fix: The new *Autotools* build helper accepts a *build_script_folder ` argument in the `configure()* method to specify are subfolder where the configure script is. #9393 . Docs here
- Fix: Use *frameworks* in Premake generator. #9371 . Docs here
- Fix: The tool *conan.tools.files.apply_conandata_patches* will use the root source folder to find the patch file and the tool *conan.tools.files.patch* will take the current source folder declared in the *layout()* method to know where is the source to apply the patches. #9361 . Docs here
- Fix: Avoid checking other remotes when `-r=remote` is defined and revisions are activated and binary is not found in the defined remote. #9355
- Bugfix: Setting the *CMAKE_OSX_DEPLOYMENT_TARGET* variable as a cache entry. #9498
- BugFix: Use topological ordering to define *VirtualBuildEnv* composition and precedence of appending variables. #9491
- Bugfix: Bazel build files have an extra *]* if there are no dependencies. #9480
- Bugfix: Add AlmaLinux to *with_yum*. #9463
- Bugfix: **CMakeToolchain**. Fixed a bugfix whereby a variable declared at the *.variables* containing a boolean ended at CMake with a quoted *"True"* or *"False"* values, instead of *ON / OFF* #9455 . Docs here
- Bugfix: Fixed bug whereby Conan failed when using *compiler=gcc* with *compiler.version=5* (without specifying a minor version) and *compiler.cppstd=17*. #9431
- Bugfix: No verbose traceback was been printed for *conanfile.layout()* method. #9384
- Bugfix: Fix Bazel *cc_library*: *deps* and *linkopts*. #9381
- Bugfix: Fixed bug whereby using new *layout()* method together with *cppinfo.components* in the *package_info* method caused an exception. #9360
- Bugfix: Fix *PkgConfigDeps* that was failing in the case of components with requirements. #9341

22.25 1.39.0 (27-Jul-2021)

- Feature: Use *CMAKE_OSX_DEPLOYMENT_TARGET* to get *-version-min* set in _CMakeToolchain_. #9301
- Feature: Display `python_requires` information in the **`conan info`** output. #9290
- Feature: Now it is possible to define settings for a downstream consumer using *-s &:setting=value* or the same syntax in the profile even if the consumer is a *conanfile.txt* or a *conanfile.py* not declaring a name. e.g: Building a Debug application with Release dependencies: *-s build_type=Release -s &:build_type=Debug* #9267 . Docs here

- Feature: The *AutotoolsDeps* allows to alter the generated environment corresponding to the information read from the dependencies before calling the *generate()* method. #9256 . Docs here
- Feature: new *remove* and *items* methods for the *Environment* objects. #9256 . Docs here
- Feature: New *VCVars* generator that generates a *conanvcvars.bat* that activates the Visual Studio Developer Command Prompt. #9230 . Docs here
- Feature: Skip build helper test and using [conf]. #9218 . Docs here
- Feature: Implement a new `requires = "pkg/(alias)"` syntax to be able to dissambiguate alias requirements and resolve them earlier in the flow, solving some limitations of the previous alias definition. This approach is intended to be the one in Conan 2.0 (issue backported from https://github.com/conan-io/tribe/pull/25). #9217 . Docs here
- Feature: Introduce the `-require-override` argument to define dependency overrides directly on command line. #9195 . Docs here
- Feature: New *self.win_bash* mechanism to enable running commands in a bash shell in Windows. It works only with the new environment definition from the dependencies (*env_buildinfo* and *run_buildinfo*) as long as the new *AutotoolsToolchain*, *AutotoolsDeps* and *Autotools* build helper. It supports automatic conversion of the environment variables values declared as "path" according to the declared subsystem in the conf *tools.win.bash:subsystem*, that is not being auto-detected anymore. #9194 . Docs here
- Feature: A unique environment launcher (*conanenv.bat/sh*) is generated to aggregate all the environment generators (*VirtualRunEnv*, *VirtualBuildEnv*, *AutotoolsToolchain* and *AutotoolsDeps*) that had been generated so the user can easily activate all of them with one command. #9161 . Docs here
- Feature: Use _CMake_ File API. #9005
- Fix: Add `bindirs` definition to `cmake_layout()`. #9276
- Fix: Improve error message when `conan search <ref>` a package in editable mode. #9262
- Fix: Add `options` to `conanfile.dependencies` model. #9258
- Fix: Fix _CMake_ rejecting library name with special characters. #9245
- Fix: Use filename *[PKG-NAME]-[COMP-NAME]* for *PkgConfigDeps*. #9228 . Docs here
- Fix: Saving all the toolchain args information into *conanbuild.conf* instead of json file. #9225 . Docs here
- Fix: Added warning in the new toolchains (the used in the generate() method) if no build profile is being used. #9206 . Docs here
- Fix: Implemented check that will raise an error in the *CMakeDeps* generator when using the *build_context_activated*, *build_context_suffix* or *build_context_build_modules* attributes if no build profile is being used. #9206 . Docs here
- Fix: The`CMakeDeps` generator will check if the targets specified in the *find_package(foo components x y z)* exist instead of checking against an internal variable. Also, this check will be done at the end of the *xxx-config.cmake* so any included *build_module* can declare the needed targets. #9206 . Docs here
- Fix: Consistent help message for conan profile (sub-command part). #9204 . Docs here
- Fix: Consistently put short arguments (-a) before long ones (–args). #9199
- Fix: *CC=clang –gcc-toolchain* is now identified as _clang_. #9198
- Fix: The new *VirtualEnv* generator has been split into *VirtualRunEnv* and *VirtualBuildEnv*. Both are automatically generated as before but only *VirtualBuildEnv* will be activated by default. #9161 . Docs here
- Bugfix: Fixing `workspace install` when conanfile has `imports()`. #9281
- Bugfix: Fix QbsProfile toolchain `qbs.architecture` KeyError. #9192

- Bugfix: Do not define CMAKE_GENERATOR_TOOLSET in CMakeToolchain for Ninja generator, and define it in `vcvars_ver` instead. #9187
- BugFix: `build_requires` in host context, like gtest, are being propagated downstream by generators in the `dependencies` model. #9171 . Docs here
- Bugfix: Fix that overridden requirements _"cannot be found in lockfile"_. #8907

22.26 1.38.0 (30-Jun-2021)

- Feature: New `PkgConfigDeps` generator. #9152 . Docs here
- Feature: Proposal of jinja2 templates for profiles. #9147 . Docs here
- Feature: Add support for *CMAKE_CXX_STANDARD_REQUIRED* in _CMakeToolchain_. #9144
- Feature: Add `context` information to **`conan info`** output both to stdout and json outputs. #9137 . Docs here
- Feature: Improved the new *AutotoolsToolchain*, *AutotoolsDeps* and *Autotools* build helper. #9131 . Docs here
- Feature: Initial cross-build support in `CMakeToolchain` with definition of `CMAKE_SYSTEM_NAME`, `CMAKE_SYSTEM_PROCESSOR` and `CMAKE_SYSTEM_VERSION`, deduced from `self.settings_build` (only using the 2 profiles) and from new `[conf]` items. #9115 . Docs here
- Feature: Easier access to modify or update context values in `CMakeToolchain` blocks. #9109 . Docs here
- Feature: Provide *[conf]* command line support. #9103 . Docs here
- Feature: Added support for using server config from a custom location, setting *CONAN_SERVER_HOME* env variable or using *-d* or *–server_dir* flag when launching the server with *conan_server* command. #9099 . Docs here
- Feature: Support for *CMakeDeps* generator of a new property *"skip_deps_file"* to be declared in the *cpp_info* of a package to skip creating *xxx-config.cmake* files for it, allowing to create "system wrapper" recipes easily. #9087 . Docs here
- Feature: New `conanfile.dependencies` model, using a dict {requirement: ConanFileInterface} to prepare for Conan 2.0. #9062 . Docs here
- Feature: Allow a explicit `requires = "pkg..#recipe_revision"` to update cache revision without `--update`. #9058 . Docs here
- Feature: New `cmake_layout()` layout helper to define a multi-platform CMake layout that will work for different generators (ninja, xcode, visual, unix), and is multi-config. #9057 . Docs here
- Feature: The *conan_toolchain.cmake* now includes *xxx_DIR* variables for the dependencies to ease the *find_package* mechanism to locate them. The declaration of these directories is a must when cross-building in OSX where CMake ignores *CMAKE_PREFIX_PATH* and *CMAKE_MODULE_PATH* to look only at the system framework directories. #9032 . Docs here
- Feature: Provide access in the recipes to the environment declared with the [new environment system](https://docs.conan.io/en/latest/reference/conanfile/tools/env.html). #9030 . Docs here
- Fix: Fix Bazel build string defines. #9139
- Fix: Fixed behavior in the *self.folders* feature whereby the sources saved or downloaded inside the *source(self)* were saved at the *self.folders.source* folder. But *self.folders.source* is intended to describe where the sources are instead of forcing where the sources are saved. #9124 . Docs here
- Fix: Properly generate qbs profile for msvc. #9122
- Fix: Configuration general.user_home_short works with "None" value. #9118

- Fix: Avoid `CMakeToolchain` to generate OSX and Apple config for non Apple builds. #9107
- Fix: The new *MesonToolchain* now takes the declared environment variables (*CC*, *CXX*...) from build-requires and profiles to set the variables *c*, *cpp*, *c_ld*, *cpp_ld* etc, into the *conan_meson_native.ini* #8353 . Docs here
- Fix: Added new *preprocessor_definitions* to new Meson build helper. #8353 . Docs here
- Fix: The new *MesonToolchain* now allows adjusting any variable before generating the *conan_meson_native.ini* file. #8353 . Docs here
- Bugfix: Disabled remotes shouldn't fail if not used at all #9184
- BugFix: `ConanFileDependencies.build["dep"]` was retrieving the `host` dependency if existing, or failing otherwise, because the default requires was hardcoded to fetch the host (build=False) dependency. #9148 . Docs here
- Bugfix: Now, *conan profile {show, update, get, remove}* is working fine with new experimental *[conf]* section. #9114

22.27 1.37.2 (14-Jun-2021)

- Bugfix: Avoid crash when using `--lockfile` with the **`conan test`** command. #9089
- Bugfix: The *CMakeDeps* generator variables were named wrongly when a component had the same name as the package. #9073

22.28 1.37.1 (08-Jun-2021)

- Fix: Update the experimental `conan new ... -m=v2_cmake` template to start using the new `layout()` basic info. #9053
- Fix: Do not fail in `CMakeDeps` when there are `build_requires` with same name as host requires, unless `build_context_activated` is enabled for those and a different suffix has not been defined. #9046
- Fix: When using the new *self.folders.source* (at *layout(self)* method) the sources (from *export*, *export_sources* and *scm*) are copied to the base source folder and not to the *self.folders.source* that is intended to describe where the sources are after fetching them. #9043 . Docs here
- BugFix: Do not quote all values and allow integer and macro referencing in `MSBuildToolchain.preprocessor_definitions` #9056
- Bugfix: The new generators like *CMakeDeps* and *CMakeToolchain`write the generated files defaulting to the `install folder* if no *self.folders.generators* is specified in the *layout()* method. #9050
- Bugfix: The *CMakeToolchain* generator now manages correctly a recipe without *arch* declared in an Apple system. #9045

22.29 1.37.0 (31-May-2021)

- Feature: Remove `CMAKE_SKIP_RPATHS` by default to True in `CMakeToolchain`, it is not necessary by default, users can opt-in, and new test validates shared libs will work with `VirtualEnv` generator `conanrunenv`. #9024 . Docs here
- Feature: simplified `CMakeToolchain` with only 1 category of blocks, made `try-compile` template code as another block, and reordered blocks so relevant flags for try-compile are taken into account. #9009 . Docs here
- Feature: Add new default *conancenter* remote for *https://center.conan.io* as first in the list. #8999 . Docs here
- Feature: Implements a new experimental `conan.tools.google` Bazel integration with `BazelDeps`, `BazelToolchain` and `Bazel`. #8991 . Docs here
- Feature: Introduced new options for the *CMakeDeps* generator allowing to manage *build_requires* even declaring the same package as a *require* and *build_require* avoiding the collision of the *config* cmake files and enabling to specify which *build_modules* should be included (e.g protobuf issue) #8985 . Docs here
- Feature: Expand user-agent string to include OS info. #8947
- Feature: Implement `build_policy=never` for **`conan export-pkg`** packages that cannot be rebuilt with `--build=xxx`. #8946 . Docs here
- Feature: Define `[conf]` for defining the user toolchain for `CMakeToolchain`, both for injecting a user toolchain in the `CMakeToolchain` generated `conan_toolchain.cmake` and for completely replacing `conan_toolchain.cmake`. #8945 . Docs here
- Feature: add GCC 11 to _settings.yml_. #8924
- Feature: Add new *tools.rename()* interface. #8915 . Docs here
- Feature: Update urlib3 Conan dependency setting version *>=1.25.8* to avoid CVE-2020-7212. #8914
- Feature: Build-requires can define [conf] for its consumers. #8895 . Docs here
- Feature: support M1 Catalyst. #8818
- Feature: New `conan install <ref> --build-require` and `conan create <path> --build-require` (when not using `test_package`) arguments to explicitly define that the installed or created package has to be a `build-require`, receiving the build profile instead of the host one. #8627 . Docs here
- Feature: Introduced the *layout()* method to the recipe to be able to declare the folder structure both for the local development methods (conan source, conan build...) and in the cache. Also, associated to the folders, cppinfo objects to be used in editable packages and file pattern descriptions to enable "auto packaging". #8554 . Docs here
- Fix: **CMakeDeps** generator: The transitive requirements for a build_require are not included in the *xxx-config.cmake* files generated. #9015
- Fix: The *CMakeToolchain* now supports Apple M1 cross-building with a profile without environment declared pointing to the system toolchain. #9011
- Fix: Set *env_info.DYLD_FRAMEWORK_PATH* correctly. #8984
- Fix: Fix some typos in the code. #8977
- Fix: Improve error message when a directory doesn't contain a valid repository. #8956
- Fix: The *build_modules* defined per generator in *cpp_info* now are rendered properly using the *markdown* generator. #8942
- Fix: Simplify code access to [conf] variables removing attribute based access. #8901

- Bugfix: Prevent unintended evil insertions into metadata.json resulted in corrupted package and inability to install. #9022
- Bugfix: Allow `MSBuildDeps` to correctly process packages with dots in the package name. #9012
- Bugfix: Avoid errors because of `package_id` mismatch in lockfiles when using `compatible_packages` feature. #9008
- BugFix: Respect order of declared directories when using components. #8927
- Bugfix: Raise an exception when response header *Content-type* is different than *application/json* or *application/json; charset=utf-8*. #8912
- Bugfix: Fix exception in CMakeToolchain when settings remove known compilers. #8900
- Bugfix: Fix current directory definition in `vcvars` commands in new toolchains. #8899
- Bugfix: AptTool: add repo key before running apt-add-repository. #8861
- BugFix: Prevent evil insertions into metadata.json resulted in corrupted package and inability to install. #8532

22.30 1.36.0 (28-Apr-2021)

- Feature: Add support to `tools.cmake.CMake` for Ninja toolchain defined with `CMakeToolchain`. #8887
- Feature: The *CMakeDeps* generator will print CMake traces with the declared targets. e.g: *Target declared: 'OpenSSL::Crypto'*. #8843
- Feature: Add clang 12 support. #8828
- Feature: List tools and core from profile and _global.conf_. #8821 . Docs here
- Feature: Add cross-building tests for new AutoTools build helper. #8819
- Feature: `CMakeToolchain` generates a `conanbuild.json` file with the generator to be used in the `CMake` command line later, so it is not necessary to duplicate logic, and is explicit what generator should be used. #8815 . Docs here
- Feature: `CMakeToolchain` learned to build with different toolsets, down to the minor compiler version, for the `msvc` compiler. #8815 . Docs here
- Feature: Validate checksum and retry download for corrupted downloaded cache files. #8806
- Feature: `CMakeToolchain` defining *CMAKE_GENERATOR_TOOLSET* for msvc version different than the default. #8800
- Feature: Implement `test_build_require` in `test_package/conanfile.py` recipes, so build_requires can be tested as such. #8787 . Docs here
- Feature: Make available the full recipe and package reference to consumers via `.dependencies`. #8765
- Feature: New `CMakeToolchain` customization and extensibility mechanism with blocks of components instead of inheritance. #8749 . Docs here
- Feature: Add *set_property* and *get_property* to set properties and access them in generators. Can be set only for a specific generator or as a default value for all of them. #8727 . Docs here
- Feature: Use *set_property* and *get_property* to support custom defined content in *pkg_config* generator. #8727 . Docs here
- Feature: Add new property names: *cmake_target_name*, *cmake_file_name*, *pkg_config_name* and *cmake_build_modules* that can be used for multiple generators of the same type allowing also an easier migration of *names*, *filenames* and *build_modules* properties to this model. #8727 . Docs here

- Feature: Skip package when building all package from sources at once using *–build=!<package>* syntax. #8483 . Docs here
- Feature: `CMakeToolchain` will generate `conanvcvars.bat` for Ninja builds for `msvc`. #8005
- Fix: Remove `tools.gnu.MakeToolchain`, superseded by `tools.gnu.AutotoolsToolchain`. #8880 . Docs here
- Fix: Allow spaces in the path for new environment files and `conancvvars.bat` Visual toolchain file. #8847
- Fix: Return *deprecated* attribute in **`conan inspect`** command. #8832
- Fix: Check if Artifactory url for publishing the build_info has *artifactory* string as the service context and remove from the API url if it doesn't. #8826
- Fix: Recognize `Ninja Multi-Config` as a CMake multi-configuration generator. #8814
- Fix: using *CMAKE_CURRENT_LIST_DIR* in *CMakeToolchain* to locate *CMakeDeps* config files. #8810
- Fix: *config_install_interval* no longer enter in loop when invalid. #8769 . Docs here
- Fix: Remove multi-config support for `CMakeDeps` generator. #8767 . Docs here
- Fix: Accept relative profile path when folder is on same tree level. #8685 . Docs here
- Bugfix: Fixed test_package/conanfile.py using `build_requires` for a package belonging to a lockfile. #8793

22.31 1.35.2 (19-Apr-2021)

- Bugfix: Revert regression that replaces first `/` by `-` in `cpp_info.xxxxlinkflags` in _CMake_ generators because it can break passing objects and other paths that start with `/`. #8812

22.32 1.35.1 (13-Apr-2021)

- Fix: Avoid breaking users calling forbidden private api `conanfile.__init__`. #8746
- Bugfix: Fix opensuse SystemPackageTools incorrectly using apt-get when zypper-aptitude. #8747
- Bugfix: Fix linker flags in cmake (find_package based) generators. #8740
- Bugfix: Fixed bug in transitive build_requires of MSBuildDeps. #8734

22.33 1.35.0 (30-Mar-2021)

- Feature: `MSBuildDeps` generator uses new visitor model and handles conditional requirements correctly. #8733 . Docs here
- Feature: CMake toolchain supports include_guard() feature #8728
- Feature: New `conan lock bundle clean-modified` command. #8726 . Docs here
- Feature: Use `conancvvars.bat` file for Meson toolchain #8719
- Feature: Allow arbitrary defines in **`conan new`** templates. #8718 . Docs here
- Feature: Automatically handle *CONAN_RUN_TESTS* to avoid extra boilerplate. #8687 . Docs here
- Feature: More fine-grained control (using [conf]) for build parallelization. #8665 . Docs here

- Feature: Add support for testing with different tools versions. #8656
- Feature: Add different CMake versions for testing. #8656
- Feature: Move the definition of CMakeDeps variables to its own file #8655 . Docs here
- Feature: Added *conan.tools.files.patch* to apply a single patch (new interface for legacy *conans.tools.patch* function. #8650 . Docs here
- Feature: Added *conan.tools.files.apply_conandata_patches* to apply patches defined in *conandata.yml*. #8650 . Docs here
- Feature: Allow integers as `preprocessor_definitions` in `CMakeToolchain`. #8645
- Feature: New `Environment` model for recipes and profiles #8630 . Docs here
- Feature: Do not remove sh from the path in the new CMake helper. #8625 . Docs here
- Feature: Allow definition of custom Visual Studio version for msvc compiler in MSBuild helpers. #8603 . Docs here
- Feature: MSBuildToolchain creates conanvcvars.bat containing vcvars command for command line building. #8603 . Docs here
- Feature: Set *CMAKE_FIND_PACKAGE_PREFER_CONFIG=ON*. #8599
- Feature: Include the recipe name when constrained settings prevent install. #8559 . Docs here
- Feature: Create new conan.tools.files for 2.0. #8550
- Feature: New AutotoolsDeps, AutotoolsToolchain helpers in conan.tools.gnu #8457 . Docs here
- Feature: Experimental `conan lock install` that can install a lockfile in the cache, all the binaries or only the recipes with `--recipes`, intended for CI flows. #8021 . Docs here
- Fix: Fix incorrect output of `default_user` and `default_channel` in `export`. #8732
- Fix: remotes not being loaded for the **`conan alias`** command, which was preventing **`conan alias`** from working if python_requires is used. #8704
- Fix: Improve error message for `lock create` providing a path instead of full path with filename. #8695
- Fix: Rename *tools.microsoft:msbuild_verbosity* to *tools.microsoft.msbuild:verbosity* #8692 . Docs here
- Fix: Simplifications to `CMakeDeps` generator to remove legacy code. #8666
- Fix: Add dirty management in download cache, so interrupted downloads doesn't need a manual cleaning of such download cache. #8664
- Fix: Build helper qbs install now installs directly into package_folder. #8660
- Fix: Allow arbitrary template structure. #8641
- Fix: Restoring the behavior that *exports* and *exports_sources* were case sensitive by default. #8585
- Fix: Remove default dummy value for iOS XCode signature. #8576
- Fix: Do not order Settings lists, so error messages are in declared order. #8573
- BugFix: Command **`conan new`** accepts short reference with address sign. #8721
- Bugfix: Fix profile definitions of env-vars per-package using patterns, not only the package name. #8688
- Bugfix: Preserve the explicit value *None* for SCM attributes if the default is a different value. #8622
- Bugfix: Properly detect Amazon Linux 2 distro. #8612
- Bugfix: Fix config install not working when .git* folder is in the path. #8605

- Bugfix: Fix: Transitive python requires not working with the new syntax. #8604

22.34 1.34.1 (10-Mar-2021)

- Fix: Allow `cmake_find_package_multi` and `CMakeDeps` to be aliases for `cpp_info.names` and `cpp_info.filenames` to allow easy migration. #8568
- Bugfix: Restoring the behavior that *exports* and *exports_sources* were case sensitive by default. #8621
- BugFix: Solved issues with already existing packages appearing in `conan lock bundle build-order`. #8579

22.35 1.34.0 (26-Feb-2021)

- Feature: Add *path* and *repository* properties to conan_build_info v2. #8436
- Feature: Setting _conan_ as name for *buildAgent* in *conan_build_info*. #8433
- Feature: Using actual conan version in version for *buildAgent* in *conan_build_info* instead of 1.X. #8433
- Feature: Add *type* _conan_ to Conan build info modules. #8433
- Feature: Add `scm` output in **`conan info`** command. #8380
- Feature: Forked `cmake_find_package_multi` into `CMakeDeps`, to allow evolution without breaking. #8371
- Feature: Use built-in retries in requests lib to retry http requests with _5xx_ response code. #8352
- Feature: New lockfile "bundle" feature that can integrate different lockfiles for different configurations and different graphs into a single lockfile bundle that can be used to vastly optimize CI (specially for multiple products), implementing bundle build-order and bundle update operations. #8344 . Docs here
- Fix: Renamed generator *QbsToolchain* to *QbsProfile*. #8537 . Docs here
- Fix: Renamed default filename of _QbsProfile_ generated file to _conan_toolchain_profile_.qbs. #8537 . Docs here
- Fix: Renamed Qbs attribute *use_toolchain_profile* to *profile*. #8537 . Docs here
- Fix: Remove extra spaces in flags and colons in path variables. #8496
- Fix: *conan_v2_error* if *scm_to_conandata* is not enabled. #8447
- Fix: *CONAN_V2_MODE* env-var does not longer alter behavior, only raises errors for Conan 2.0 incompatibilities #8399 . Docs here
- Fix: meson : Add target and jobs arguments. #8384 . Docs here
- Fix: Set *qbs.targetPlatform* with qbs toolchain. #8372
- Fix: Remove warnings for old toolchains imports and `generate_toolchain_files()` calls (use new imports and `generate()` calls. #8361
- BugFix: Improve *tools.unix_path* for Cygwin. #8509
- BugFix: Allow *run_in_windows_bash* in MSYS/Cygwin. #8506
- BugFix: Add some sanity check to avoid a vague error for custom architectures. #8502
- BugFix: Fix Apple M1 detection. #8501

- Bugfix: Fix repeated `build_requires`, including conflicting versions in profile composition or inclusion that repeats `[build_requires]` values. #8463
- Bugfix: Fixing a *CMakeDeps* bug with components, not finding the _conan_macros.cmake_ file. #8445
- Bugfix: Fix exit code for *conan_build_info*. #8408

22.36 1.33.1 (02-Feb-2021)

- Fix: Rename _conan.cfg_ to _global.conf_. #8422 . Docs here
- Fix: Make CMakeDeps generator available in declarative mode `generators = "CMakeDeps"` #8416
- Fix: Make the new Macos subsystem Catalyst lowercase to be consistent with existing subsystems. #8389 . Docs here
- BugFix: Fix Apple Catalyst flags. #8389 . Docs here

22.37 1.33.0 (20-Jan-2021)

- Feature: Introducing a new `[conf]` section in profiles that allows a more systematic configuration management for recipes and helpers (build helpers, toolchains). Introducing a new `conan_conf.txt` cache configuration file that contains configuration definition with the same syntax as in profiles. #8266 . Docs here
- Feature: Add Apple Catalyst support (as new os.subsystem) #8264 . Docs here
- Feature: Add os.sdk sub-settings for Apple #8263 . Docs here
- Feature: Provide support for `msvc` compiler in `MSBuild` tools #8238 . Docs here
- Feature: Specify build modules by the generator in *cpp_info*. Added backwards compatibility for **.cmake* build modules added at global scope, but not for other file extensions. #8232 . Docs here
- Feature: The *tools.get*, *tools.unzip* and *tools.untargz* now accept a new argument *strip_root=True* to unzip moving all the files to the parent folder when all of them belongs to a single folder. #8208 . Docs here
- Feature: Add new `msvc` compiler setting and preliminary support in `conan.tools.cmake` generator and toolchain. #8201 . Docs here
- Feature: `CMakeDeps` now takes values for configurations from `settings.yml`. #8194 . Docs here
- Feature: Implement `ConanXXXRootFolder` in `MSBuildDeps` generator. #8177
- Feature: Add _Meson_ build helper. #8147 . Docs here
- Feature: Add *QbsToolchain* and a Qbs build helper class (currently working for Mcus, not for Android or iOS). #8125 . Docs here
- Feature: Add e2k (elbrus) architectures and mcst-lcc compiler #8032 . Docs here
- Feature: New `CMakeDeps` generator (at the moment is the `cmake_find_package_multi`, that allows custom configurations, like `ReleaseShared`. #8024 . Docs here
- Feature: Allow `MSBuildToolchain` custom configurations in `generate()` method. #7754 . Docs here
- Fix: Fixed help message in command *conan remove –outdated* with reference or pattern #8350 . Docs here
- Fix: Do not define CMAKE_GENERATOR_PLATFORM and CMAKE_GENERATOR_TOOLSET in the `CMakeToolchain` file unless the CMake generator is "Visual Studio". Fix https://github.com/conan-io/conan/issues/7485 #8333

- Fix: Remove spurious '-find' argument to XCodes xcrun tool. #8329
- Fix: Update pylint plugin, some fields are now available in the base *ConanFile*. #8320
- Fix: Remove PyJWT deprecation warning by adding explicitly `algorithms` argument. #8267
- Fix: CMake's generator name for Visual Studio compiler uses only the major version. #8257
- Fix: Remove nosetests support, now using pytest for the test suite. #8253
- Fix: Remove *CMAKE_PROJECT_INCLUDE* in *CMakeToolchain*, no longer necessary as the MSVC runtime can be defined with a generator expression in the toolchain. #8251
- Fix: Temporarily allow `cmake_paths` generator for `CMakeToolchain`, to allow start using the toolchain for users that depend on that generator. #8230
- Fix: Let CMake generator generate code for checking against "ClangCL" msvc toolset. #8218
- Fix: Include `build_requires` in the global `conandeps.props` file generated by MSBuildDeps. #8186
- Fix: Change *MSBuildDeps* file `conan_deps.props` to `conandeps.props` to avoid collision with a package named "deps". #8186
- Fix: Throw error when the recipe description is not a string. #8143
- Fix: Inject build modules after CMake targets are created #8130 . Docs here
- Fix: Define `package_folder` in the `test_package` folder (defaulting to "package"), so the test recipe can execute `cmake.install()` in its `build()` method. #8117
- Fix: Remove the downloaded file if it doesn't satisfy provided checksums (modifies *tools.download*). #8116 . Docs here
- Bugfix: Solved `assert node.package_id != PACKAGE_ID_UNKNOWN` assertion that happened when using `build_requires` that also exist in `requires`, and using `package_revision_mode` and `full_transitive_package_id=1` #8358
- BugFix: Fix SCM user and password by making them url-encoded #8355
- Bugfix: Fix bug in definition of `ROOT` variables in `MakeGenerator`. #8301
- BugFix: fix *-j* being passed to _NMake_ in *AutotoolsBuildEnvironment*. #8285
- BugFix: Fix per-package settings exact match for packages without user/channel. #8281
- BugFix: Fix detected_architecture() for Apple M1, mapping to `armv8` (from returned `arm64`). #8262
- Bugfix: Make prompt names unique when using multiple virtualenv scripts in Powershell. #8228
- Bugfix: Fix when _conandata.yml_ is empty and *scm_to_conandata* is enabled #8215
- Bugfix: Removed a reference to deprecated *FlagsForFile* function in place of current *Settings* function. #8167
- Bugfix: Add test for AutoToolsBuildEnvironment on Apple platforms #8118
- Bugfix: Change the *rpath_flags* flag to always use the comma separator instead of "=", because the current behaviour causes linker error messages when attempting to cross-compile to Mac OS, and the comma separator is accepted everywhere. #7716

22.38 1.32.1 (15-Dec-2020)

- Bugfix: Avoid conflict of user custom generators names with new generators. #8183
- Bugfix: Fix errors when using `conan info --paths` and `short_paths=True` in Windows, due to creation of empty folders in the short-paths storage. #8181
- Bugfix: `conan new <pkg-name>/version -t` wrong include when not using `-s` (using the hardcoded git repo). #8175
- Bugfix: Fix `excludes` pattern case-insensitive in non Windows platforms. #8155
- Bugfix: Enabling set_name, set_version for lockfile roo not location. #8151

22.39 1.32.0 (03-Dec-2020)

- Feature: Generate *<pkgname>-config.cmake* files for lowercase packages to improve case compatibility. #8129 . Docs here
- Feature: Add meson cross-build toolchain. #8111
- Feature: Temporary acquire write permissions in *replace_in_file*. #8107
- Feature: Update **`conan new`** to latest guidelines. #8106
- Feature: Deprecate experimental `toolchain()` in favor of more generic `generate()` method. Deprecate toolchains `write_toolchain_files()` to new `generate()` method. #8101 . Docs here
- Feature: Move the `CMakeToolchain` and new `CMake` experimental helpers to the new `from conan.tools.cmake` import. #8096 . Docs here
- Feature: Move the `MSBuildToolchain` and new `MSBuild` experimental helpers to the new `from conan.tools.microsoft` import. #8096 . Docs here
- Feature: Move the `MakeToolchain` experimental helper to the new `from conan.tools.gnu` import. #8096 . Docs here
- Feature: Add `conan remote list_ref --no-remote` to list recipes without a remote defined. #8094 . Docs here
- Feature: Add `conan remote list_pref --no-remote` to list packages without a remote defined. #8094 . Docs here
- Feature: Add `--lockfile-node-id` argument to `conan install --lockfile` so it can target different packages with same reference (different binary, this can happen with private requirements). #8077 . Docs here
- Feature: Proof that `python_requires` can be used (as a workaround) to affect the `package_id` of consumers of `build_requires` that otherwise will not be rebuilt based on changes. #8076 . Docs here
- Feature: Introduce configuration `general.keep_python_files` to allow packaging of Python .pyc files. #8070 . Docs here
- Feature: Tests for toolchains and Intel compiler. #8062
- Feature: Add recipe and package revision to show a complete Conan reference when generating the *build_info –v2* id fields. #8055
- Feature: Introduce a new *BINARY_INVALID* mode for more flexible definition and management of invalid configurations. #8053 . Docs here
- Feature: Add headers with settings and options to HTTP GET requests when searching for packages. #8046

- Feature: Preliminary experimental support for toolchains with CMake + Visual + Ninja. #8034
- Feature: Allow (experimental) custom configuration of the `msbuild` generator. #8014
- Feature: Rename `msbuild` generator to `MSBuildDeps` and use the new `generate()` method. #8014
- Feature: Make the `conan new bye/0.1 -s -t` to provide variable filenames and messages that include the package name and version, instead of a hardcoded "hello" one. #7989
- Feature: Tagged tests and created a *conftest.py* to run the tests with *pytest* skipping the tests using not available tools (cmake, visual studio...). #7975
- Feature: Provide correct *–pure_c* implementation to **`conan new`**. #7947
- Feature: System package tools can install a list of different packages. #7779 . Docs here
- Feature: meson toolchain #7662 . Docs here
- Feature: Add Conan package name and version to Visual Studio generator properties file. #7645
- Fix: Remove `__init__.py` in the root of the repo, which was useless, without a purpose, but caused issues with other projects importing Conan Python code. #8132
- Fix: Make variables defined in `CMakeToolchain` cache variables, so they can define directly values defined in `CMakeLists.txt`. #8124
- Fix: Remove cryptography, pyopenssl and idna from OSX requirements in Python. #8075
- Fix: Rename the generated file of `MSBuildToolchain` to `conantoolchain.props` so it doesn't collide with a potential `toolchain` package name and the `msbuild` generator. #8073 . Docs here
- Fix: Avoid warning in `msbuild` generator importing multiple times the same .props file due to transitive dependencies. #8072
- Fix: Set username or password individually in git SCM with ssh. #8016
- Fix: When using lockfiles, allow `config_options` and `configure` to compute different options as long as the final evaluated values match the locked ones. #7993
- Fix: Make the `conan new --pure_c` pure C template to remove both `compiler.libcxx` and `compiler.cppstd` settings, as described in the docs. #7989
- BugFix: Fix linkage to a same global target of different package components in *cmake_find_package/_multi* generators. #8114 . Docs here
- Bugfix: Solve `os.rename` crash when using `short_paths` with a short path storage located in another Windows drive unit. #8103
- BugFix: Allow lockfiles to be relaxed with the *–build* argument. #8054 . Docs here
- Bugfix: Append existing `LocalDebuggerEnvironment` in `msbuild` generator. #8040
- Bugfix: Remove correctly short-paths folders in Windows. #7986

22.40 1.31.4 (25-Nov-2020)

- Feature: Add new *CONAN_CMAKE_SYSROOT* environment variable to enable the definition of sysroot from environment, without abusing *CONAN_CMAKE_FIND_ROOT_PATH*. #8097 . Docs here
- Bugfix: remove definition of sysroot from *CONAN_CMAKE_FIND_ROOT_PATH*. #8097 . Docs here
- Bugfix: Bugfix: Solve os.rename crash when using short_paths with a short path storage located in another Windows drive unit. Ported from: #8103

22.41 1.31.3 (17-Nov-2020)

- Bugfix: Fix addition of CMAKE_SYSTEM_NAME for SunOS and AIX 64->32 bits builds #8059

22.42 1.31.2 (11-Nov-2020)

- Bugfix: Recent `liburl3` 1.26 library updates is breaking the constraints in Conan `requirements.txt` as `requests` 2.24 has a limitation for `liburl3`. This PR constrains `liburl3` version to be less than 1.26, so it does not break with requests 2.24. #8042

22.43 1.31.1 (10-Nov-2020)

- Fix: Bump _cryptography_ dependency in MacOS to equal or later than 3.2. #7962
- Bugfix: Fix a problem with the `init()` function not being called when the recipe loader uses some cached data, which can happen when using lockfiles and with `python_requires`. #8018
- Bugfix: Fixed `self.copy()` incorrectly handling `ignore_case`. #8009
- Bugfix: Fixed wrong `ignore_case` default in `[imports]` section of *conanfile.txt*. #8009
- Bugfix: Do not try to encrypt a *None* value when using *CONAN_LOGIN_ENCRYPTION_KEY* environment variable. #8004

22.44 1.31.0 (30-Oct-2020)

- Feature: Add argument *conanfile* to *pre_download_package* and *post_download_package* hook functions. #7968 . Docs here
- Feature: Add *CONAN_LOGIN_ENCRYPTION_KEY* environment variable to obfuscate stored auth token. #7958 . Docs here
- Feature: Use profile to filter results in the **`conan search`** HTML output. #7956
- Feature: Changed recommended way to launch test suite, with pytest over nosetests. #7952
- Feature: Provide a `MSBuildCmd` helper class that encapsulates calling MSBuild. #7941 . Docs here
- Feature: Download and keep the `conan_export.tgz` and `conan_source.tgz` in the cache, so they are not affected by different Operating Systems compression and de-compression and uploading is way more efficient. #7938
- Feature: Add *provides* and *deprecated* fields to **`conan info`** output #7916

- Feature: Including package revision information in output from **`conan info`** (when revisions are enabled). #7890
- Feature: Download and keep the conan_package.tgz in the cache, so they are not affected by different Operating Systems compression and de-compression and uploading is way more efficient. #7886
- Feature: Add POC on a toolchain for iOS (using CMake XCode generator). #7855 . Docs here
- Feature: Add POC on a toolchain for Android (using CMake provided modules). #7843 . Docs here
- Feature: Allow `conan config install` of a single file #7840 . Docs here
- Feature: Use Python loggers for Conan output in cli 2.0. #7502
- Fix: Improve permission error message when migrating cache folder. #7966
- Fix: Make per-package settings definition complete the existing settings values, not requiring a complete redefinition. #7953
- Fix: Avoid unnecessary extra loading of *conan.conf* file in the version migrations check. #7949
- Fix: Simplified MakeToolchain to remove things that were not checked by tests or unused. #7942
- Fix: displayed message when settings of the recipe are constrained. #7930 . Docs here
- Fix: Set *CMAKE_SYSTEM_NAME* set to *iOS*, *tvOS* or *watchOS* or *Darwin* depending on the CMake version. #7924
- Fix: Remove duplicate entries while modifying PATH-like environment variables internally. Especially important for Windows where system PATH size is limited by 8192 charachers (when using cmd.exe). #7891
- Fix: Make default behaviour explicit in search help output. #7877 . Docs here
- Fix: Automatically add OSX deployment flags in `AutootoolsBuildEnvironment` with the value of `os_version`, unless the values are already defined in environment variables *CFLAGS* or *CXXFLAGS*. #7862
- Fix: Remove toolset variability from the `msbuild` generator and `MSBuildToolchain`. #7825
- Fix: Component requirement checking now properly handles private and override requirements. #7585
- Bugfix: Set default storage_folder to .conan/data in case if storage_path entry fails to be defined by conan.conf. #7910
- Bugfix: Fix regression in self.run(output=xxxx) that have a write() method but do not wrap a stream. #7905
- Bugfix: Fix local flow (conan install + build) support for `cpp_info.names` and `cpp_info.filenames`. #7867
- Bugfix: Fix `inspect --remote` forcing to retrieve the remote for evaluation, overwriting what is in the local cache. #7749
- Bugfix: Copy symbolic links to directory with deploy generator. #7655 . Docs here

22.45 1.30.2 (15-Oct-2020)

- Feature: Supports Clang 11. #7871 . Docs here
- Bugfix: Fix regression https://github.com/conan-io/conan/issues/7856, `imports` failing to match subfolders in Windows due to backslash differences. #7861
- Bugfix: Allow defining new options values when creating a new lockfile from an existing base lockfile. #7859

22.46 1.30.1 (09-Oct-2020)

- Fix: Use quotes around the install path, it can contain spaces. #7842
- Fix: prefix intel functions with `intel_` because they are now exposed via tools. Fixes https://github.com/conan-io/conan/issues/7820. #7821 . Docs here
- Bugfix: Fix regression introduced in 1.30 (https://github.com/conan-io/conan/pull/7673), with incorrect matches of user/channel for version ranges. #7847
- Bugfix: Fix `CMakeToolchain` with multiple variables definitions. #7833
- Bugfix: Check comparing the host and the build architecture to decide if cross building and set *CMAKE_SYSTEM_NAME* in the `CMake` build helper. #7827

22.47 1.30.0 (05-Oct-2020)

- Feature: Implement real detection of `compiler.libcxx` value for `gcc` compiler. Only enabled in *CONAN_V2_MODE*, otherwise it would be breaking. #7776
- Feature: Added [Depends](https://doc.qt.io/qbs/qml-qbslanguageitems-depends.html) items for every public dependency of conanfiles requires/dependencies. #7729 . Docs here
- Feature: Instructions on how to run conan tests against a real Artifactory server. #7697
- Feature: List *cpp_info.names* and *cpp_info.filenames* in JSON and Markdown generator. #7690 . Docs here
- Feature: Add information about components to *markdown* generator. #7677
- Feature: New experimental `MSBuildToolchain` to generate `conan_toolchain.props` files (it is multi-config, will generate one specific toolchain file per configuration) for more transparent integration and better developer experience with Visual Studio. #7674 . Docs here
- Feature: Allow packages that do not declare components to depend on other packages components and manage transitivity correctly, with the new `self.cpp_info.requires` attribute. #7644 . Docs here
- Feature: Add MacOS 11 ("Big Sur") support. #7601 . Docs here
- Feature: Expose intel_installation_path, compilervars, compilervars_dict, and compilervars_command under tools module in order to support usage of the intel compiler. #7572 . Docs here
- Feature: Allow user-defined generators to be installed and used from the Conan cache. #7527 . Docs here
- Feature: Add **`conan remote`** proposal for cli 2.0. #7401
- Fix: Allow usage of MD5 checksums in FIPS systems that would raise error otherwise. #7807
- Fix: Fix capture output when running tests that call the ConanRunner in the conanfile. #7799
- Fix: Consider absolute paths when parsing *conanbuildinfo.txt* #7797
- Fix: Update parallel uploads help message. #7785 . Docs here
- Fix: Removed check in lockfiles computed from other lockfile that it should be part of it. Users can check the resulting lockfile themselves if they want to. #7763 . Docs here
- Fix: Extend help message indicating how to run **`conan export`** without *user/channel*. #7757 . Docs here
- Fix: Conan copy shows better description when using full reference for destination. #7741
- Fix: Do not capture output for normal conan run (no logging or testing) when launching processes via *ConanRunner* so that color from build tools output is not lost. #7740

- Fix: *self.copy()* follows *igore_case* correctly on Windows. #7704 . Docs here
- Fix: Use patterns in server query when resolving version ranges. #7673
- Fix: Raising conflict errors when options doesn't match in the evaluation of graphs with lockfiles. #7513
- Bugfix: Fixed bug where uploading to multiple remotes in a single conan upload command would fail. #7781
- BugFix: Add `armv5hf` and `armv5el` to the Android ABI architectures. #7730
- Bugfix: Correctly inherit and use `system_requirements` when using `python_requires`. #7721
- Bugfix: Translate *settings.os* value *Macos* to *Darwin* for *CMAKE_SYSTEM_NAME* to allow compiling CMake-based packages for MacOS. #7695

22.48 1.29.2 (21-Sept-2020)

- Feature: Add support for apple-clang 12.0. #7722 . Docs here

22.49 1.29.1 (17-Sept-2020)

- Bugfix: Fix `pkg_config` generator adding to .pc files empty include and lib dirs. #7703
- Bugfix: Fix non existing (failed import) `tools.remove_files_by_mask`. #7700
- BugFix: Removed lockfile checking build_requires when they come from 2 different origins: profiles and recipes. #7698
- Bugfix: CMake build helper: Use actual CMake generator version to append platform generator, instead of the Conan setting *compiler.version*. #7684

22.50 1.29.0 (02-Sept-2020)

- Feature: Add QNX Neutrino version 7.1 to settings. #7627
- Feature: Added support for *cpp_info.system_libs*, *cpp_info.framework_paths* and *cpp_info.frameworks* for `qbs` generator. #7619
- Feature: Provide useful information trying to compute the build order using a *–base* lockfile. #7551
- Feature: Add *user_info_build* field to JSON generator. #7550
- Feature: *PkgConfig* tools now exposes the packages's version as property. #7534 . Docs here
- Feature: Support from iOS 13.2 to 13.6. #7507 . Docs here
- Feature: Add an experimental toolchain for gnu make. #7430 . Docs here
- Feature: New `tools.rename` function to rename a file or folder to avoid 'Access is denied' on Windows. #6774 . Docs here
- Fix: Fix *conan info –build-order* deprecation message. #7632
- Fix: Set CMake targets compile options based on language #7600
- Fix: Support installing configs from non-regular files. #7583 . Docs here
- Fix: Update docs in *conan info -bo* command. #7570

- Fix: Relax python six dependency to allow 1.15. #7538
- Fix: Add pre-release versions when resolving *required_conan_version*. #7535
- Fix: Adds support of URL-like git ssh syntax. #7509
- Fix: Improve message of missing dependencies for components. #7483
- Fix: Changed _requirements.txt_ to include distro package version 1.5.0. #7461
- Fix: Avoid requiring the existence of all `conanbuildinfo_xxx.cmake` files in `cmake_multi` generator. #7376
- Bugfix: Fix *cpp_info* filename in FindPackageHandleStandardArgs for cmake_find_package generator. #7610
- Bugfix: Avoid marking as "modified" packages in a lockfile computed from a base lockfile. #7592
- Bugfix: Update correctly "Package_ID_Unknown" nodes when using `conan lock update` and `package_revision_mode`. #7592
- Bugfix: Respect *winsdk_version* for WindowsStore. #7584
- Bugfix: Fix frameworks usage with components for *cmake_find_package_multi* generator. #7580
- Bugfix: Support *frameworks* and *framework_paths* in _qmake_ generator. #7579
- Bugfix: Provide a more descriptive error when an unknown statement is added to a profile #7577
- Bugfix: Add support for *cpp_info.system_libs* to _QMake_ generator. #7563
- Bugfix: Make frogarian show up as a whole (not sliced) on linux terminal. #7553
- Bugfix: Fix import of *collections.Iterable* compatible with Python2. #7545
- Bugfix: Propagate the global version of the recipe for components. #7524
- Bugfix: Use *CMAKE_FIND_ROOT_PATH_BOTH* to locate frameworks. #7515

22.51 1.28.2 (31-Aug-2020)

- Fix: Fix import of `six.moves.collections_abc` non existing for some six versions. #7622
- Fix: Add system libs and frameworks to components targets in *cmake_find_package* and *cmake_find_package_multi* generators. #7611
- Bugfix: Fix *cpp_info* filename in FindPackageHandleStandardArgs for cmake_find_package generator. #7625
- Bugfix: Fix regression in `deps_cpp_info` incorrectly adding directories when reading from `conanbuildinfo.txt` file. #7599

22.52 1.28.1 (06-Aug-2020)

- Feature: Add *user_info_build* attribute to *txt* generator. #7488
- Fix: Attribute *user_info_build* is available for commands in the local development workflow. #7488
- Fix: Do not override value of *public_deps* in *pkg_config* generator. #7482
- Bugfix: correctly set *CMAKE_OSX_SYSROOT* and *CMAKE_OSX_ARCHITECTURES*. #7512
- Bugfix: When using `build_requires` defined in a profile that is passed as `profile_host`, it was not being applied to `build_requires` that live in the host context (with `force_host_context=True`). #7500

- Bugfix: Fix broken `cmake_find_package_multi` when using components, as different configurations were being resolved to the same name, overwriting each other. #7492
- Bugfix: Powershell files generated by *virtualenv* generators use proper path separators. #7472

22.53 1.28.0 (31-Jul-2020)

- Feature: Show Conan version on HTML output. #7443 . Docs here
- Feature: Support for *cpp_info.components* in *pkg_config* generator. #7413 . Docs here
- Feature: Adds ps1 virtualenv to other OS for use with powershell 7. #7407 #7408 . Docs here
- Feature: Propose `init()` method to unconditionally initialize class attributes like `license` or `description`. #7404 . Docs here
- Feature: add deprecated attribute #7399 . Docs here
- Feature: Allow `conan.conf` user configuration of paths to client certificate and key, outside of the Conan cache. #7398 . Docs here
- Feature: Document return value of *self.copy()* in the *package()* method. #7389 . Docs here
- Feature: Complete cli2.0 framework to handle sub-commands and add **`conan user`** command for cli 2.0 #7372
- Feature: Implement `required_conan_version` in `conanfile.py`, will raise if the current Conan version does not match the defined version range. #7360 . Docs here
- Feature: Add *provides* attribute to *ConanFile*: recipes can declare what they provide and Conan will fail if several recipes provide the same functionality (ODR violation). #7337 . Docs here
- Feature: When using *CONAN_V2_MODE* if build_type or compiler are not defined Conan will raise an error. #7327 . Docs here
- Feature: Adds "filenames" to cppinfo attribute, and changes *cmake_find_package* and *cmake_find_package_multi* generators so that they support it. #7320 . Docs here
- Feature: Define `recipe_folder` attribute pointing to the folder containing `conanfile.py` #7314 . Docs here
- Feature: Checking if a Linux distro uses *apt* is now based on the existence of *apt* in the system, instead of checking if the distro currently being used is in a hard-coded list of distros known to use *apt*. #7309
- Feature: Add commands management for cli 2.0. #7278
- Feature: Complete revamp of the **lockfiles** feature. Including version-only lockfiles, partial lockfiles, new command line syntax, improved management of build-order and many pending fixes. #7243 . Docs here
- Feature: More detailed description for *–update* argument. #7167 . Docs here
- Feature: improve compiler detection for *CONAN_V2_MODE*. #5740 . Docs here
- Feature: Add settings for clang-cl (clang on Windows). #5705 . Docs here
- Fix: Relax `pluginbase` requirement to `pluginbase>=0.5`, including latest 1.0.0 . #7441
- Fix: Make explicit the file writing of `toolchain()` helpers, so the method can be used to save custom files. #7435 . Docs here
- Fix: Fixing *–help* for commands in proposal for command line v2.0. #7394
- Fix: Show outdated packages when running *search –table*. #7364 . Docs here
- Fix: Relax `msbuild` generator to not raise in Linux. #7361
- Fix: Conan config install does not trigger scheduled config command. #7311

- Fix: Implement missing `__contains__` method, so checking `if "myoption" in self.info.options` is possible in `package_id()`. #7303
- Fix: Build first ocurrence of a node in a lockfile when it is repeated (build requires) #7144
- BugFix: Only add User-Agent to headers dict if it was not provided by the user. #7390
- Bugfix: *cppstd* was missing in *settings.yml* for the qcc compiler and updates to 8.3. #7384
- BugFix: Fix missing download of `conan_sources.tgz` created using `export_sources()` method. #7380
- Bugfix: Intel Compiler install location detection on Windows. #7370
- Bugfix: Avoid crash while computing `package_id` when using `package_revision_mode`, and also incorrectly using installed binaries and reporting them installed after the re-computation of `package_id` resolved to a different binary. #7353
- Bugfix: cmake_multi generator used with Xcode CMake generator. #7341
- Bugfix: Do not fail for *conan remove -r remote -p* when there are no packages in the remote. #7338
- Bugfix: Add `system_libs` to `scons` generator. #7302

22.54 1.27.1 (10-Jul-2020)

- Bugfix: Recover quotes around linker flags in CMake generators, fix failure with Macos frameworks #7322

22.55 1.27.0 (01-Jul-2020)

- Feature: (Only if using two profiles) Information from the *self.user_info* field is provided to consumers: information from the _host_ context is accessible via *deps_user_info* attribute, and information from the _build_ context via *user_info_build* attribute. #7266 . Docs here
- Feature: New `conan config install --list` and `conan config install --remove=index` arguments to display and remove conan config install origins. #7263 . Docs here
- Feature: Support components for *cmake_find_package_multi* generator. #7259 . Docs here
- Feature: Add Pop!_OS to the list of APT based distributions. #7237
- Feature: Use Bootstrap in search table template style. #7224
- Feature: Added support for template dir in **`conan new`**. #7215 . Docs here
- Feature: Configuration for checking the required Conan client version. #7183 . Docs here
- Feature: Adds tool to fix symlinks in the *package_folder*. #7178 . Docs here
- Feature: Templates for *conan search –table* and *conan info –graph* can be overridden by the user. #7176 . Docs here
- Feature: Add support for the *CLICOLOR/CLICOLOR_FORCE/NO_COLOR* output colorization control variables. #7154 . Docs here
- Fix: Remove message from the qmake generator. #7228
- Fix: Allow `--build=Pkg/0.1@` to match the `Pkg/0.1` package, so the `conan install Pkg/0.1@ --build=Pkg/0.1@` also works. #7219
- Fix: Improve error message when svn or git are not in the installed or in the path. #7194
- Fix: Graph created for the *test_package/conanfile.py* recipe takes the *profile:build* if given. #7182

- Fix: Define user variables in the `conan_toolchain.cmake` file, not in the project-include file. #7160
- Fix: Set toolset for MSBuild in case of Intel C++. #6809
- Bugfix: Allow to extend classes with `python_requires_extend` from packages that contain "." dots in the package name. #7262
- Bugfix: Correctly inherit `scm` definitions from `python_requires` base classes. #7238
- Bugfix: Change GNU triplet for iOS, watchOS, tvOS to allow simulator builds. #6748
- SCM mode with `scm_to_conandata` and revisions marked as stable. Docs here

22.56 1.26.1 (23-Jun-2020)

- Fix: Add missing migrations. #7213
- Fix: Packages listed as *build_requires* in recipes that belong to the _host_ context don't add as *build_requires* those listed in the _host_ profile. #7169

22.57 1.26.0 (10-Jun-2020)

- Feature: Expose *msvs_toolset* tool. #7134 . Docs here
- Feature: Add components to *cmake_find_package* generator. #7108 . Docs here
- Feature: Add *stdcpp_library* tool. #7082 . Docs here
- Feature: Add remove_files_by_mask helper #7080 . Docs here
- Feature: New `toolchain()` recipe method, as a new paradigm for integrating build systems, and simplifying developer flows. #7076 . Docs here
- Feature: New experimental `msvc` generator that generates a .props file per dependency and is also multi-configuration. #7035 . Docs here
- Feature: Add *conan config init* command. #6959 . Docs here
- Feature: Add `export()` and `export_sources()` methods, that provide the `self.copy()` helper to add files to recipe or sources in the same way as the corresponding attributes. #6945 . Docs here
- Feature: Allow access to `self.name` and `self.version` in `set_name()` and `set_version()` methods. #6940 . Docs here
- Feature: Use a template approach for the *html* and *dot* output of the Conan graph. #6833
- Feature: Handle C++ standard flag for Intel C++ compiler. #6766
- Feature: Call compilervars.sh within CMake helper (Intel C++). #6735 . Docs here
- Feature: Pass command to Runner as a sequence instead of string. #5583 . Docs here
- Fix: JSON-serialize sets as a list when using *conan inspect –json*. #7151
- Fix: Update the lockfile passed as an argument to the install command instead of the default *conan.lock*. #7127
- Fix: Adding a package as editable stores full path to *conanfile.py*. #7079
- Fix: Fix broken test *PkgGeneratorTest*. #7065
- Fix: Fix wrong naming of variables in the `pkg_config` generator. #7059
- Fix: Do not modify *scm* attribute when the *origin* remote cannot be deduced. #7048

- Fix: *vcvars_dict* should accept a conanfile too. #7010 . Docs here
- Fix: `conan config install` can overwrite read-only files and won't copy permissions. #7004
- Fix: Better error message for missing binaries, including multiple "–build=xxx" outputs. #7003
- Fix: Add quotes to folders to accept paths with spaces when calling pyinstaller. #6955
- Fix: Previously *conan* always set *cpp_std* option in *meson* project, even if *cppstd* option was not set in *conan* profile. Now it sets the option only if *cppstd* profile option has a concrete value. #6895
- Fix: Handle compiler flags for Intel C++ (AutoToolsBuildEnvironment, Meson). #6819
- Fix: Set the default CMake generator and toolset for Intel C++. #6804
- Bugfix: Fix iOS CMake architecture. #7164
- Bugfix: Getting attribute of `self.deps_user_info["dep"]` now raise `AttributeError` instead of a (wrong) `KeyError`, enabling `hasattr()` and correct `getattr()` behaviors. #7131
- Bugfix: Fix crash while computing the `package_id` of a package when different `package_id_mode` are mixed and include `package_revision_mode`. #7051
- Bugfix: Do not allow uploading packages with missing information in the *scm* attribute. #7048
- Bugfix: Fixes an issue where Apple Framework lookup wasn't working on *RelWithDebInfo* CMake build types. #7024
- Bugfix: Do not check patch compiler version in the `cmake` generators. #6976

22.58 1.25.2 (19-May-2020)

- Bugfix: Previously conan always set `cpp_std` option in meson project, even if `cppstd` option was not set in conan profile. Now it sets the option only if `cppstd` profile option has a concrete value. #7047
- Bugfix: Fix deploy generator management of relative symlinks. #7044
- Bugfix: Fixes an issue where Apple Framework lookup wasn't working on RelWithDebInfo. #7041
- Bugfix: Fix broken `AutoToolsBuildEnvironment` when a profile:build is defined. #7032

22.59 1.25.1 (13-May-2020)

- Feature: Add missing gcc versions: 6.5, 7.5, 8.4, 10.1. #6993 . Docs here
- Bugfix: Resumable download introduced a bug when there is a fronted (like Apache) to Artifactory or other server that gzips the returned files, returning an incorrect `Content-Length` header that doesn't match the real content length. #6996
- Bugfix: Set `shared_linker_flags` to CMake `MODULE` targets too in `cmake` generators, not only to `SHARED_LIBRARIES`. #6983
- Bugfix: Fix *conan_get_policy* return value. #6982
- Bugfix: Fix json output serialization for `cpp_info.components`. #6966

22.60 1.25.0 (06-May-2020)

- Feature: Consume `settings_build` to get the value of the OS and arch from the `build` machine (only when `--profile:build` is provided). #6916 . Docs here
- Feature: Implements `cpp_info.components` dependencies. #6871 . Docs here
- Feature: Change HTML output for *conan search –table* command. #6832 . Docs here
- Feature: Execute periodic config install command. #6824 . Docs here
- Feature: Add *build_modules* to markdown generator output. #6800
- Feature: Resume interrupted file downloads if server supports it. #6791
- Feature: Using *CONAN_V2_MODE* the *version* attribute in a *ConanFile* is always a string (already documented). #6782 . Docs here
- Feature: Support GCC 9.3. #6772 . Docs here
- Feature: Populate *settings_build* and *settings_target* in conanfile (only if provided `--profile:build`). #6769 . Docs here
- Feature: handle C++ standard for Intel C++ compiler #6766
- Feature: add Intel 19.1 (2020). #6733
- Fix: *tools.unix_path* is noop in all platforms but Windows (already documented behavior). #6935
- Fix: Preserve symbolic links for deploy generator. #6922 . Docs here
- Fix: Adds missing version GCC 10 to default settings. #6911 . Docs here
- Fix: Populate *requires* returned by the servers from the search endpoint using *requires* (Artifactory) or *full_requires* (conan_server) fields. #6861
- Fix: Avoid failures that happen when Conan runs in a non-existing folder. #6825
- Fix: Use pep508 environment markers for defining Conan pip requirements. #6798
- Fix: Improve error message when `[options]` are not specified correctly in conanfile.txt. #6794
- Fix: add missing compiler version check for Intel. #6734
- Bugfix: Prevent crash when mixing package_id modes for the same dependency. #6947
- BugFix: Propagate arch parameter to `tools.vcvars_command()` in *MSBuild()* build helper. #6928
- Bugfix: Fix the output of **`conan info`** package folder when using `build_id()` method. #6917
- Bugfix: Generate correct PACKAGE_VERSION in `cmake_find_package_multi` generator for multi-config packages. #6914
- Bugfix: enable C++20 on Apple Clang. #6858
- Bugfix: Variable *package_name* in *conan new -t <template>* command contains a _CamelCase_ version of the name of the package. #6821 . Docs here
- Bugfix: Changed the CMake generator template to properly handle exelinkflags and sharedlinkflags using generator expressions. #6780

22.61 1.24.1 (21-Apr-2020)

- Bugfix: correct the *cmake* generator target name in the *markdown* generator output. #6788
- Bugfix: Avoid *FileNotFoundError* as it is not compatible with Python 2. #6786

22.62 1.24.0 (31-Mar-2020)

- Feature: Add the needed command-line arguments to existing commands to provide information about host and build profiles. #5594 . Docs: here
- Feature: Add *markdown* generator, it exposes useful information to consume the installed packages. #6758 . Docs here
- Feature: Add new tool *cppstd_flag* to retrieve the compiler flag for the given settings. #6744 . Docs here
- Feature: Short paths feature is available for Cygwin. #6741 . Docs here
- Feature: Add Apple Clang as a base compiler for Intel C++. #6740 . Docs here
- Feature: Make *settings.get_safe* and *options.get_safe* accept a default value. #6739 . Docs here
- Feature: *CONAN_V2_MODE* deprecates two legacy ways of reusing python code: the *<cache>/python* path and the automatic *PYTHONPATH* environment variable. #6737 . Docs here
- Feature: Add the _description_ field to the output of the **`conan info`** command. #6724 . Docs here
- Feature: Add more detailed information when there are *missing* packages. #6700 . Docs here
- Feature: Support mirrors for *tools.download* and *tools.get*. #6679 . Docs here
- Feature: Modify the default behaviour in *SystemPackageTool* to be able to create a recipe that does not install system requirements by default if the *CONAN_SYSREQUIRES_MODE* is not set. #6677 . Docs here
- Feature: Add *cpp_info.components* package creator interface to model internal dependencies inside a recipe. #6653 . Docs here
- Feature: Add a new `init()` method to `conanfile.py` recipes that can be used to add extra logic when inheriting from `python_requires` classes. #6614 . Docs here
- Fix: Add Sun C compiler version 5.15 into default settings.yml. #6767
- Fix: Raises *ConanException* when package folder is invalid for *export-pkg*. #6720 . Docs here
- Fix: Added print to stderr and exit into pyinstaller script when it detects python usage of python 3.8 or higher as currently pyinstaller does not support python 3.8. #6686
- Fix: Improve the command line help for the *conan install –build* option. #6681 . Docs here
- Fix: Add build policy help for *–build* argument when used in *conan graph build-order* command. #6650
- Fix: Remove file before copying in `conan config install` to avoid permission issues. #6601
- Fix: check_min_cppstd raises an exception for an unknown compiler. #6548 . Docs here
- Fix: cmake_find_package no longer seeks to find packages which are already found. #6389
- Bugfix: Fixes the auto-detection of `sun-cc` compiler when it outputs `Studio 12.5 Sun C`. #6757
- Bugfix: Add values to `definitions` passed to `MSBuild` build helper which values are not None (0, False…). #6730

- Bugfix: Include name and version in the data from `conanbuildinfo.txt`, so it is available in `self.deps_cpp_info["dep"].version` and `self.deps_cpp_info["dep"].name`, so it can be used in **`conan build`** and in `test_package/conanfile.py`. #6723 . Docs here
- Bugfix: Fix *check_output_runner()* to handle dirs with whitespaces. #6703
- Bugfix: Fix vcvars_arch usage before assignment, that can cause a crash in `tools.vcvars_command()` that is also used internally by `MSBuild` helper. #6675
- Bugfix: Silent output from cmake_find_package generator with *CONAN_CMAKE_SILENT_OUTPUT*. #6672
- Bugfix: Use always LF line separator for .sh scripts generated by `virtualenv` generators. #6670
- Bugfix: Use the real settings value to check the compiler and compiler version in the `cmake` generator local flow when the `package_id()` method changes values. #6659

22.63 1.23.0 (10-Mar-2020)

- Feature: New `general.parallel_download=<num threads>` configuration, for parallel installation of binaries, to speed up populating packages in a cache. #6632 . Docs here
- Feature: Fixed inability to run execute *test* and *install* separately, that is, without *build* step. Added *meson_test()* method, which executes *meson test* (compared to *ninja test* in *test()*). Added *meson_install()* method, which executes *meson install* (compared to *ninja install* in *install()*). #6574 . Docs here
- Feature: Update python six dependency to 1.14.0. #6507
- Feature: Add environment variable 'CONAN_V2_MODE' to enable Conan v2 behavior. #6490 . Docs here
- Feature: Implement *conan graph clean-modified* subcommand to be able to clean the modified state of a lockfile and re-use it later for more operations. #6465 . Docs here
- Feature: Allow building dependency graphs when using lockfiles even if some requirements are not in the lockfiles. This can happen for example when `test_package/conanfile.py` has other requirements, as they will not be part of the lockfile. #6457 . Docs here
- Feature: Implement a new package-ID computation that includes transitive dependencies even when the direct dependencies have remove them, for example when depending on a header-only library that depends on a static library. #6451 . Docs here
- Fix: inspect command can be executed without remote.json (#6558) #6559
- Fix: Raise an error if `MSBuild` argument `targets` is not a list, instead of splitting a string passed as argument instead of a list. #6555
- Bugfix: Check the *CMP0091* policy and set *CMAKE_MSVC_RUNTIME_LIBRARY* accordingly to *CONAN_LINK_RUNTIME* if it's set to *NEW*. #6626
- Bugfix: Fix error parsing *system_libs* from *conanbuildinfo.txt* file. #6616
- Bugfix: Environment variables from the profiles are not set in the _conaninfo.txt_ file of the packages exported with the *export-pkg* command. #6607
- BugFix: Set the `self.develop=True` attribute for recipes when they are used with **`conan export-pkg`**, in all methods, it was previously only setting it for the `package()` method. #6585
- Bugfix: set CMAKE_OSX_DEPLOYMENT_TARGET for iOS, watchOS and tvOS. #6566
- Bugfix: Parse function of GCC version from command line now works with versions *>=10*. #6551
- Bugfix: improve Apple frameworks lookups with CMake integration #6533

22.64 1.22.3 (05-Mar-2020)

- Bugfix: Fixed crashing of recipes using both `python_requires` and `build_id()`. #6618
- Bugfix: Conan should not append generator_platform to the Visual Studio generator if it is already specified by the user. #6549

22.65 1.22.2 (13-Feb-2020)

- Bugfix: Do not re-evaluate lockfiles nodes, only update the package reference, otherwise the build-requires are broken. #6529
- Bugfix: Fixing locking system for metadata file so it can be accessed concurrently. #6524

22.66 1.22.1 (11-Feb-2020)

- Fix: Increase `six` version to allow more modern releases. #6509
- Fix: remove *GLOBAL* from targets to avoid conflicts when using *add_subdirectory*. #6488 . Docs here
- Fix: Avoid caching revision "0" under api V2 (revisions enabled) in the download cache. #6475 . Docs here
- Bugfix: Manage the `dirty` state of the cache package folder with **`conan export-pkg`**. #6498
- BugFix: Add `system_libs` to `premake` generator. #6495
- Bugfix: Upload was silently skipping exceptions that could leave the packages dirty. Long uploads or large compressing times in non-terminals (piped output, like in CI systems) crashed, leaving packages dirty too, but not reporting any error. #6486
- BugFix: Add quotes to `virtualenv` scripts, so they don't crash in pure sh shells. #6265

22.67 1.22.0 (05-Feb-2020)

- Feature: Set conan generated CMake targets as *GLOBAL* so that they can be used with an *ALIAS* for consumers. #6438 . Docs here
- Feature: Deduce *compiler.base.runtime* for Intel compiler settings when using Visual Studio as the base compiler. #6424
- Feature: Allow defining an extra user-defined properties .props file in `MSBuild` build helper. #6374 . Docs here
- Feature: Force the user to read that Python 2 has been deprecated. #6336 . Docs here
- Feature: Add opt-in *scm_to_conandata* for the SCM feature: Conan will store the data from the SCM attribute in the *conandata.yml* file (except the fields *username* and *password*). #6334 . Docs here
- Feature: Implement a download cache, which can be shared and concurrently used among different conan user homes, selectable configuring `storage.download_cache` in `conan.conf`. #6287 . Docs here
- Feature: Some improvements in the internal of lockfiles. Better ordering of nodes indexes. Separation of `requires` and `build-requires`. Better `status` field, with explicit `exported`, `built` values. #6237

- Feature: `imports` functionality can import from "symbolic" names, preceded with @, like @bindirs, @libdirs, etc. This allows importing files from variable package layouts, including custom `package_info()` layouts (like `cpp_info.bindirs = ["mybin"]` can be used with `src="@bindirs"`), and editable package layouts #6208 . Docs here
- Feature: Improve output messages for parallel uploads: the text of the uploaded files contains to which packages they belong and the output for CI is clearer. #6184
- Feature: Adds `vcvars_append` variable (defaulting to `False`) to `CMake` and `Meson` build helpers constructors, so when they need to activate the Visual Studio environment via `vcvars` (for Ninja and NMake generators), the `vcvars` environment is appended at the end, giving precedence to the environment previously defined. #6000 . Docs here
- Fix: Use CCI package reference for example command. #6463
- Fix: Generators *cmake* and *cmake_multi* use the name defined in *cpp_info.name* (reverts change from 1.21.1 as stated). #6429
- Fix: Cleaning `LD_LIBRARY_PATH` environment in `SCM` commands for "pyinstaller" installations, as SSL can fail due to using old SSL stuff from Conan instead from git/svn. #6380
- Fix: Recipe substitution for *scm* (old behavior) fixed for multiline comments in Python 3.8. #6355 . Docs here
- Fix: Avoid warning in "detect" process with Python 3.8, due to Popen with `bufsize=1` #6333
- Fix: Propagate server error (500) in *checksum_deploy*. #6324
- Fix: Fixed wrong CMake command line with `-G Visual Studio 15 ARM` for `armv8` architectures. #6312
- Fix: Add all the system_libs and requirements to the CMake targets constructed by the generators. It will impact header-only libraries that are consumed using targets (previously they were missing some information). #6298
- Fix: Avoid WindowsStore `tools.vcvars()` management when the environment is already set. #6296
- Fix: When the token is empty, and `conan user myuser -p=mypass -r=remote` is used, the user-password are send in HttpBasic so it can be used for completely protected servers that do not expose the ping endpoint. #6254
- Fix: Add *cpp_info.<config>* information to *cmake_find_package_multi* and *cmake_find_package* generators. #6230 . Docs here
- Fix: Multi-generators cannot be used without *build_type* setting. A failure is forced to *cmake_find_package_multi* and *visual_studio_multi* as it was in *cmake_multi*. #6228
- Fix: Fix typo in error message from `tools.get()`. #6204
- Fix: Raise error for symlinks in Windows that point to a different unit. #6201
- BugFix: Avoid included profiles overwriting variables in the current profile. #6398
- Bugfix: Lockfiles were not correctly applying locked `options` to packages, which produced incorrect evaluation of `requirements()` method. #6395
- Bugfix: Fix broken compression of .tgz files due to Python 3.8 changing tar default schema. #6355 . Docs here
- Bugfix: Include MacOS frameworks definitions in autotools LDFLAGS (also Meson). #6309
- Bugfix: Apply `system_libs` information in autotools build helper. #6309
- Bugfix: The `environment_append()` helper does not modify the argument anymore, which caused problems if the argument was reused. #6285
- Bugfix: Include "Package ID Unknown" nodes in `conan graph build-order`, as they need to be processed in that order. #6251
- Bugfix: *–raw* argument is ignored when searching for a specific reference. #6241

- Bugfix: Avoid raising a version conflict error when aliases have not been resolved yet, typically for aliased `build-requires` that are also in the `requires`. #6236
- Bugfix: **`conan inspect`** now is able to properly show name and version coming from `set_name()` and `set_version()` methods. #6214

22.68 1.21.3 (03-Mar-2020)

- Bugfix: Fixing locking system for metadata file so it can be accessed concurrently. #6543
- Bugfix: Manage the dirty state of the cache package folder with conan export-pkg. #6517
- Bugfix: BugFix: Add quotes to virtualenv scripts, so they don't crash in pure sh shells. #6516
- Bugfix: Upload was silently skipping exceptions, which could result in packages not uploaded, but user not realizing about the error. #6515
- BugFix: Add `system_libs` to `premake` generator. #6496

22.69 1.21.2 (31-Jan-2020)

- Fix: Recipe substitution for scm (old behavior) fixed for multiline comments in Python 3.8 #6439
- Bugfix: Fix broken compression of .tgz files due to Python 3.8 changing tar default schema. #6439
- Bugfix: Append *CONAN_LIBS* in *cmake* generator to avoid overwriting user-defined libs. #6433

22.70 1.21.1 (14-Jan-2020)

- Fix: Fix options type detection using *six.string_types*. #6322
- Fix: Fix minor issues in *cmake* and *cmake_multi* generators: wrong variable used in *conan_find_apple_frameworks* macro. #6295
- Fix: Generators *cmake* and *cmake_multi* use the name of the package instead of *cpp_info.name* (this change is to be reverted in 1.22) #6288
- Bugfix: Fixing readout of backslashes for virtualenv generator files so they are not interpreted as escape characters. #6320
- Bugfix: Fix uninformative crash when `tools.download()` gets a 403 and it is not providing an `auth` field. #6317
- Bugfix: Enhance validation of the *short_paths_home* property to correctly handle the scenarios where it is set to a path that contains the value of the Conan cache path, but is not a subdirectory of it. #6304
- Bugfix: Fixes *cpp_info.name* vs. *cpp_info.names* issue in *pkg_config* generator #6223

22.71 1.21.0 (10-Dec-2019)

- Feature: The generator *cmake_find_package_multi* generates a *PackageConfigVersion.cmake* file that allows using *find_package* with the *VERSION* argument. #6063 . Docs here
- Feature: Settings support for Intel compiler. #6052 . Docs here
- Feature: Allow setting different cpp_info name for each generator that supports that property using the new cpp_info.names["generator_name"] property. #6033 . Docs here
- Feature: Provide *_INCLUDE_DIR* variables in the *cmake_find_package* generator #6017
- Feature: Information in the *artifacts.properties* file is sent using matrix-params too when a package is uploaded to a server (if it has the capability). This will be the recommended way to send these properties to Artifactory (release TBD) to bypass Nginx blocking properties with periods. #6014 . Docs here
- Feature: New *tools.check_min_cppstd* and *tools.valid_min_cppstd* to check if the cppstd version is valid for a specific package. #5997 . Docs here
- Feature: New parameter for *tools.patch* to opt-in applying fuzzy patches. #5996 . Docs here
- Feature: Environment variables for virtual environments are stored in *.env* files containing just the key-value pairs. It will help other processes that need to read these variables to run their own commands. #5989
- Feature: New argument of **`conan upload`** command *–parallel* to upload packages using multithreading. #5856 . Docs here
- Feature: New `python_requires` declared as Conanfile class attributes. Includes extension of base class, they affect the binary packageID with `minor_mode` default mode. They are also locked in lockfiles. #5804 . Docs here
- Feature: Accept logging level as logging names #5772 . Docs here
- Fix: Add the RES_DIRS as variable to the variables when using the `cmake_find_package` generator. #6166
- Fix: Fix SyntaxWarning when comparing a literal with for identity in Python 3.8 #6165
- Fix: Remove recipe linter from codebase, it is no longer a built-in feature. It has been moved to hooks. Install the hook and update your "conan.conf" to activate it. #6152 . Docs here
- Fix: Make lockfiles invariant when the graph doesn't change. Now 2 different lockfiles captured with the same resulting graph in 2 different instants will be identical. #6139
- Fix: Make the `compatible_packages` feature to follow the `--build=missing` build policy. Packages that find a compatible binary will not fire a binary build with the "missing" build policy. #6134 . Docs here
- Fix: Fix create command build policy help message to reflect correct behavior. #6131 . Docs here
- Fix: Improved error message when sources can't be retrieved from remote #6085
- Fix: Raise a meaningful error when the *settings.yml* file is invalid #6059
- Fix: Move the warning about mixing 'os' and 'os_build' to just before the pre_export stage #6021
- Bugfix: Implement `SystemPackageTool.installed(package_name)` as described in the documentation. #6198
- Bugfix: Remove carriage returns from build info *.json* file to avoid Artifactory errors in some cases when publishing the build info to the remote. #6180
- Bugfix: Upload correct packages when specifying revisions and fail with incorrect ones. #6143
- Bugfix: Fix different problems when using **`conan download`** with revisions. #6138

- Bugfix: Make sure `set_version()` runs in the `conanfile.py` folder, not in the current folder, so relative paths are not broken if executing from a different location. #6130 . Docs here
- Bugfix: Fix the help message for **`conan export-pkg`** command for the –options parameter. #6092
- Bugfix: Use a context manager to change the folder during *build_package* to avoid propagating the directory change to other tasks. #6060
- Bugfix: The *AutoToolsBuildEnvironment* build helper now uses the *win_bash* parameter of the constructor when calling to *configure()*. #6026
- Bugfix: Conan's virtualenvironments restore the environment to the state it was before activating them (previously it was restored to the state it was when the **`conan install`** was run). #5989

22.72 1.20.5 (3-Dec-2019)

- Bugfix: Removing *–skip-env* and *–multi-module* arguments for *conan_build_info –v2*. Now the environment is not captured (will be handled by the Artifactory plugin) and recipes and packages are saved as different modules in build info. #6169 . Docs here

22.73 1.20.4 (19-Nov-2019)

- Feature: Added traces to *check_output* internal call to log the called command and the output as INFO traces (can be adjusted with *export CONAN_LOGGING_LEVEL=20*) #6091
- Bugfix: Using *scm* with *auto* values with a *conanfile.py* not being in the root scm folder it failed to export the right source code directory if not using *–ignore-dirty* and the repo was not pristine. #6098
- Bugfix: Fix *conan_build_info* command when conan_sources.tgz not present in remote. #6088

22.74 1.20.3 (11-Nov-2019)

- Bugfix: Using the *scm* feature with *auto* fields was not using correctly the freeze sources from the local user directory from the second call to **`conan create`**. #6048
- Bugfix: Each Apple framework found using CMake *find_library* is stored in a different *CONAN_FRAMEWORK_<name>_FOUND* variable #6042

22.75 1.20.2 (6-Nov-2019)

- Bugfix: Fix Six package version to be compatible with Astroid #6031

22.76 1.20.1 (5-Nov-2019)

- Bugfix: Fixed authentication with an Artifactory repository without anonymous access enabled. #6022

22.77 1.20.0 (4-Nov-2019)

- Feature: Provide *CONAN_FRAMEWORKS* and *CONAN_FRAMEWORKS_FOUND* for Apple frameworks in CMake generators and *conan_find_apple_frameworks()* macro helper in CMake generators. #6003 . Docs here
- Feature: Saving profile list as a json file #5954 . Docs here
- Feature: Improve *conan_build_info* command maintaining old functionality. #5950 . Docs here
- Feature: Add *–json `argument to the `config home* subcommand to output the result to a JSON file. #5946 . Docs here
- Feature: Add *cpp_info.build_modules* to manage build system modules like additional CMake functions in packages #5940 . Docs here
- Feature: Add support for Clang 10. #5936
- Feature: Store *md5* and *sha1* checksums of downloaded and uploaded packages in *metadata.json*. #5910
- Feature: Allow the possibility to avoid *x86_64* to *x86* building when cross-building. #5904 . Docs here
- Feature: Allow to specify encoding for *tools.load*, *tools.save* and *tools.replace_in_files*. #5902 . Docs here
- Feature: Add support for gcc 7.4. #5898 . Docs here
- Feature: New `set_name()` and `set_version()` member methods to dynamically obtain the name and version (at export time). #5881 . Docs here
- Feature: New binary compatibility mode. Recipes can define in their `package_id()` an ordered list of binary package variants that would be binary compatible with the default one. These variants will be checked in order if the main package ID is not found (missing), and the first one will be installed and used. #5837 . Docs here
- Feature: Support for DNF system package manager (Fedora 31+ and others) when present. #5791 . Docs here
- Feature: Refactor Conan Upload, Download and Compress progress bars. #5763
- Feature: Add *system_deps* attribute for cpp_info and deps_cpp_info. #5582 . Docs here
- Feature: The *scm* feature does not replace the *scm.revision="auto"* field with the commit when uncommitted changes unless `--scm-dirty` argument is specified. The recipe in the local cache will be kept with *revision=auto*. #5543 . Docs here
- Feature: The **`conan upload`** command forbids to upload a recipe that uses the *scm* feature containing *revision=auto* or *url=auto*, unless *–force* is used. #5543 . Docs here
- Feature: The *scm* feature captures the local sources in the local cache during the export, avoiding later issues of modified local sources. #5543 . Docs here
- Fix: Deprecate argument *–build-order* in **`conan info`** command. #5965 . Docs here
- Fix: Avoid doing complex `conan search --query` in the server, do them always in the client. #5960
- Fix: Improved `conan remove --help` message for `--packages` #5899
- Fix: Improved cmake compiler check message to explain the problem with different compiler versions when installing dependencies #5858
- Fix: Adds support for transitive dependencies to b2 generator. #5812

- Fix: Add support for recipes without *settings.compiler* in b2 generator. #5810
- Fix: Add and remove out-of-tree git patches (#5320) #5761
- Fix: Add quiet output for *inspect –raw*. #5702
- Bugfix: Allow **`conan download`** for packages without user/channel #6010
- Bugfix: Avoid erroneous case-sensitive conflict for packages without user/channel. #5981
- Bugfix: Fix crashing when using lockfiles with a `conanfile.txt` instead of `conanfile.py`. #5894
- Bugfix: Fix incorrect propagation of build-requires to downstream consumers, resulting in missing dependencies in `deps_cpp_info`. #5886
- Bugfix: Adds the *short_paths_home* property to *ConanClientConfigParser* to validate that it is not a subdirectory of the conan cache. #5864 . Docs here
- Bugfix: Use imported python requires' *short_path* value instead of the defined in the *conanfile* that imports it. #5841
- Bugfix: Avoid repeated copies of absolute paths when using *self.copy()*. #5792
- Bugfix: Downstream overrides to exact dependencies versions are always used, even if the upstream has a version range that does not satisfy the override. #5713

22.78 1.19.3 (29-Oct-2019)

- Fix: Fixed range of pylint and astroid requirements to keep compatibility with python 2 #5987
- Fix: Force `conan search --query` queries to be resolved always in the client to avoid servers failures due to unsupported syntax #5970
- Bugfix: Use cpp_info.name lower case in pkg-config generator when defined #5988
- Bugfix: Fix `cpp_info.name` not used in cmake find generators for dependencies #5973
- Bugfix: Fixed bug when overriden dependencies that don't exist and make the CMake generated code crash #5971
- Bugfix: Fixed bug when overriden dependencies that don't exist and make the CMake generated code crash #5945

22.79 1.19.2 (16-Oct-2019)

- Feature: Implement `self.info.shared_library_package_id()` to better manage shared libraries package-ID, specially when they depend on static libraries #5893 . Docs here
- Bugfix: Allow `conan install pkg/[*]@user/channel` resolving to a reference, not a path. #5908
- Bugfix: The dependency overriding mechanism was not working properly when using the same version with different build metadata (*1.2.0+xyz* vs *1.2.0+abc*). #5903
- Bugfix: Artifactory was returning an error on the first login attempt because the server capabilities were not assigned correctly. #5880
- Bugfix: conan export failed if there is no user/channel and a lockfile is applied #5875
- Bugfix: SCM component failed for url pointing to local path in Windows with backslash. #5875
- Bugfix: Fix *conan graph build-order* output so it uses references including its recipe revision #5863

22.80 1.19.1 (3-Oct-2019)

- Bugfix: Use imported python requires' *short_path* value instead of the defined in the *conanfile* that imports it. #5849
- Bugfix: Fix regression in `visual_studio` generator adding a <Lib> task. #5846 . Docs here

22.81 1.19.0 (30-Sept-2019)

- Feature: Update settings.yml file with macOS, watchOS, tvOS, iOS version numbers #5823
- Feature: Add clang 9 to the settings.yml file #5786 . Docs here
- Feature: Show suggestions when typing an incorrect command conan command. #5725
- Feature: Client support for using refresh tokens in the auth process with Artifactory. #5662
- Feature: Add GCC 9.2 to default settings.yml file #5650 . Docs here
- Feature: Add subcommand for enabling and disabling remotes #5623 . Docs here
- Feature: New *conan config home* command for getting Conan home directory #5613 . Docs here
- Feature: Adds *name* attribute to *CppInfo* and use *cpp_info.name* in all CMake and pkg-config generators as the find scripts files names, target names, etc. #5598 . Docs here
- Feature: Enhanced vs-generator by providing more properties that can be referenced by other projects; added library paths also to <Lib> so it's possible to compile static libraries that reference other libs #5564
- Feature: Better support OSX frameworks by declaring *cppinfo.frameworks*. #5552 . Docs here
- Feature: Virtual environment generator for gathering only the PYTHONPATH. #5511 . Docs here
- Fix: **`conan upload`** with a reference without user and channel and package id `name/version:package_id` should work #5824
- Fix: Dropped support for python 3.4. That version is widely being dropped by the python community. Since Conan 1.19, the tests won't be run with python 3.4 and we won't be aware if something is not working correctly. #5820 . Docs here
- Fix: Apply lockfile to the node before updating with downstream requirements #5771
- Fix: Make **`conan new`** generate default options as a dictionary #5767
- Fix: Output search result for remotes in order by version, as local search #5723
- Fix: Excluded also *ftp_proxy* and *all_proxy* variables from the environment when proxy configuration is specified in the *conan.conf* file. #5697
- Fix: Relax restriction on the future python dependency #5692
- Fix: Call *post_package* hook before computing the manifest #5647
- Fix: Show friendly message when can't get remote path #5638
- Fix: Detect the number of CPUs used by Docker (#5464) #5466 . Docs here
- Bugfix: Set Ninja to use *cpu_count* value when building with *parallel* option with CMake #5832
- Bugfix: output of references without user/channel is done with _/_, like in lockfiles. #5817
- Bugfix: A lockfile generated from a consumer should be able to generate a build-order too. #5800
- Bugfix: Fix system detection on Solaris. #5630

- Bugfix: *SVN* uses *username* and *password* if provided #5601
- Bugfix: Use the final package folder as the *conanfile.package_folder* attribute for the *pre_package* hook. #5600
- BugFix: Fix crash with custom generators using `install_folder` #5569

22.82 1.18.5 (24-Sept-2019)

- Bugfix: A bug in *urllib3* caused bad encoded URLs causing failures when using any repository from Bintray, like *conan-center*. #5801

22.83 1.18.4 (12-Sept-2019)

- Fix: `package_id` should be used for `recipe_revision_mode` #5729 . Docs here

22.84 1.18.3 (10-Sept-2019)

- Fix: Version ranges resolution using references without user/channel #5707

22.85 1.18.2 (30-Aug-2019)

- Feature: Add opt-out for Git shallow clone in *SCM* feature #5677 . Docs here
- Fix: Use the value of argument *useEnv* provided by the user to the *MSBuild* helper also to adjust */p:UseEnv=false* when the arg is *False*. #5609
- Bugfix: Fixed assertion when using nested build_requires that depend on packages that are also used in the main dependency graph #5689
- Bugfix: When Artifactory doesn't have the anonymous access activated, the conan client wasn't able to capture the server capabilities and therefore never used the *revisions* mechanism. #5688
- Bugfix: When no *user/channel* is specified creating a package, upload it to a remote using *None* as the "folder" in the storage, instead of _. #5671
- Bugfix: Using the version ranges mechanism Conan wasn't able to resolve the correct reference if a library with the same name but different user/channel was found in an earlier remote. #5657
- Bugfix: Broken cache package collection for packages without user/channel #5607

22.86 1.18.1 (8-Aug-2019)

- Bugfix: The *scm* feature was trying to run a checkout after a shallow clone. #5571

22.87 1.18.0 (30-Jul-2019)

- Feature: The "user/channel" fields are now optional. e.g: *conan create .* is valid if the *name* and *version* are declared in the recipe. e.g: *conan create . lib/1.0@* to omit user and channel. The same for other commands. The *user* and *channel* can also be omitted while specifying requirements in the conanfiles. #5381 . Docs here
- Feature: Output current revision from references in local cache when using a pattern #5537 . Docs here
- Feature: New parameter `--skip-auth` for the **`conan user`** command to avoid trying to authenticate when the client already has credentials stored. #5532 . Docs here
- Feature: Allow patterns in per-package settings definitions, not only the package name #5523 . Docs here
- Feature: Search custom settings (#5378) #5521 . Docs here
- Feature: shallow git clone #5514 . Docs here
- Fix: Remove `conan graph clean-modified` command, it is automatic and no longer necessary. #5533 . Docs here
- Fix: Incomplete references (for local conanfile.py files) are not printed with *@None/None* anymore. #5509
- Fix: Discard empty string values in SCM including *subfolder* #5459
- Bugfix: The *stderr* was not printed when a command failed running the *tools.check_output* function. #5548
- Bugfix: Avoid dependency (mainly build-requires) being marked as skipped when another node exists in the graph that is being skipped because of being private #5547
- Bugfix: fix processing of UTF-8 files with BOM #5506
- Bugfix: apply http.sslVerify to the current Git command only #5470
- Bugfix: Do not raise when accessing the metadata of editable packages #5461
- Bugfix: Use cxxFlags instead of cppFlags in `qbs` generator. #5452 . Docs here

22.88 1.17.2 (25-Jul-2019)

- Bugfix: Lock transitive python-requires in lockfiles, not only direct ones. #5531

22.89 1.17.1 (22-Jul-2019)

- Feature: support 7.1 clang version #5492
- Bugfix: When a profile was detected, for GCC 5.X the warning message about the default *libcxx* was not shown. #5524
- Bugfix: Update python-dateutil dependency to ensure availability of *dateutil.parser.isoparse* #5485
- Bugfix: Solve regression in `conan info <ref>` command, incorrectly reading the graph_info.json and lockfiles #5481
- Bugfix: Trailing files left when packages are not found in conan info and install, restricted further installs with different case in Windows, without `rm -rf ~/.conan/data/pkg_name` #5480
- Bugfix: The lock files mechanism now allows to update a node providing new information, like a retrieved package revision, if the "base" reference was the same. #5467
- Bugfix: search command table output has invalid HTML code syntax #5460

22.90 1.17.0 (9-Jul-2019)

- Feature: Better UX for no_proxy (#3943) #5438 . Docs here
- Feature: Show warning when URLs for remotes is invalid (missing schema, host, etc). #5418
- Feature: Implementation of lockfiles. Lockfiles store in a file all the configuration, exact versions (including revisions), necessary to achieve reproducible builds, even when using version-ranges or package revisions. #5412 . Docs here
- Feature: Change progress bar output to tqdm to make it look better #5407
- Feature: Define 2 new modes and helpers for the package binary ID: `recipe_revision_mode` and `package_revision_mode`, that take into account the revisions. The second one will use all the information from dependencies, resulting in fully deterministic and complete package IDs: if some dependency change, it will be necessary to build a new binary of consumers #5363 . Docs here
- Feature: Add apple-clang 11.0 to settings.yml (#5328) #5357 . Docs here
- Feature: SystemPackageTool platform detection (#5026) #5215 . Docs here
- Fix: Enable the definition of revisions in conanfile.txt #5435
- Fix: Improve resolution of version ranges for remotes #5433
- Fix: The conan process returns *6* when a *ConanInvalidConfiguration* is thrown during **`conan info`**. #5421
- Fix: Inspect missing attribute is not an error (#3953) #5419
- Fix: Allow –build-order and –graph together for conan info (#3447) #5417
- Fix: Handling error when reference not found using conan download #5399
- Fix: Update Yum cache (#5370) #5387
- Fix: Remove old folder for conan install (#5376) #5384
- Fix: Add missing call to super constructor to *VirtualEnvGenerator*. #5375
- Fix: Force forward slashes in the variable *$PROFILE_DIR* #5373 . Docs here
- Fix: Accept a list for the requires attribute #5371 . Docs here
- Fix: Remove packages when version is asterisk (#5297) #5346
- Fix: Make conan_data visible to pylint (#5327) #5337
- Fix: Improve the output to show the remote (or cache) that a version range is resolved to. #5336
- Fix: Deprecated `conan copy|download|upload <ref> -p=ID`, use `conan .... <pref>` instead #5293 . Docs here
- Fix: *AutoToolsBuildEnvironment* is now aware of *os_target* and *arch_target* to calculate the gnu triplet when declared. #5283
- Fix: Better message for gcc warning of libstdc++ at default profile detection #5275
- Bugfix: *verify_ssl* field in SCM being discarded when used with *False* value. #5441
- Bugfix: enable retry for requests #5400
- Bugfix: Allow creation and deletion of files in `tools.patch` with `strip>0` #5334
- Bugfix: Use case insensitive comparison for SHA256 checksums #5306

22.91 1.16.1 (14-Jun-2019)

- Feature: Print nicer error messages when receive an error from Artifactory. #5326
- Fix: Make `conan config get storage.path` return an absolute, resolved path #5350
- Fix: Skipped the compiler version check in the cmake generator when a *-s compiler.toolset* is specified (Visual Studio). #5348
- Fix: Constraint transitive dependency `typed-ast` (required by astroid) in python3.4, as they stopped releasing wheels, and it fails to build in some Windows platforms with older SDKs. #5324
- Fix: Accept v140 and VS 15.0 for CMake generator (#5318) #5321
- Fix: Accept only .lib and .dll as Visual extensions (#5316) #5319
- Bugfix: Do not copy directories inside a symlinked one #5342
- Bugfix: Conan was retrying the upload when failed with error 400 (request error). #5326

22.92 1.16.0 (4-Jun-2019)

- Feature: The **`conan upload`** command can receive now the full package reference to upload a binary package. The *-p* argument is now deprecated. #5224 . Docs here
- Feature: Add hooks *pre_package_info* and *post_package_info* #5223 . Docs here
- Feature: New build mode *–build cascade* that forces building from sources any node with dependencies also built from sources. #5218 . Docs here
- Feature: Print errors and warnings to *stderr* #5206
- Feature: New `conan new --template=mytemplate` to initialize recipes with your own templates #5189 . Docs here
- Feature: Allow using wildcards to remove system requirements sentinel from cache. #5176 . Docs here
- Feature: Implement conan.conf `retry` and `retry-wait` and `CONAN_RETRY` and `CONAN_RETRY_WAIT` to configure all retries for all transfers, including upload, download, and `tools.download()`. #5174 . Docs here
- Feature: Support yaml lists in workspace `root` field. #5156 . Docs here
- Feature: Add gcc 8.3 and 9.1 new versions to default *settings.yml* #5112
- Feature: Retry upload or download for error in response message (e.g. status is '500') #4984
- Fix: Do not retry file transfer operations for 401 and 403 auth and permissions errors. #5278
- Fix: Copy symlinked folder when using *merge_directories* function #5237
- Fix: Add the ability to avoid the */verbosity* argument in CMake command line for MSBuild #5220 . Docs here
- Fix: self.copy with symlinks=True does not copy symlink if the .conan directory is a symlink #5114 #5125
- Fix: Export detected_os from tools.oss (#5101) #5102 . Docs here
- Fix: Use *revision* as the SVN's *peg_revision* (broken for an edge case) #5029
- Bugfix: `--update` was not updating `python_requires` using version ranges. #5265
- Bugfix: `visual_studio` generator only adds ".lib" extension for lib names without extension, otherwise (like ".a") respect it. #5254
- Bugfix: Fix **`conan search`** command showing revisions timestamps in a different time offset than UTC. #5232

- Bugfix: Meson build-helper gets correct compiler flags, AutoTools build environment adds compiler.runtime flags #5222
- Bugfix: The *cmake_multi* generator was not managing correctly the *RelWithDebInfo* and *MinSizeRel* build types. #5221
- Bugfix: Check that registry file exists before removing it #5219
- Bugfix: do not append "-T " if generator doesn't support it #5201
- Bugfix: **`conan download`** always retrieve the sources, also with `--recipe` argument, which should only skip download binaries, not the sources. #5194
- Bugfix: Using *scm* declared in a superclass failed exporting the recipe with the error *ERROR: The conanfile.py defines more than one class level 'scm' attribute*. #5185
- Bugfix: Conan command returns 6 (Invalid configuration) also when the settings are restricted in the recipe #5178
- Bugfix: Make sure that proxy "http_proxy", "https_proxy", "no_proxy" vars are correctly removed if custom ones are defined in the conan.conf. Also, avoid using `urllib.request.getproxies()`, they are broken. #5162
- Bugfix: Use *copy()* for deploy generator so that permissions of files are preserved. Required if you want to use the deploy generator to deploy executables. #5136

22.93 1.15.4

- Fix: Accept v140 and VS 15.0 for CMake generator (#5318) #5331
- Fix: Constraint transitive dependency typed-ast (required by astroid) in python3.4, as they stopped releasing wheels, and it fails to build in some Windows platforms with older SDKs. #5331

22.94 1.15.3

- Please, do not use this version, there was a critical error in the release process and changes from the 1.16 branch were merged.

22.95 1.15.2 (31-May-2019)

- Bugfix: Fix bug with python-requires not being updated with `--update` if using version-ranges. #5266
- Bugfix: Fix computation of ancestors performance regression #5260

22.96 1.15.1 (16-May-2019)

- Fix: Fix regression of `conan remote update --insert` using the same URL it had before #5110
- Fix: Fix migration of *registry.json|txt* file including reference to non existing remotes. #5103
- Bugfix: Avoid crash of commands copy, imports, editable-add for packages using python_requires #5150

22.97 1.15.0 (6-May-2019)

- Feature: Updated the generated *conanfile.py* in **conan new** to the new conan-io/hello repository #5069 . Docs here
- Feature: The *MSBuild* build helper allows the parameter *toolset* with *False* value to skip the toolset adjustment. #5052 . Docs here
- Feature: Add GCC 9 to default settings.yml #5046 . Docs here
- Feature: You can disable broken symlinks checks when packaging using *CONAN_SKIP_BROKEN_SYMLINKS_CHECK* env var or *config.skip_broken_symlinks_check=1* #4991 . Docs here
- Feature: New `deploy` generator to export files from a dependency graph to an installation folder #4972 . Docs here
- Feature: Create *tools.Version* with _limited_ capabilities #4963 . Docs here
- Feature: Default filename for workspaces: *conanws.yml* (used in install command) #4941 . Docs here
- Feature: Add install folder to command 'conan workspace install' #4940 . Docs here
- Feature: Add *compiler.cppstd* setting (mark *cppstd* as deprecated) #4917 . Docs here
- Feature: Add a *–raw* argument to **conan inspect** command to get an output only with the value of the requested attributes #4903 . Docs here
- Feature: `tools.get()` and `tools.unzip()` now handle also `.gz` compressed files #4883 . Docs here
- Feature: Add argument *–force* to command *profile new* to overwrite existing one #4880 . Docs here
- Feature: Get commit message #4877 . Docs here
- Fix: Remove sudo from Travis CI template #5073 . Docs here
- Fix: Handle quoted path and libraries in the premake generator #5051
- Fix: A simple addition to ensure right compiler version is found on windows. #5041
- Fix: Include CMAKE_MODULE_PATH for CMake find_dependency (#4956) #5021
- Fix: Add default_package_id_mode in the default conan.conf (#4947) #5005 . Docs here
- Fix: Use back slashes for `visual_studio` generator instead of forward slashes #5003
- Fix: Adding *subparsers.required = True* makes both Py2 and Py3 print an error when no arguments are entered in commands that have subarguments #4902
- Fix: Example bare package recipe excludes *conanfile.py* from copy #4892
- Fix: More meaningful error message when a remote communication fails to try to download a binary package. #4888
- Bugfix: `conan upload --force` force also the upload of package binaries, not only recipes #5088
- BugFix: MSYS 3.x detection #5078
- Bugfix: Don't crash when an editable declare a `build_folder` in the layout, but not used in a workspace #5070
- Bugfix: Made compatible the *cmake_find_package_multi* generator with *CMake < 3.9* #5042
- Bugfix: Fix broken local development flow (**conan source**, **conan build**, **conan package**, **conan export-pkg**) with recipes with `python-requires` #4979
- Bugfix: 'tar_extract' function was failing if there was a linked folder in the working dir that matches one inside the tar file. Now we use the *destination_dir* as base directory to check this condition. #4965

- Bugfix: Remove package folder in `conan create` even when using `--keep-build` #4918

22.98 1.14.5 (30-Apr-2019)

- Bugfix: Uncompressing a *tgz* package with a broken symlink failed while touching the destination file. #5065
- Bugfix: The symlinks compressed in a *tgz* had invalid nonzero size. #5064
- Bugfix: Fixing exception of transitive build-requires mixed with normal requires #5056

22.99 1.14.4 (25-Apr-2019)

- Bugfix: Fixed error while using Visual Studio 2019 with Ninja generator. #5028
- Bugfix: Fixed error while using Visual Studio 2019 with Ninja generator. #5025
- Bugfix: Solved errors in concurrent uploads of same recipe #5014
- Bugfix: Fixed a bug that intermittently raised *ERROR: 'NoneType' object has no attribute 'file_sums'* when uploading a recipe. #5012
- Bugfix: Bug in *cmake_find_package_multi* caused *CMake* to find incorrect modules in *CMake* modules paths when only *Config* files should be taken into account. #4995
- Bugfix: Fix skipping binaries because of transitive `private` requirements #4987
- Bugfix: Fix broken local development flow (conan source, conan build, conan package, conan export-pkg) with recipes with python-requires #4983

22.100 1.14.3 (11-Apr-2019)

- Bugfix: `build-requires` and `private` requirements that resolve to a dependency that is already in the graph won't span a new node, nor will be `build-requires` or `private`. They can conflict too. #4937

22.101 1.14.2 (11-Apr-2019)

- Bugfix: Run a full metadata migration in the cache to avoid old `null` revisions in package metadata #4934

22.102 1.14.1 (1-Apr-2019)

- Fix: Print a message for unhandled Conan errors building the API and collaborators #4869
- Bugfix: Client does not require credentials for anonymous downloads from remotes. #4872
- Bugfix: Fix a migration problem of `conan config install` for Conan versions 1.9 and older #4870
- Feature: Now Conan will crush your enemies, see them driven before you, and to hear the lamentation of their women! (April's fools)

22.103 1.14.0 (28-Mar-2019)

- Feature: support new architectures s390 and s390x #4810 . Docs here
- Feature: *–build* parameter now applies fnmatching onto the whole reference, allowing to control rebuilding in a much broader way. #4787 . Docs here
- Feature: Add config variable *general.error_on_override* and environment variable *CONAN_ERROR_ON_OVERRIDE* (defaulting to *False*) to configure if an overridden requirement should raise an error when overridden from downstream consumers. #4771 . Docs here
- Feature: Allow to specify *revision_mode* for each recipe, values accepted are *scm* or *hash* (default) #4767 . Docs here
- Feature: Sort library list name when calling tools.collect_libs #4761 . Docs here
- Feature: Add *cmake_find_package_multi* generator. #4714 . Docs here
- Feature: Implement `--source-folder` and `--target-folder` to `conan config install` command to select subfolder to install from the source origin, and also the destination folder within the cache. #4709 . Docs here
- Feature: Implement `--update` argument for `python-requires` too. #4660
- Fix: Apply environment variables from profile and from requirements to **`conan export-pkg`** #4852
- Fix: Do not run *export_sources* automatically for python_requires #4838
- Fix: Show the correct profile name when detect a new one (#4818) #4824
- Fix: Allow using `reference` object in workspaces in templates for out of source builds #4812 . Docs here
- Fix: Look for `vswhere` in `PATH` when using `tools.vswhere()` #4805
- Fix: SystemPackageTools doesn't run sudo when it's not found (#4470) #4774 . Docs here
- Fix: Show warning if repo is not pristine and using SCM mode to set the revisions #4764
- Fix: avoid double call to `package()` method #4748 . Docs here
- Fix: The *cmake_paths* generator now declares the *CONAN_XXX_ROOT* variables in case some exported cmake module file like *XXXConfig.cmake* has been patched with the *cmake.patch_config_paths()* to replace absolute paths to the local cache. #4719 . Docs here
- Fix: Do not distribute the tests in the python package nor in the installers. #4713
- Fix: add support for CMake generator platform #4708 . Docs here
- Fix: Fix corrupted packages with missing conanmanifest.txt files #4662
- Fix: Include information about all the configurations in the JSON generator #4657 . Docs here
- Bugfix: Fixed authentication management when a server returns 401 uploading a file. #4857
- Bugfix: Fixed recipe revision detection when some error output or unexpected output was printed to the stdout running the *git* command. #4854
- Bugfix: The error output was piped to stdout causing issues while running git commands, especially during the detection of the scm revision #4853
- Bugfix: **`conan export-pkg`** should never resolve build-requires #4851
- bugfix: The *–build* pattern was case sensitive depending on the os file system, now it is always case sensitive, following the **`conan search`** behavior. #4842
- Bugfix: Fix metadata not being updated for **`conan export-pkg`** when using `--package-folder` #4834

- Bugfix: *–build* parameter now is always case-sensitive, previously it depended to the file system type. #4787 . Docs here
- Bugfix: Raise an error if source files cannot be correctly copied to build folder because of long paths in Windows. #4766
- Bugfix: Use the same interface in `conan_basic_setup()` for the `cmake_multi` generator #4721 . Docs here

22.104 1.13.3 (27-Mar-2019)

- Bugfix: Revision computation failed when a git repo was present but without commits #4830

22.105 1.13.2 (21-Mar-2019)

- Bugfix: Installing a reference with "update" and "build outdated" options raised an exception. #4790
- Bugfix: Solved bug with build-requires transitive build-requires #4783
- Bugfix: Fixed workspace crash when no layout was specified #4783
- Bugfix: Do not generate multiple `add_subdirectories()` for workspaces build-requires #4783

22.106 1.13.1 (15-Mar-2019)

- Bugfix: Fix computation of graph when transitive diamonds are processed. #4737

22.107 1.13.0 (07-Mar-2019)

- Feature: Added with_login parameter to tools.run_in_windows_bash() #4673 . Docs here
- Feature: The *deb* and *windows* Conan installers now use Python 3. #4663
- Feature: Allow configuring in *conan.conf* a different default `package_id` mode. #4644 . Docs here
- Feature: Apply Jinja2 to layout files before parsing them #4596 . Docs here
- Feature: Accept a PackageReference for the command **`conan get`** (argument *-p* is accepted, but hidden) #4494 . Docs here
- Feature: Re-implement Workspaces based on Editable packages. #4481 . Docs here
- Feature: Removed old "compatibility" mode of revisions. #4462 . Docs here
- Fix: When revisions enabled, add the revision to the json output of the info/install commands. #4667
- Fix: JSON output for *multi_config* now works in *install* and *create* commands #4656
- Fix: Deprecate 'cppflags' in favor of 'cxxflags' in class CppInfo #4611 . Docs here
- Fix: Return empty list if env variable is an empty string #4594
- Fix: *conan profile list* will now recursively list profiles. #4591
- Fix: *Instance of 'TestConan' has no 'install_folder' member* when exporting recipe #4585
- Fix: SCM replacement with comments below it #4580

- Fix: Remove package references associated to a remote in *registry.json* when that remote is deleted #4568
- Fix: Fixed issue with Artifactory when the anonymous user is enabled, causing the uploads to fail without requesting the user and password. #4526
- Fix: Do not allow an alias to override an existing package #4495
- Fix: Do not display the warning when there are files in the package folder (#4438). #4464
- Fix: Renamed the **`conan link`** command to **`conan editable`** to put packages into editable mode. #4481 . Docs here
- Bugfix: Solve problem with loading recipe python files in Python 3.7 because of `module.__file__ = None` #4669
- Bugfix: Do not attempt to upload non-existing packages, due to empty short_paths folders, or to explicit `upload -p=id` command. #4615
- Bugfix: Fix LIB overwrite in `virtualbuildenv` generator #4583
- Bugfix: Avoid `str(self.settings.xxx)` crash when the value is None. #4571 . Docs here
- Bugfix: Build-requires expand over the closure of the package they apply to, so they can create conflicts too. Previously, those conflicts were silently skipped, and builds would use an undetermined version and configuration of dependencies. #4514
- Bugfix: meson build type actually reflects recipe shared option #4489
- Bugfix: Fixed several bugs related to revisions. #4462 . Docs here
- Bugfix: Fixed several bugs related to the package *metadata.json* #4462 . Docs here

22.108 1.12.3 (18-Feb-2019)

- Fix: Fix potential downgrade from future 1.13 to 1.12 #4547
- Fix: Remove output warnings in MSBuild helper. #4518
- Fix: Revert default cmake generator on Windows (#4265) #4509 . Docs here
- Bugfix: Fixed problem with conanfile.txt [imports] sections using the '@' character. #4539 . Docs here
- Bugfix: Fix search packages function when remote is called *all* #4502

22.109 1.12.2 (8-Feb-2019)

- Bugfix: Regression in `MSBuild` helper, incorrectly ignoring the `conan_build.props` file because of using a relative path instead of absolute one. #4488

22.110 1.12.1 (5-Feb-2019)

- Bugfix: GraphInfo parsing of existing `graph_info.json` files raises KeyError over "root". #4458
- Bugfix: Transitive Editable packages fail to install #4448

22.111 1.12.0 (30-Jan-2019)

- Feature: Add JSON output to 'info' command #4359 . Docs here
- Feature: Remove system requirements conan folders (not installed binaries) from cache #4354 . Docs here
- Feature: Updated *CONTRIBUTING.md* with code style #4348
- Feature: Updated OS versions for apple products #4345
- Feature: add environment variable CONAN_CACHE_NO_LOCKS to simplify debugging #4309 . Docs here
- Feature: The commands **`conan install`**, **`conan info`**, **`conan create`** and **`conan export-pkg`** now can receive multiple profile arguments. The applied profile will be the composition of them, prioritizing the latest applied. #4308 . Docs here
- Feature: Added `get_tag()` methods to `tools.Git()` and `tools.SVN()` helpers. #4306 . Docs here
- Feature: Package reference is now accepted as an argument in `conan install --build` #4305 . Docs here
- Feature: define environment variables for CTest #4299 . Docs here
- Feature: Added a configuration entry at the *conan.conf* file to be able to specify a custom *CMake* executable. #4298 . Docs here
- Feature: Skip "README.md" and "LICENSE.txt" during the installation of a custom config via *conan config install*. #4259 . Docs here
- Feature: allow to specify MSBuild verbosity level #4251 . Docs here
- Feature: add definitions to MSBuild build helper (and `tools.build_sln_command()`) #4239 . Docs here
- Feature: Generate deterministic short paths on Windows #4238
- Feature: The *tools.environment_append()* now accepts unsetting variables by means of appending such variable with a value equal to None. #4224 . Docs here
- Feature: Enable a new `reference` argument in `conan install <path> <reference>`, where `reference` can be a partial reference too (identical to what is passed to **`conan create`** or **`conan export`**. This allows defining all pkg,version,user,channel fields of the recipe for the local flow. #4197 . Docs here
- Feature: Added support for new architecture `ppc32` #4195 . Docs here
- Feature: Added support for new architecture `armv8.3` #4195 . Docs here
- Feature: Added support for new architecture `armv8_32` #4195 . Docs here
- Feature: Add experimental support for packages in editable mode #4181 . Docs here
- Fix: Conditionally expand list-like environment variables in `virtualenv` generator #4396
- Fix: get_cross_building_settings for MSYS #4390
- Fix: Implemented retrial of output to stdout stream when the OS (Windows) is holding it and producing IOError for output #4375
- Fix: Validate CONAN_CPU_COUNT and output user-friendly message for invalid values #4372

- Fix: Map `cpp_info.cppflags` to `CONAN_CXXFLAGS` in `make` generator. #4349 . Docs here
- Fix: Use `*_DIRS` instead of `*_PATHS` ending for varaibles generated by the `make` generator: `INCLUDE_DIRS`, `LIB_DIRS`, `BIN_DIRS`, `BUILD_DIRS` and `RES_DIRS` #4349 . Docs here
- Fix: Bumped requirement of pyOpenSSL on OSX to *>=16.0.0, <19.0.0* #4333
- Fix: Fixed a bug in the migration of the server storage to the revisions layout. #4325
- Fix: ensure tools.environment_append doesn't raise trying to unset variables #4324 . Docs here
- Fix: Improve error message when a server (like a proxy), returns 200-OK for a conan api call, but with an unexpected message. #4317
- Fix: ensure is_windows, detect_windows_subsystem, uname work under MSYS/Cygwin #4313
- Fix: uname shouldn't use -o flag, which is GNU extention #4311
- Fix: `get_branch()` method of `tools.SVN()` helper now returns only the branch name, not the tag when present. #4306 . Docs here
- Fix: Conan client now always include the *X-Checksum-Sha1* header in the file uploads, not only when checking if the file is already there with a remote supporting checksum deploy (Artifactory) #4303
- Fix: SCM optimization related to *scm_folder.txt* is taken into account only for packages under development. #4301
- Fix: Update premake generator, rename conanbuildinfo.premake -> conanbuildinfo.premake.lua, conan_cppdefines -> conan_defines #4296 . Docs here
- Fix: Using `yaml.safe_load` instead of `load` #4285
- Fix: Fixes default CMake generator on Windows to use MinGW Makefiles. #4281 . Docs here
- Fix: Visual Studio toolset is passed from settings to the MSBuild helper #4250 . Docs here
- Fix: Handle corner cases related to SCM with local sources optimization #4249
- Fix: Allow referring to projects created by b2 generator for dependencies with absolute paths. #4211
- Fix: Credentials are removed from SCM *url* attribute if Conan is automatically resolving it. #4207 . Docs here
- Fix: Remove client/server versions check on every request. Return server capabilities only in *ping* endpoint. #4205
- Fix: Updated contributing guidelines to the new workflow #4173
- Bugfix: Fixes config install when copying hooks #4412
- BugFix: Meson generator was failing in case of package_folder == None (test_package using Meson) #4391
- BugFix: Prepend environment variables are applied twice in conanfile #4380
- Bugfix: Caching of several internal loaders broke the conan_api usage #4362
- Bugfix: Removing usage of FileNotFoundError which is Py3 only #4361
- Bugfix: Custom generator allow to use imports #4358 . Docs here
- Bugfix: conanbuildinfo.cmake won't fail if `project()` LANGUAGE is None, but the user defines `CONAN_DISABLE_CHECK_COMPILER`. #4276
- Bugfix: Fix version ranges containing spaces and not separated by commas. #4273
- Bugfix: When running consecutively Conan python API calls to *create* the default profile object became modified and cached between calls. #4256
- Bugfix: Fixes a bug in the CMake build helper about how flags are appended #4227

- Bugfix: Apply the environment to the local conan package command #4204
- Bugfix: b2 generator was failing when package recipe didn't use compiler setting #4202

22.112 1.11.2 (8-Jan-2019)

- Bugfix: The migrated data in the server from a version previous to Conan *1.10.0* was not migrated creating the needed indexes. This fixes the migration and creates the index on the fly for fixing broken migrations. Also the server doesn't try to migrate while running but warns the user to run *conan server –migrate* after doing a backup of the data, avoiding issues when running the production servers like gunicorn where the process doesn't accept input from the user. #4229

22.113 1.11.1 (20-Dec-2018)

- BugFix: Fix *conan config install* requester for zip file download #4172

22.114 1.11.0 (19-Dec-2018)

- Feature: Store `verify_ssl` argument in **`conan config install`** #4158 . Docs here
- Feature: Tox launcher to run the test suite. #4151
- Feature: Allow `--graph=file.html` html output using local *vis.min.js* and *vis.min.css* resources if they are found in the local cache (can be deployed via **`conan config install`**) #4133 . Docs here
- Feature: Improve client DEBUG traces with better and more complete messages. #4128
- Feature: Server prints the configuration used at startup to help debugging issues. #4128
- Feature: Allow hooks to be stored in folders #4106 . Docs here
- Feature: Remove files containing Macos meta-data (files beginning by ._) #4103 . Docs here
- Feature: Allow arguments in **`git clone`** for **`conan config install`** #4083 . Docs here
- Feature: Display the version-ranges resolutions in a cleaner way. #4065
- Feature: allow `conan export . version@user/channel` and `conan create . version@user/channel` #4062 . Docs here
- Fix: *cmake_find_package* generator not forwarding all dependency properties #4125
- Fix: Recent updates in python break `ConfigParser` with `%` in values, like in path names containing % (jenkins) #4122
- Fix: The property file that the `MSBuild()` is now generated in the *build_folder* instead of a temporary folder to allow more reproducible builds. #4113 . Docs here
- Fix: Fixed the check of the return code from Artifactory when using the checksum deploy feature. #4100
- Fix: Evaluate always SCM attribute before exporting the recipe #4088 . Docs here
- Fix: Reordered Python imports #4064
- Bugfix: In ftp_download function there is extra call to `ftp.login()` with empty args. This causes ftp lib to login again with empty credentials and throwing exception because authentication is required by server. #4092
- Bugfix: Take into account `os_build` and `arch_build` for search queries. #4061

22.115 1.10.2 (17-Dec-2018)

- Bugfix: Fixed bad URL schema in ApiV2 that could cause URLs collisions #4138

22.116 1.10.1 (11-Dec-2018)

- Fix: Handle some corner cases of python_requires #4099
- Bugfix: Add v1_only argument in Conan server class #4096
- Bugfix: Handle invalid use of *python_requires* when imported like *conans.python_requires* #4090

22.117 1.10.0 (4-Dec-2018)

- Feature: Add *include_prerelease* and *loose* option to version range expression #3898
- Feature: Merged "revisions" feature code in develop branch, still disabled by default until it gets stabilized. #3055
- Feature: CMake global variable to disable Conan output `CONAN_CMAKE_SILENT_OUTPUT` #4042
- Feature: Added new `make` generator. #4003
- Feature: Deploy a conan snapshot package to test.pypi.org for every develop commit. #4000
- Fix: Using the *scm* feature when Conan is not able to read the gitignored files (local optimization mechanism) print a warning to improve the debug information but not crash. #4045
- Fix: The *tools.get* tool (download + unzip) now supports all the arguments of the *download* tool. e.g: *verify*, *retry*, *retry_wait* etc. #4041
- Fix: Improve `make` generator test #4018
- Fix: Add space and dot in `conan new --help` #3999
- Fix: Resolve aliased packages in python_requires #3957
- Bugfix: Better checks of package reference `pkg/version@user/channel`, avoids bugs for conanfile in 4 nested folders and `conan install path/to/the/file` #4044
- Bugfix: Running Windows subsystem scripts crashed when the PATH environment variable passed as a list. #4039
- Bugfix: Fix removal of conanfile.py with **`conan source`** command and the removal of source folder in the local cache when something fails #4033
- Bugfix: A **`conan install`** with a reference failed when running in the operating system root folder because python tried to create the directory even when nothing is going to be written. #4012
- Bugfix: Fix qbs generator mixing sharedlinkflags and exelinkflags #3980
- Bugfix: `compiler_args` generated "mytool.lib.lib" for Visual Studio libraries that were defined with the `.lib` extension in the `self.cpp_info.libs` field of `package_info()`. #3976

22.118 1.9.2 (20-Nov-2018)

- Bugfix: SVN API changes are relevant since version 1.9 #3954
- Bugfix: Fixed bug in *vcvars_dict* tool when using *filter_known_paths* argument. #3941

22.119 1.9.1 (08-Nov-2018)

- Fix: Fix regression introduced in 1.7, setting `amd64_x86` when no `arch_build` is defined. #3918
- Fix: Do not look for binaries in other remotes than the recipe, if it is defined. #3890
- Bugfix: `sudo --askpass` breaks CentOS 6 package installation. The sudo version on CentOS 6 is 1.8.6. The option of `askpass` for sudo version 1.8.7 or older is *sudo -A*. #3885

22.120 1.9.0 (30-October-2018)

- Feature: Support for `srcdirs` in `package_info()`. Packages can package sources, and specify their location, which will be propagated to consumers. Includes support for CMake generator. #3857
- Feature: Added *remote_name* and *remote_url* to upload json output. #3850
- Feature: Add environment variable *CONAN_USE_ALWAYS_SHORT_PATHS* to let the consumer override short_paths behavior from recipes #3846
- Feature: Added `--json` output to `conan export_pkg` command #3809
- Feature: Add *conan remote clean* subcommand #3767
- Feature: New *premake* generator incorporated to the Conan code base from the external generator at https://github.com/memsharded/conan-premake. #3751
- Feature: New *conan remote list_pref/add_pref/remove_pref/update_pref* commands added to manage the new Registry entries for binary packages. #3726
- Feature: Add cpp_info data to json output of `install` and `create` commands at package level. #3717
- Feature: Now the default templates of the **`conan new`** command use the docker images from the *conanio* organization: https://hub.docker.com/u/conanio #3710
- Feature: Added `topics` attribute to the *ConanFile* to specify topics (a.k.a tags, a.k.a keywords) to the recipe. #3702
- Feature: Internal refactor to the remote registry to manage a json file. Also improved internal interface. #3676
- Feature: Implement reuse of sources (`exports_sources`) in recipes used as `python_requires()`. #3661
- Feature: Added support for Clang >=8 and the new versioning schema, where only the major and the patch is used. #3643
- Fix: Renamed Plugins as Hooks #3867
- Fix: Adds GCC 8.2 to default settings.yml #3865
- Fix: Hidden confusing messages *download conaninfo.txt* when requesting the server to check package manifests. #3861
- Fix: The `MSBuild()` build helper doesn't adjust the compiler flags for the build_type anymore because they are adjusted by the project itself. #3860

- Fix: Add `neon` as linux distro for SystemPackageTools #3845
- Fix: remove error that was raised for custom compiler & compiler version, while checking `cppstd` setting. #3844
- Fix: do not allow wildcards in command `conan download <ref-without-wildcards>` #3843
- Fix: do not populate `arch` nor `arch_build` in autodetected profile if `platform.machine` returns an empty string. #3841
- Fix: The registry won't remove a reference to a remote removed recipe or package. #3838
- Fix: Internal improvements of the ConanFile loader #3837
- Fix: environment variables are passed verbatim to generators. #3836
- Fix: Implement dirty checks in the cache build folder, so failed builds are not packaged when there is a `build_id()` method. #3834
- Fix: `vcvars` is also called in the `CMake()` build helper when *clang* compiler is used, not only with Visual Studio`compiler. `#3832
- Fix: Ignore empty line when parsing output inside `SVN::excluded_files` function. #3830
- Fix: Bump version of `tqdm` requirement to `>=4.28.0` #3823
- Fix: Handling corrupted lock files in cache #3816
- Fix: Implement download concurrency checks, to allow simultaneous download of the same package (as header-only) while installing different configurations that depend on that package. #3806
- Fix: `vcvars` is also called in the *CMake()* build helper when using *Ninja* or *NMake* generators. #3803
- Fix: Fixed `link_flags` management in `MSBuild` build helper #3791
- Fix: Allow providing `--profile` argument (and settings, options, env, too) to **`conan export-pkg`**, so it is able to correctly compute the binary package_id in case the information captured in the installed conaninfo.txt in previous **`conan install`** does not contain all information to reconstruct the graph. #3768
- Fix: Upgrade dependency of `tqdm` to >=4.27: solves issue with weakref assertion. #3763
- Fix: Use XML output to retrieve information from SVN command line if its client version is less than 1.8 (command `--show-item` is not available). #3757
- Fix: SVN v1.7 does not have `-r` argument in `svn status`, so functionality `SVN::is_pristine` won't be available. #3757
- Fix: Add `--askpass` argument to `sudo` if it is not an interactive terminal #3727
- Fix: The remote used to download a binary package is now stored, so any update for the specific binary will come from the right remote. #3726
- Fix: Use XML output from SVN command line interface to compute if the repository is pristine. #3653
- Fix: Updated templates of the **`conan new`** command with the latest conan package tools changes. #3651
- Fix: Improve error messages if conanfile was not found #3554
- BugFix: Fix conflicting multiple local imports for python_requires #3876
- Bugfix: do not ask for the username if it is already given when login into a remote. #3839
- Bugfix: `yum update` needs user's confirmation, which breaks system update in CentOS non-interactive terminal. #3747

22.121 1.8.4 (19-October-2018)

- Feature: Increase debugging information when an error uploading a recipe with different timestamp occurs. #3801
- Fix: Changed `tqdm` dependency to a temporarily forked removing the "man" directory write permissions issue installing the *pip* package. #3802
- Fix: Removed *ndg-httpsclient* and *pyasn* dependencies from OSX requirements file because they shouldn't be necessary. #3802

22.122 1.8.3 (17-October-2018)

- Feature: New attributes `default_user` and `default_channel` that can be declared in a conanfile to specify the *user* and *channel* for conan local methods when neither *CONAN_USERNAME* and *CONAN_CHANNEL* environment variables exist. #3758
- Bugfix: AST parsing of `conanfile.py` with shebang and encoding header lines was failing in python 2. This fix also allows non-ascii chars in `conanfile.py` if proper encoding is declared. #3750

22.123 1.8.2 (10-October-2018)

- Fix: Fix misleading warning message in `tools.collect_libs()` #3718
- BugFix: Fixed wrong naming of `--sbindir` and `--libexecdir` in AutoTools build helper. #3715

22.124 1.8.1 (10-October-2018)

- Fix: Remove warnings related to `python_requires()`, both in linter and due to Python2. #3706
- Fix: Use *share* folder for `DATAROOTDIR` in CMake and AutoTools build helpers. #3705
- Fix: Disabled *apiv2* until the new protocol becomes stable. #3703

22.125 1.8.0 (9-October-2018)

- Feature: Allow *conan config install* to install configuration from a folder and not only from compressed files. #3680
- Feature: The environment variable CONAN_DEFAULT_PROFILE_PATH allows the user to define the path (existing) to the default profile that will be used by Conan. #3675
- Feature: New **`conan inspect`** command that provides individual attributes of a recipe, like name, version, or options. Work with `-r=remote` repos too, and is able to produce `--json` output. #3634
- Feature: Validate parameter for ConanFileReference objects to avoid unnecessary checks #3623
- Feature: The environment variable *CONAN_DEFAULT_PROFILE_PATH* allows the user to define the path (absolute and existing) to the default profile that will be used by Conan. #3615
- Feature: Warning message printed if Conan cannot deduce an architecture of a GNU triplet. #3603

- Feature: The `AutotoolsBuildEnvironment` and `CMake` build helpers now adjust default for the GNU standard installation directories: `bindir`, `sbin`, `libexec`, `includedir`, `oldincludedir`, `datarootdir` #3599
- Feature: Added `use_default_install_dirs` in `AutotoolsBuildEnvironment.configure()` to opt-out from the defaulted installation dirs. #3599
- Feature: Clean repeated entries in the `PATH` when `vcvars` is run, mitigating the max size of the env var issues. #3598
- Feature: Allow `vcvars` to run if `clang-cl` compiler is detected. #3574
- Feature: Added python 2 deprecation message in the output of the conan commands. #3567
- Feature: The **`conan install`** command now prints information about the applied configuration. #3561
- Feature: New naming convention for conanfile reserved/public/private attributes. #3560
- Feature: Experimental support for Conan plugins. #3555
- Feature: Progress bars for files unzipping. #3545
- Feature: Improved graph propagation performance from `O(n2)` to `O(n)`. #3528
- Feature: Added `ConanInvalidConfiguration` as the standard way to indicate that a specific configuration is not valid for the current package. e.g library not compatible with Windows. #3517
- Feature: Added `libtool()` function to the *tools.XCRun()* tool to locate the system `libtool`. #3515
- Feature: The tool `tools.collect_libs()` now search into each folder declared in `self.cpp_info.libdirs`. #3503
- Feature: Added definition `CMAKE_OSX_DEPLOYMENT_TARGET` to the `CMake` build helper following the `os.version` setting for `Macos`. #3486
- Feature: The upload of files now uses the *conanmanifest.txt* file to know if a file has to be uploaded or not. It avoids issues associated with the metadata of the files permissions contained in the *tgz* files. #3480
- Feature: The *default_options* in a *conanfile.py* can be specified now as a dictionary. #3477
- Feature: The command *conan config install* now support relative paths. #3468
- Feature: Added a definition *CONAN_IN_LOCAL_CACHE* to the *CMake()* build helper. #3450
- Feature: Improved *AptTool* at *SystemPackageTool* adding a function *add_repository* to add new apt repositories. #3445
- Feature: Experimental and initial support for the REST *apiv2* that will allow transfers in one step and revisions in the future. #3442
- Feature: Improve the output of a **`conan install`** command printing dependencies when a binary is not found. #3438
- Feature: New *b2* generator. It replaces the old incomplete *boost_build* generator that is now deprecated. #3416
- Feature: New `tool.replace_path_in_file` to replace Windows paths in a file doing case-insensitive comparison and indistinct path separators comparison: "/" == "\" #3399
- Feature: **[Experimental]** Add SCM support for SVN. #3192
- Fix: `None` option value was not being propagated upstream in the dependency graph #3684
- Fix: Apply `system_requirements()` always on install, in case the folder was removed. #3647
- Fix: Included `bottle` package in the development requirements #3646
- Fix: More complete architecture list in the detection of the gnu triplet and the detection of the build machine architecture. #3581

- Fix: Avoid downloading the manifest of the recipe twice for uploads. Making this download quiet, without output. #3552
- Fix: Fixed `Git` scm class avoiding to replace any character in the `get_branch()` function. #3496
- Fix: Removed login username syntax checks that were no longer necessary. #3464
- Fix: Removed bad duplicated messages about dependency overriding when using conan alias. #3456
- Fix: Improve **`conan info`** help message. #3415
- Fix: The generator files are only written in disk if the content of the generated file changes. #3412
- Fix: Improved error message when parsing a bad conanfile reference. #3410
- Fix: Paths are replaced correctly on Windows when using `CMake().patch_config_files()`. #3399
- Fix: Fixed *AptTool* at *SystemPackageTool* to improve the detection of an installed package. #3033
- BugFix: Fixes `python_requires` overwritten when using more than one of them in a recipe #3628
- BugFix: Fix output overlap of decompress progress and plugins #3622
- Bugfix: Check if the `system_requirements()` have to be executed even when the package is retrieved from the local cache. #3616
- Bugfix: All API calls are now logged into the `CONAN_TRACE_FILE` log file. #3613
- Bugfix: Renamed `os` (reserved symbol) parameter to `os_` in the `get_gnu_triplet` tool. #3603
- Bugfix: **`conan get`** command now works correctly with enabled `short paths`. #3600
- Bugfix: Fixed `scm` replacement of the variable when exporting a conanfile. #3576
- Bugfix: *apiv2* was retrying the downloads even when a 404 error was raised. #3562
- Bugfix: Fixed `export_sources` excluded patterns containing symlinks. #3537
- Bugfix: Fixed bug with transitive private dependencies. #3525
- Bugfix: `get_cased_path` crashed when the path didn't exist. #3516
- BugFix: Fixed failures when Conan walk directories with files containing not ASCCI characters in the file name. #3505
- Bugfix: The *scm* feature now looks for the repo root even when the *conanfile.py* is in a subfolder. #3479
- Bugfix: Fixed *OSInfo.bash_path()* when there is no *windows_subsystem*. #3455
- Bugfix: AutotoolsBuildEnvironment was not defaulting the output library directory causing broken consumption of packages when rebuilding from sources in different Linux distros using lib64 default. Read more *here*. #3388

22.126 1.7.4 (18-September-2018)

- Bugfix: Fixed a bug in *apiv2*.
- Fix: Disabled *apiv2* by default until it gets more stability.

22.127 1.7.3 (6-September-2018)

- Bugfix: Uncontrolled exception was raised while printing the output of an error downloading a file.
- Bugfix: Fixed **`*:option`** pattern for conanfile consumers.

22.128 1.7.2 (4-September-2018)

- Bugfix: Reverted default options initialization to empty string with *varname=*.
- Bugfix: Fixed *conan build* command with *–test* and *–install* arguments.

22.129 1.7.1 (31-August-2018)

- Fix: Trailing sentences in Conan help command.
- Fix: Removed hardcoded **`-c init.templateDir=`** argument in **`git clone`** for **`conan config install`**, in favor of a new **`--args`** parameter that allows custom arguments.
- Fix: SCM can now handle nested subfolders.
- BugFix: Fix **`conan export-pkg`** unnecessarily checking remotes.

22.130 1.7.0 (29-August-2018)

- Feature: Support for C++20 in CMake > 3.12.
- Feature: Included support for Python 3.7 in all platforms.
- Feature: **[Experimental]** New `python_requires` function that allows you to reuse Python code by "requiring" it in Conan packages, even to extend the `ConanFile` class. See: *Python requires: reusing python code in recipes*
- Feature: `CMake` method `patch_config_paths` replaces absolute paths to a Conan package's dependencies as well as to the Conan package itself.
- Feature: `MSBuild` and `VisualStudioBuildEnvironment` build helpers adjust the `/MP` flag to build code in parallel using multiple cores.
- Feature: Added a `print_errors` parameter to `tools.PkgConfig()` helper.
- Feature: Added **`--query`** argument to **`conan upload`**.
- Feature: `virtualenv/virtualbuildenv/virtualrunenv` generators now create bash scripts in Windows for use in subsystems.
- Feature: Improved resolution speed for version ranges through caching of remote requests.
- Feature: Improved the result of `tools.vcvars_dict(only_diff=True)` including a "list" return type that can be used with `tools.environment_append()`.
- Fix: `AutoToolsBuildEnvironment` build helper now keeps the `PKG_CONFIG_PATHS` variable previously set in the environment.
- Fix: The SCM feature keeps the `.git` folder during the copy of a local directory to the local cache.
- Fix: The SCM feature now correctly excludes the folders ignored by Git during the copy of a local directory to the local cache.

- Fix: Conan messages now spell "overridden" correctly.
- Fix: `MSBuild` build helper arguments using quotes.
- Fix: `vcvars_command` and `MSBuild` build helper use the `amd64_x86` parameter when Visual Studio > 12 and when cross building for x86.
- Fix: Disabled `-c init.TemplateDir` in **`conan config install`** from a Git repository.
- Fix: Clang compiler check in `cmake` generator.
- Fix: Detection of Zypper package tool on latest versions of openSUSE.
- Fix: Improved help output of some commands.
- BugFix: `qmake` generator hyphen.
- Bugfix: Git submodules are now initialized from repo *HEAD* **after** checking out the referenced revision when using the `scm` attribute.
- BugFix: Declaration `default_options` without value, e.g. `default_options = "config="`. Now it will throw an exception.
- BugFix: Deactivate script in `virtualenv` generator causes PS1 to go unset.
- BugFix: Apply general scope options to a consumer ConanFile first.
- BugFix: Fixed detection of a valid repository for Git in the SCM feature.

22.131 1.6.1 (27-July-2018)

- Bugfix: **`conan info --build-order`** was showing duplicated nodes for build-requires and private dependencies.
- Fix: Fixed failure with the `alias` packages when the name of the package (excluded the version) was different from the aliased package. Now it is limited in the **`conan alias`** command.

22.132 1.6.0 (19-July-2018)

- Feature: Added a new `self.run(..., run_environment=True)` argument, that automatically applies `PATH`, `LD_LIBRARY_PATH` and `DYLD_LIBRARY_PATH` environment variables from the dependencies, to the execution of the current command.
- Feature: Added a new `tools.run_environment()` method as a shortcut to using `tools.environment_append` and `RunEnvironment()` together.
- Feature: Added a new `self.run(..., ignore_errors=True)` argument that represses launching an exception if the commands fails, so user can capture the return code.
- Feature: Improved `tools.Git` to allow capturing the current branch and enabling the export of a package whose version is based on the branch and commit.
- Feature: The `json` generator now outputs settings and options
- Feature: **`conan remote list --raw`** prints remote information in a format valid for *remotes.txt*, so it can be used for `conan config install`
- Feature: Visual Studio generator creates the *conanbuildinfo.props* file using the `$(USERPROFILE)` macro.
- Feature: Added a `filename` parameter to `tools.get()` in case it cannot be deduced from the URL.

- Feature: Propagated `keep_permissions` and `pattern` parameters from `tools.get()` to `tools.unzip()`.
- Feature: Added XZ extensions to `unzip()`. This will only work in Python 3 with lzma support enabled, otherwise, and error is produced.
- Feature: Added `FRAMEWORK_SEARCH_PATHS` var to the Xcode generator to support packaging Apple Frameworks. Read more *here*.
- Feature: Added **`conan build --test`** and a `should_configure` attribute to control the test stage. Read more *here*.
- Feature: New tools to convert between files with LF and CRLF line endings: *tools.unix2dos()* and *tools.dos2unix()*.
- Feature: Added **`conan config install [url] --type=git`** to force cloning of a Git repo for `http://...` git urls.
- Feature: Improved output information when a package is missing in a remote to show which package requires the missing one.
- Feature: Improved the management of an upload interruption to avoid uploads of incomplete tarballs.
- Feature: Added new `LLVM` toolsets to the base `settings.yml` (Visual Studio).
- Feature: Created a plugin for pylint with the previous Conan checks (run in the export) enabling usage of the plugin in IDEs and command line to check if recipes are correct.
- Feature: Improved the `deb` installer to guarantee that it runs correctly in Debian 9 and other distros.
- Fix: Fixed **`conan search -q`** and **`conan remove -q`** to not return packages that don't have the setting specified in the query.
- Fix: Fixed `SystemPackageTool` when calling to update with `sudo` is not enabled and `mode=verify`.
- Fix: Removed `pyinstaller` shared libraries from the linker environment for any Conan subprocess.
- BugFix: The `YumTool` now calls `yum update` instead of `yum check-update`.
- Bugfix: Solved a bug in which using `--manifest` parameter with **`conan create`** caused the deletion of information in the dependency graph.
- Bugfix: Solved bug in which the `build` method of the `Version` model was not showing the version build field correctly .
- Bugfix: Fixed a Conan crash caused by a dependency tree containing transitive private nodes.

22.133 1.5.2 (5-July-2018)

- Bugfix: Fixed bug with pre-1.0 packages with sources.
- Bugfix: Fixed regression in private requirements.

22.134 1.5.1 (29-June-2018)

- Bugfix: Sources in the local cache weren't removed when using scm pointing to the local source directory, causing changes in local sources not applied to the conan create process.
- Bugfix: Fixed bug causing duplication of build requires in the dependency graph.

22.135 1.5.0 (27-June-2018)

- Feature: **conan search <pkg-ref> -r=all** now is able to search for binaries too in all remotes
- Feature: Dependency graph improvements: `build_requires` are represented in the graph (visible in **conan info**`, also in the HTML graph). **conan install** and **conan info** commands shows extended information of the binaries status (represented in colors in HTML graph). The dependencies declaration order in recipes is respected (as long as it doesn't break the dependency graph order).
- Feature: improved remote management, it is possible to get binaries from different remotes.
- Feature: **conan user** command is now able to show authenticated users.
- Feature: Added **conan user --json** json output to the command.
- Feature: New `pattern` argument to `tools.unzip()` and `tools.untargz` functions, that allow efficient extraction of certain files only.
- Feature : Added Manjaro support for `SystemPackageTools`.
- Feature: Added `Macos` `version` subsetting in the default *settings.yml* file, to account for the "min OSX version" configuration.
- Feature: SCM helper argument to recursively clone submodules
- Feature: SCM helper management of subfolder, allows using `exports` and `exports_sources`, manage symlinks, and do not copy files that are *.gitignored*. Also, works better in the local development flow.
- Feature: Modifies user agent header to output the Conan client version and the Python version. Example: `Conan/1.5.0 (Python 2.7.1)`
- Fix: The `CMake()` helper now doesn't require a compiler input to deduce the default generator.
- Fix: **conan search <pattern>** now works consistently in local cache and remotes.
- Fix: Proxy related environment variables are removed if *conan.conf* declares proxy configuration.
- Fix: Fixed the parsing of invalid JSON when Microsoft `vswhere` tool outputs invalid non utf-8 text.
- Fix: Applying `winsdk` and `vcvars_ver` to MSBuild and `vcvars_command` for VS 14 too.
- Fix: Workspaces now support `build_requires`.
- Fix: `CMake()` helper now defines by default `CMAKE_EXPORT_NO_PACKAGE_REGISTRY`.
- Fix: Settings constraints declared in recipes now don't error for single strings (instead of a list with a string element).
- Fix: `cmake_minimum_required()` is now before `project()` in templates and examples.
- Fix: `CONAN_SYSREQUIRES_MODE=Disabled` now doesn't try to update the system packages registry.
- Bugfix: Fixed SCM origin path of windows folder (with backslashes).
- Bugfix: Fixed SCM dictionary order when doing replacement.
- Bugfix: Fixed auto-detection of apple-clang 10.0.

- Bugfix: Fixed bug when doing a **conan search** without registry file (just before installation).

22.136 1.4.5 (22-June-2018)

- Bugfix: The package_id recipe method was being called twice causing issues with info objects being populated with wrong information.

22.137 1.4.4 (11-June-2018)

- Bugfix: Fix link order with private requirements.
- Bugfix: Removed duplicate `-std` flag in CMake < 3 or when the standard is not yet supported by `CMAKE_CXX_STANDARD`.
- Bugfix: Check `scm` attribute to avoid breaking recipes with already defined one.
- Feature: Conan workspaces.

22.138 1.4.3 (6-June-2018)

- Bugfix: Added system libraries to the cmake_find_package generator.
- Fix: Added SIGTERM signal handler to quit safely.
- Bugfix: Fixed miss-detection of gcc 1 when no gcc was on a Linux machine.

22.139 1.4.2 (4-June-2018)

- Bugfix: Fixed multi-config packages.
- Bugfix: Fixed *cppstd* management with CMake and 20 standard version.

22.140 1.4.1 (31-May-2018)

- Bugfix: Solved issue with symlinks making recipes to fail with `self.copy`.
- Bugfix: Fixed c++20 standard usage with modern compilers and the creation of the *settings.yml* containing the settings values.
- Bugfix: Fixed error with cased directory names in Windows.
- BugFix: Modified confusing warning message in the SCM tool when the remote couldn't be detected.

22.141 1.4.0 (30-May-2018)

- Feature: Added `scm` conanfile attribute, to easily clone/checkout from remote repositories and to capture the remote and commit in the exported recipe when the recipe and the sources lives in the same repository. Read more in "*Recipe and sources in a different repo*" and "*Recipe and sources in the same repo*".
- Feature: Added `cmake_paths` generator to create a file setting `CMAKE_MODULE_PATH` and `CMAKE_PREFIX_PATH` to the packages folders. It can be used as a CMake toolchain to perform a transparent CMake usage, without include any line of cmake code related to Conan. Read more *here*.
- Feature: Added `cmake_find_package` generator that generates one `FindXXX.cmake` file per each dependency both with classic CMake approach and modern using transitive CMake targets. Read more *here*.
- Feature: Added **`conan search --json`** json output to the command.
- Feature: CMake build helper now sets `PKG_CONFIG_PATH` automatically and receives new parameter `pkg_config_paths` to override it.
- Feature: CMake build helper doesn't require to specify "arch" nor "compiler" anymore when the generator is "Unix Makefiles".
- Feature: Introduced default settings for GCC 8, Clang 7.
- Feature: Introduced support for c++ language standard c++20.
- Feature: Auto-managed `fPIC` option in AutoTools build helper.
- Feature: `tools.vcvars_command()` and `tools.vcvars_dict()` now take `vcvars_ver` and `winsdk_version` as parameters.
- Feature: `tools.vcvars_dict()` gets only the env vars set by vcvars with new parameter `only_diff=True`.
- Feature: Generator `virtualbuildenv` now sets Visual Studio env vars via `tool.vcvars_dict()`.
- Feature: New tools for Apple development including XCRun wrapper.
- Fix: Message "Package '1' created" in package commands with `short_paths=True` now shows package ID.
- Fix: `tools.vcvars_dict()` failing to create dictionary due to newlines in vcvars command output.
- Bugfix: `tools.which()` returning directories instead of only files.
- Bugfix: Inconsistent local cache when developing a recipe with `short_paths=True`.
- Bugfix: Fixed reusing MSBuild() helper object for multi-configuration packages.
- Bugfix: Fixed authentication using env vars such as `CONAN_PASSWORD` when `CONAN_NON_INTERACTIVE=True`.
- Bugfix: Fixed Android api_level was not used to adjust CMAKE_SYSTEM_VERSION.
- Bugfix: Fixed MSBuild() build helper creating empty XML node for runtime when the setting was not declared.
- Bugfix: Fixed `default_options` not supporting = in value when specified as tuple.
- Bugfix: AutoToolsBuildEnvironment build helper's `pkg_config_paths` parameter now sets paths relative to the install folder or absolute ones if provided.

22.142 1.3.3 (10-May-2018)

- Bugfix: Fixed encoding issues writing to files and calculating md5 sums.

22.143 1.3.2 (7-May-2018)

- Bugfix: Fixed broken `run_in_windows_bash` due to wrong argument.
- Bugfix: Fixed `VisualStudioBuildEnvironment` when toolset was not defined.
- Bugfix: Fixed md5 computation of conan .tgz files for recipe, exported sources and packages due to file ordering and flags.
- Bugfix: Fixed `conan download -p=wrong_id` command
- Fix: Added apple-clang 9.1

22.144 1.3.1 (3-May-2018)

- Bugfix: Fixed regression with `AutoToolsBuildEnvironment` build helper that raised exception with not supported architectures during the calculation of the GNU triplet.
- Bugfix: Fixed `pkg_config` generator, previously crashing when there was no library directories in the requirements.
- Bugfix: Fixed `conanfile.run()` with `win_bash=True` quoting the paths correctly.
- Bugfix: Recovered parameter "append" to the `tools.save` function.
- Bugfix: Added support (documented but missing) to delete options in `package_id()` method using `del self.info.options.<option>`

22.145 1.3.0 (30-April-2018)

- Feature: Added new build types to default `settings.yml`: **RelWithDebInfo** and **MinSizeRel**. Compiler flags will be automatically defined in build helpers that do not understand them (`MSBuild`, `AutotoolsBuildEnvironment`)
- Feature: Improved package integrity. Interrupted downloads or builds shouldn't leave corrupted packages.
- Feature: Added **`conan upload --json`** json output to the command.
- Feature: new **`conan remove --locks`** to clear cache locks. Useful when killing conan.
- Feature: New **CircleCI** template scripts can be generated with the **`conan new`** command.
- Feature: The CMake() build helper manages the fPIC flag automatically based on the options `fPIC` and `shared` when present.
- Feature: Allowing requiring color output with `CONAN_COLOR_DISPLAY=1` environment variable. If `CONAN_COLOR_DISPLAY` is not set rely on tty detection for colored output.
- Feature: New **`conan remote rename`** and **`conan add --force`** commands to handle remotes.
- Feature: Added parameter `use_env` to the `MSBuild().build()` build helper method to control the `/p:UseEnv` msbuild argument.

- Feature: Timeout for downloading files from remotes is now configurable (defaulted to 60 seconds)
- Feature: Improved Autotools build helper with new parameters and automatic set of `--prefix` to `self.package_folder`.
- Feature: Added new tool to compose GNU like triplets for cross-building: `tools.get_gnu_triplet()`
- Fix: Use International Units for download/upload transfer sizes (Mb, Kb, etc).
- Fix: Removed duplicated paths in `cmake_multi` generated files.
- Fix: Removed false positive linter warning for local imports.
- Fix: Improved command line help for positional arguments
- Fix **`-ks`** alias for **`--keep-source`** argument in **`conan create`** and **`conan export`**.
- Fix: removed confusing warnings when `self.copy()` doesn't copy files in the `package()` method.
- Fix: `None` is now a possible value for settings with nested subsettings in `settings.yml`.
- Fix: if `vcvars_command` is called and Visual is not found, raise an error instead of warning.
- Bugfix: `self.env_info.paths` and `self.env_info.PATHS` both map now to PATHS env-var.
- Bugfix: Local flow was not correctly recovering state for option values.
- Bugfix: Windows NTFS permissions failed in case USERDOMAIN env-var was not defined.
- Bugfix: Fixed generator `pkg_config` when there are absolute paths (not use prefix)
- Bugfix: Fixed parsing of settings values with "=" character in conaninfo.txt files.
- Bugfix: Fixed misdetection of MSYS environments (generation of default profile)
- Bugfix: Fixed string escaping in CMake files for preprocessor definitions.
- Bugfix: `upload --no-overwrite` failed when the remote package didn't exist.
- Bugfix: Don't raise an error if `detect_windows_subsystem` doesn't detect a subsystem.

22.146 1.2.3 (10-Apr-2017)

- Bugfix: Removed invalid version field from scons generator.

22.147 1.2.1 (3-Apr-2018)

- Feature: Support for *apple-clang 9.1*
- Bugfix: *compiler_args* generator manage correctly the flag for the *cppstd* setting.
- Bugfix: Replaced exception with a warning message (recommending the *six* module) when using *StringIO* class from the *io* module.

22.148 1.2.0 (28-Mar-2018)

- Feature: The command **conan build** has new `--configure, --build, --install` arguments to control the different stages of the `build()` method.
- Feature: The command **conan export-pkg** now has a **--package-folder** that can be used to export an exact copy of the provided folder, irrespective of the `package()` method. It assumes the package has been locally created with a previous **conan package** or with a **conan build** using a `cmake.install()` or equivalent feature.
- Feature: New `json` generator, generates a json file with machine readable information from dependencies.
- Feature: Improved proxies configuration with `no_proxy_match` configuration variable.
- Feature: New **conan upload** parameter **--no-overwrite** to forbid the overwriting of recipe/packages if they have changed.
- Feature: Exports are now copied to `source_folder` when doing **conan source**.
- Feature: `tools.vcvars()` context manager has no effect if platform is different from Windows.
- Feature: **conan download** has new optional argument **--recipe** to download only the recipe of a package.
- Feature: Added `CONAN_NON_INTERACTIVE` environment variable to disable interactive prompts.
- Feature: Improved `MSbuild()` build helper using `vcvars()` and generating property file to adjust the runtime automatically. New method `get_command()` with the call to `msbuild` tool. Deprecates `tools.build_sln_command()` and `tools.msvc_build_command()`.
- Feature: Support for clang 6.0 correctly managing cppstd flags.
- Feature: Added configuration to specify a client certificate to connect to SSL server.
- Feature: Improved `ycm` generator to show json dependencies.
- Feature: Experimental `--json` parameter for **conan install** and **conan create** to generate a JSON file with install information.
- Fix: **conan install --build** does not absorb more than one parameter.
- Fix: Made conanfile templates generated with **conan new** PEP8 compliant.
- Fix: **conan search** output improved when there are no packages for the given reference.
- Fix: Made **conan download** also retrieve sources.
- Fix: Pylint now runs as an external process.
- Fix: Made `self.user` and `self.channel` available in test_package.
- Fix: Made files writable after a `deploy()` or `imports()` when `CONAN_READ_ONLY_CACHE`/general.read_only_cache` environment/config variable is `True`.
- Fix: Linter showing warnings with `cpp_info` object in `deploy()` method.
- Fix: Disabled linter for Conan pyinstaller as it was not able to find the python modules.
- Fix: **conan user -r=remote_name** showed all users for all remotes, not the one given.
- BugFix: Python reuse code failing to import module in `package_info()`.
- BugFix: Added escapes for backslashes in `cmake` generator.
- BugFix: **conan config install** now raises error if **git clone** fails.
- BugFix: Alias resolution not working in diamond shaped dependency trees.

- BugFix: Fixed builds with Cygwin/MSYS2 failing in Windows with *self.short_paths=True* and NTFS file systems due to ACL permissions.
- BugFix: Failed to adjust architecture when running Conan platform detection in ARM devices.
- BugFix: Output to StringIO failing in Python 2.
- BugFix: **conan profile update** not working to update `[env]` section.
- BugFix: **conan search** not creating default remotes when running it as the very first command after Conan installation.
- BugFix: Package folder was not cleaned after the installation and download of a package had failed.

22.149 1.1.1 (5-Mar-2018)

- Feature: `build_sln_command()` and `msvc_build_command()` receive a new optional parameter `platforms` to match the definition of the *.sln* Visual Studio project architecture. (Typically Win32 vs x86 problem).
- Bugfix: Flags for Visual Studio command (cl.exe) using "-" instead of "/" to avoid problems in builds using AutoTools scripts with Visual Studio compiler.
- Bugfix: Visual Studio runtime flags adjusted correctly in `AutoToolsBuildEnvironment()` build helper
- Bugfix: `AutoToolsBuildEnvironment()` build helper now adjust the correct build flag, not using eabi suffix, for architecture x86.

22.150 1.1.0 (27-Feb-2018)

- Feature: New **conan create --keep-build** option that allows re-packaging from conan local cache, without re-building.
- Feature: **conan search <pattern> -r=all** now searches in all defined remotes.
- Feature: Added setting `cppstd` to manage the C++ standard. Also improved build helpers to adjust the standard automatically when the user activates the setting. `AutoToolsBuildEnvironment()`, `CMake()`, `MSBuild()` and `VisualStudioBuildEnvironment()`.
- Feature: New `compiler_args` generator, for directly calling the compiler from command line, for multiple compilers: VS, gcc, clang.
- Feature: Defined `sysrequires_mode` variable (`CONAN_SYSREQUIRES_MODE` env-var) with values `enabled, verify, disabled` to control the installation of system dependencies via `SystemPackageTool` typically used in *system_requirements()*.
- Feature: automatically apply `pythonpath` environment variable for dependencies containing python code to be reused to recipe `source()`, `build()`, `package()` methods.
- Feature: `CMake` new `patch_config_paths()` methods that will replace absolute paths to conan package path variables, so cmake find scripts are relocatable.
- Feature: new **--test-build-folder** command line argument to define the location of the *test_package* build folder, and new conan.conf `temp_test_folder` and environment variable `CONAN_TEMP_TEST_FOLDER`, that if set to True will automatically clean the test_package build folder after running.
- Feature: Conan manages relative urls for upload/download to allow access the server from different configured networks or in domain subdirectories.

- Feature: Added `CONAN_SKIP_VS_PROJECTS_UPGRADE` environment variable to skip the upgrade of Visual Studio project when using *tools.build_sln_command() [DEPRECATED]*, the *msvc_build_command* and the *MSBuild()* build helper.
- Feature: Improved detection of Visual Studio installations, possible to prioritize between multiple installed Visual tools with the `CONAN_VS_INSTALLATION_PREFERENCE` env-var and `vs_installation_preference` conan.conf variable.
- Feature: Added `keep_path` parameter to `self.copy()` within the `imports()` method.
- Feature: Added `[build_requires]` section to *conanfile.txt*.
- Feature: Added new **conan help <command>** command, as an alternative to **--help**.
- Feature: Added `target` parameter to `AutoToolsBuildEnvironment.make` method, allowing to select build target on running make
- Feature: The `CONAN_MAKE_PROGRAM` environment variable now it is used by the `CMake()` build helper to set a custom make program.
- Feature: Added **--verify-ssl** optional parameter to **conan config install** to allow self-signed SSL certificates in download.
- Feature: `tools.get_env()` helper method to automatically convert environment variables to python types.
- Fix: Added a visible warning about `libcxx` compatibility and the detected one for the default profile.
- Fix: Wrong detection of compiler in OSX for gcc frontend to clang.
- Fix: Disabled *conanbuildinfo.cmake* compiler checks for unknown compilers.
- Fix: `visual_studio` generator added missing *ResourceCompile* information.
- Fix: Don't output password from URL for **conan config install** command.
- Fix: Signals exit with error code instead of 0.
- Fix: Added package versions to generated SCons file.
- Fix: Error message when package was not found in remotes has been improved.
- Fix: **conan profile** help message.
- Fix: Use gcc architecture flags -m32, -m64 for MinGW as well.
- Fix: `CMake()` helper do not require settings if `CONAN_CMAKE_GENERATOR` is defined.
- Fix: improved output of package remote origins.
- Fix: Profiles files use same structure as **conan profile show** command.
- Fix: *conanpath.bat* file is removed after conan Windows installer uninstall.
- Fix: Do not add GCC-style flags -m32, -m64, -g, -s to MSVC when using `AutoToolsBuildEnvironment`
- Fix: "Can't find a binary package" message now includes the Package ID.
- Fix: added clang 5.0 and gcc 7.3 to default *settings.yml*.
- Bugfix: `build_id()` logic does not apply unless the `build_id` is effectively changed.
- Bugfix: `self.install_folder` was not correctly set in all necessary cases.
- Bugfix: **--update** option does not ignore local packages for version-ranges.
- Bugfix: Set `self.develop=True` for `export-pkg` command.
- Bugfix: Server HTTP responses were incorrectly captured, not showing errors for some server errors.

- Bugfix: Fixed `config` section update for sequential calls over the python API.
- Bugfix: Fixed wrong `self.develop` set to `False` for **`conan create`** with *test_package*.
- Deprecation: Removed **conan-transit** from default remotes registry.

22.151 1.0.4 (30-January-2018)

- Bugfix: Fixed default profile defined in *conan.conf* that includes another profile
- Bugfix: added missing management of `sysroot` in *conanbuildinfo.txt* affecting **`conan build`** and *test_package*.
- Bugfix: Fixed warning in **`conan source`** because of incorrect management of settings.
- Bugfix: Fixed priority order of environment variables defined in included profiles
- Bugfix: NMake error for parallel builds from the `CMake` build helper have been fixed
- Bugfix: Fixed options pattern not applied to root node (`-o *:shared=True` not working for consuming package)
- Bugfix: Fixed shadowed options by package name (`-o *:shared=True -o Pkg:other=False` was not applying `shared` value to Pkg)
- Fix: Using `filter_known_paths=False` as default to `vcvars_dict()` helper.
- Fix: Fixed wrong package name for output messages regarding build-requires
- Fix: Added correct metadata to conan.exe when generated via pyinstaller

22.152 1.0.3 (22-January-2018)

- Bugfix: Correct load of stored settings in conaninfo.txt (for **`conan build`**) when `configure()` remove some setting.
- Bugfix: Correct use of unix paths in Windows subsystems (msys, cygwin) when needed.
- Fix: fixed wrong message for **`conan alias --help`**.
- Fix: Normalized all arguments to **`--xxx-folder`** in command line help.

22.153 1.0.2 (16-January-2018)

- Fix: Adding a warning message for simultaneous use of `os` and `os_build` settings.
- Fix: Do not raise error from *conanbuildinfo.cmake* for Intel MSVC toolsets.
- Fix: Added more architectures to default *settings.yml* `arch_build` setting.
- Fix: using **`--xxx-folder`** in command line help messages.
- Bugfix: using quotes for Windows bash path with spaces.
- Bugfix: vcvars/vcvars_dict not including windows and windows/system32 directories in the path.

22.154 1.0.1 (12-January-2018)

- Fix: **conan new** does not generate cross-building (like `os_build`) settings by default. They make only sense for dev-tools used as `build_requires`
- Fix: *conaninfo.txt* file does not dump settings with None values

22.155 1.0.0 (10-January-2018)

- Bugfix: Fixed bug from `remove_from_path` due to Windows path backslash
- Bugfix: Compiler detection in *conanbuildinfo.cmake* for Visual Studio using toolchains like LLVM (Clang)
- Bugfix: Added quotes to bash path.

22.156 1.0.0-beta5 (8-January-2018)

- Fix: Errors from remotes different to a 404 will raise an error. Disconnected remotes have to be removed from remotes or use explicit remote with `-r myremote`
- Fix: cross-building message when building different architecture in same OS
- Fix: **conan profile show** now shows profile with same syntax as profile files
- Fix: generated test code in **conan new** templates will not run example app if cross building.
- Fix: **conan export-pkg** uses the *conanfile.py* folder as the default **--source-folder**.
- Bugfix: **conan download** didn't download recipe if there are no binaries. Force recipe download.
- Bugfix: Fixed blocked `self.run()` when stderr outputs large tests, due to full pipe.

22.157 1.0.0-beta4 (4-January-2018)

- Feature: `run_in_windows_bash` accepts a dict of environment variables to be prioritized inside the bash shell, mainly intended to control the priority of the tools in the path. Use with `vcvars` context manager and `vcvars_dict`, that returns the PATH environment variable only with the Visual Studio related directories
- Fix: Adding all values to `arch_target`
- Fix: **conan new** templates now use new `os_build` and `arch_build` settings
- Fix: Updated `CMake` helper to account for `os_build` and `arch_build` new settings
- Fix: Automatic creation of *default* profile when it is needed by another one (like `include(default)`)
- BugFix: Failed installation (non existing package) was leaving lock files in the cache, reporting a package for **conan search**.
- BugFix: Environment variables are now applied to `build_requirements()` for **conan install .**.
- BugFix: Dependency graph was raising conflicts for diamonds with **alias** packages.
- BugFix: Fixed **conan export-pkg** after a **conan install** when recipe has options.

22.158 1.0.0-beta3 (28-December-2017)

- Fix: Upgraded pylint and astroid to latest
- Fix: Fixed `build_requires` with transitive dependencies to other build_requires
- Fix: Improved pyinstaller creation of executable, to allow for py3-64 bits (windows)
- Deprecation: removed all **`--some_argument`**, use instead **`--some-argument`** in command line.

22.159 1.0.0-beta2 (23-December-2017)

- Feature: New command line UI. Most commands use now the path to the package recipe, like **`conan export . user/testing`** or **`conan create folder/myconanfile.py user/channel`**.
- Feature: Better cross-compiling. New settings model for `os_build`, `arch_build`, `os_target`, `arch_target`.
- Feature: Better Windows OSS ecosystem, with utilities and settings model for MSYS, Cygwin, Mingw, WSL
- Feature: `package()` will not warn of not copied files for known use cases.
- Feature: reduce the scope of definition of `cpp_info`, `env_info`, `user_info` attributes to `package_info()` method, to avoid unexpected errors.
- Feature: extended the use of addressing folder and conanfiles with different names for `source`, `package` and `export-pkg` commands
- Feature: added support for Zypper system package tool
- Fix: Fixed application of build requires from profiles that didn't apply to requires in recipes
- Fix: Improved "test package" message in output log
- Fix: updated CI templates generated with **`conan new`**
- Deprecation: Removed `self.copy_headers` and family for the `package()` method
- Deprecation: Removed `self.conanfile_directory` attribute.

Note: This is a beta release, shouldn't be installed unless you do it explicitly

$ pip install conan==1.0.0b2 –upgrade

Breaking changes

- The new command line UI breaks command line tools and integration. Most cases, just add a `.` to the command.
- Removed `self.copy_headers`, `self.copy_libs`, methods for `package()`. Use `self.copy()` instead.
- Removed `self.conanfile_directory` attribute. Use `self.source_folder`, `self.build_folder`, etc. instead

22.160 0.30.3 (15-December-2017)

- Reverted `CMake()` and `Meson()` build helpers to keep old behavior.
- Forced Astroid dependency to < 1.6 because of py3 issues.

22.161 0.30.2 (14-December-2017)

- Fix: `CMake()` and `Meson()` build helpers and relative directories regression.
- Fix: `ycm` generator, removed the access of `cpp_info` to generators, keeping the access to `deps_cpp_info`.

22.162 0.30.1 (12-December-2017)

- Feature: Introduced major versions for gcc (5, 6, 7) as defaults settings for OSS packages, as minors are compatible by default
- Feature: `VisualStudioBuildEnvironment` has added more compilation and link flags.
- Feature: new `MSBuild()` build helper that wraps the call to `msvc_build_command()` with the correct application of environment variables with the improved `VisualStudioBuildEnvironment`
- Feature: `CMake` and `Meson` build helpers got a new `cache_build_dir` argument for `configure(cache_build_dir=None)` that will be used to define a build directory while the package is being built in local cache, but not when built locally
- Feature: `conanfiles` got a new `apply_env` attribute, defaulted to `True`. If false, the environment variables from dependencies will not be automatically applied. Useful if you don't want some dependency adding itself to the PATH by default, for example
- Feature: allow recipes to use and run python code installed with **`conan config install`**.
- Feature: `conanbuildinfo.cmake` now has `KEEP_RPATHS` as argument to keep the RPATHS, as opposed to old SKIP_RPATH which was confusing. Also, it uses set(CMAKE_INSTALL_NAME_DIR "") to keep the old behavior even for CMake >= 3.9
- Feature: **`conan info`** is able to get profile information from the previous install, instead of requiring it as input again
- Feature: `tools.unix_path` support MSYS, Cygwin, WSL path flavors
- Feature: added `destination` folder argument to `tools.get()` function
- Feature: `SystemPackageTool` for apt-get now uses **`--no-install-recommends`** automatically.
- Feature: `visual_studio_multi` generator now uses toolsets instead of IDE version to identify files.
- Fix: generators failures print traces to help debugging
- Fix: typos in generator names, or non-existing generator now raise an Error instead of a warning
- Fix: `short_paths` feature is active by default in Windows. If you want to opt-out, you can use `CONAN_USER_HOME_SHORT=None`
- Fix: `SystemPackageTool` doesn't use sudo in Windows
- BugFix: Not using parallel builds for Visual<10 in CMake build helper.

- Deprecation: `conanfile_directory` shouldn't be used anymore in recipes. Use ``source_folder``, ``build_folder``, etc.

Note: Breaking changes

- `scopes` have been completely removed. You can use environment variables, or the `conanfile.develop` or `conanfile.in_local_cache` attributes instead.
- Command *test_package* has been removed. Use **`conan create`** instead, and **`conan test`** for just running package tests.
- `werror` behavior is now by default. Dependencies conflicts will now error, and have to be fixed.
- `short_paths` feature is again active by default in Windows, even with Py3.6 and system LongPathsEnabled.
- `ConfigureEnvironment` and `GCC` build helpers have been completely removed

22.163 0.29.2 (2-December-2017)

- Updated python cryptography requirement for OSX due the pyOpenSSL upgrade. See more: https://pypi.org/project/pyOpenSSL/

22.164 0.29.1 (23-November-2017)

- Support for OSX High Sierra
- Reverted concurrency locks to counters, removed `psutil` dependency
- Implemented migration for settings.yml (for new VS toolsets)
- Fixed encoding issues in conan_server

22.165 0.29.0 (21-November-2017)

- Feature: Support for WindowsStore (WinRT, UWP)
- Feature: Support for Visual Studio Toolsets.
- Feature: New `boost-build` generator for generic bjam (not only Boost)
- Feature: new `tools.PkgConfig` helper to parse pkg-config (.pc) files.
- Feature: Added `self.develop` conanfile variable. It is true for **`conan create`** packages and for local development.
- Feature: Added `self.keep_imports` to avoid removal of imported files in the `build()` method. Convenient for re-packaging.
- Feature: Autodetected MSYS2 for `SystemPackageTool`
- Feature: `AutoToolsBuildEnvironment` now auto-loads `pkg_config_path` (to use with `pkg_config` generator)
- Feature: Changed search for profiles. Profiles not found in the default `profiles` folder, will be searched for locally. Use `./myprofile` to force local search only.

- Feature: Parallel builds for Visual Studio (previously it was only parallel compilation within builds)
- Feature: implemented syntax to check options with `if "something" in self.options.myoption`
- Fix: Fixed CMake dependency graph when using TARGETS, that produced wrong link order for transitive dependencies.
- Fix: Trying to download the `exports_sources` is not longer done if such attribute is not defined
- Fix: Added output directories in `cmake` generator for RelWithDebInfo and MinSizeRel configs
- Fix: Locks for concurrent access to local cache now use process IDs (PIDs) to handle interruptions and inconsistent states. Also, adding messages when locking.
- Fix: Not remove the .zip file after a **`conan config install`** if such file is local
- Fix: Fixed `CMake.test()` for the Ninja generator
- Fix: Do not create local conaninfo.txt file for **`conan install <pkg-ref>`** commands.
- Fix: Solved issue with multiple repetitions of the same command line argument
- BugFix: Don't rebuild conan created (with conan-create) packages when `build_policy="always"`
- BugFix: **`conan copy`** was always copying binaries, now can copy only recipes
- BugFix: A bug in download was causing appends instead of overwriting for repeated downloads.
- Development: Large restructuring of files (new cmd and build folders)
- Deprecation: Removed old CMake helper methods (only valid constructor is `CMake(self)`)
- Deprecation: Removed old `conan_info()` method, that was superseded by `package_id()`

Note: Breaking changes

- CMAKE_LIBRARY_OUTPUT_DIRECTORY definition has been introduced in `conan_basic_setup()`, it will send shared libraries .so to the `lib` folder in Linux systems. Right now it was undefined.
- Profile search logic has slightly changed. For `-pr=myprofile`, such profile will be searched both in the default folder and in the local one if not existing. Use `-pr=./myprofile` to force local search only.
- The **`conan copy`** command has been fixed. To copy all binaries, it is necessary to explicit **`--all`**, as other commands do.
- The only valid use of CMake helper is `CMake(self)` syntax.
- If using `conan_info()`, replace it with `package_id()`.
- Removed environment variable `CONAN_CMAKE_TOOLSET`, now the toolset can be specified as a subsetting of Visual Studio compiler or specified in the build helpers.

22.166 0.28.1 (31-October-2017)

- BugFix: Downloading (`tools.download`) of files with `content-encoding=gzip` were raising an exception because the downloaded content length didn't match the http header `content-length`

22.167 0.28.0 (26-October-2017)

This is a big release, with many important and core changes. Also with a huge number of community contributions, thanks very much!

- Feature: Major revamp of most conan commands, making command line arguments homogeneous. Much better development flow adapting to user layouts, with `install-folder`, `source-folder`, `build-folder`, `package-folder`.
- Feature: new `deploy()` method, useful for installing binaries from conan packages
- Feature: Implemented some **concurrency** support for the conan local cache. Parallel **`conan install`** and **`conan create`** for different configurations should be possible.
- Feature: options now allow patterns in command line: `-o *:myoption=myvalue` applies to all packages
- Feature: new `pc` generator that generates files from dependencies for `pkg-config`
- Feature: new `Meson` helper, similar to `CMake` for Meson build system. Works well with `pc` generator.
- Feature: Support for read-only cache with `CONAN_READ_ONLY_CACHE` environment variable
- Feature: new `visual_studio_multi` generator to load Debug/Release, 32/64 configs at once
- Feature: new `tools.which` helper to locate executables
- Feature: new **`conan --help`** layout
- Feature: allow to override compiler version in `vcvars_command`
- Feature: **`conan user`** interactive (and not exposed) password input for empty `-p` argument
- Feature: Support for `PacManTool` for `system_requirements()` for ArchLinux
- Feature: Define VS toolset in `CMake` constructor and from environment variable CONAN_CMAKE_TOOLSET
- Feature: **`conan create`** now accepts `werror` argument
- Feature: `AutoToolsBuildEnvironment` can use `CONAN_MAKE_PROGRAM` env-var to define make program
- Feature: added xcode9 for apple-clang 9.0, clang 5 to default settings.yml
- Feature: deactivation of `short_paths` in Windows 10 with Py3.6 and long path support is automatic
- Feature: show unzip progress by percentage, not by file (do not clutters output)
- Feature: do not use `sudo` for system requirements if already running as root
- Feature: `tools.download` able to use headers/auth
- Feature: conan does not longer generate bytecode from recipes (no more .pyc, and more efficient)
- Feature: add parallel argument to `build_sln_command` for VS
- Feature: Show warning if vs150comntools is an invalid path
- Feature: `tools.get()` now has arguments for hash checking
- Fix: upload pattern now accepts `Pkg/*`
- Fix: improved downloader, make more robust, better streaming
- Fix: `tools.patch` now support adding/removal of files
- Fix: The `default` profile is no longer taken as a base and merged with user profile. Use explicit `include(default)` instead.
- Fix: properly manage x86 as cross building with autotools

- Fix: `tools.unzip` removed unnecessary long-paths check in Windows
- Fix: `package_info()` is no longer executed at install for the consumer conanfile.py
- BugFix: source folder was not being correctly removed when recipe was updated
- BugFix: fixed `CMAKE_C_FLAGS_DEBUG` definition in `cmake` generator
- BugFix: `CMAKE_SYSTEM_NAME` is now Darwin for iOS, watchOS and tvOS
- BugFix: `xcode` generator fixed handling of compiler flags
- BugFix: pyinstaller hidden import that broke .deb installer
- BugFix: **`conan profile list`** when local files matched profile names

Note: Breaking changes

This is an important release towards stabilizing conan and moving out of beta. Some breaking changes have been done, but mostly to command line arguments, so they should be easy to fix. Package recipes or existing packages shouldn't break. Please **update**, it is very important to ease the transition of future stable releases. Do not hesitate to ask questions, or for help if you need it. This is a possibly not complete list of things to take into account:

- The command **`conan install`** doesn't accept `cwd` anymore, to change the directory where the generator files are written, use the **`--install-folder`** parameter.
- The command **`conan install`** doesn't accept **`--all`** anymore. Use **`conan download <ref>`** instead.
- The command **`conan build`** now requires the path to the `conanfile.py` (optional before)
- The command **`conan package`** not longer re-package a package in the local cache, now it only operates in a user local folder. The recommended way to re-package a package is using **`conan build`** and then **`conan export-pkg`**.
- Removed **`conan package_files`** in favor of a new command **`conan export-pkg`**. It requires a local recipe with a `package()` method.
- The command **`conan source`** no longer operates in the local cache. now it only operates in a user local folder. If you used **`conan source`** with a reference to workaround the concurrency, now it natively supported, you can remove the command call and trust concurrent install processes.
- The command **`conan imports`** doesn't accept `-d, --dest` anymore, use **`--imports-folder`** parameter instead.
- If you specify a profile in a conan command, like conan create or conan install the base profile *~/.conan/profiles/default* won't be applied. Use explicit `include` to keep the old behavior.

22.168 0.27.0 (20-September-2017)

- Feature: **`conan config install <url>`** new command. Will install remotes, profiles, settings, conan.conf and other files into the local conan installation. Perfect to synchronize configuration among teams
- Feature: improved traceback printing when errors are raised for more context. Configurable via env
- Feature: filtering out non existing directories in `cpp_info` (include, lib, etc), so some build systems don't complain about them.
- Feature: Added include directories to ResourceCompiler and to MIDL compiler in `visual_studio` generator
- Feature: new `visual_studio_legacy` generator for Visual Studio 2008

- Feature: show path where manifests are locally stored
- Feature: `replace_in_file` now raises error if replacement is not done (opt-out parameter)
- Feature: enabled in conan.conf `[proxies]` section `no_proxy=url1,url2` configuration (to skip proxying for those URLs), as well as `http=None` and `https=None` to explicitly disable them.
- Feature: new conanfile `self.in_local_cache` attribute for conditional logic to apply in user folders local commands
- Feature: `CONAN_USER_HOME_SHORT=None` can disable the usage of `short_paths` in Windows, for modern Windows that enable long paths at the system level
- Feature: `if "arm" in self.settings.arch` is now a valid check (without casting to str(self.settings.arch))
- Feature: added cwd`` argument to **`conan source`** local method.
- Fix: unzip crashed for 0 Bytes zip files
- Fix: `collect_libs` moved to the `tools` module
- Bugfix: fixed wrong regex in `deps_cpp_info` causing issues with dots and dashes in package names
- Development: Several internal refactorings (tools module, installer), testing (using VS2015 as default, removing VS 12 in testing). Conditional CI in travis for faster builds in developers, downgrading to CMake 3.7 in appveyor
- Deprecation: `dev_requires` have been removed (it was not documented, but accessible via the `requires(dev=True)` parameter. Superseded by `build_requires`.
- Deprecation: sources tgz files for exported sources no longer contain ".c_src" subfolder. Packages created with 0.27 will be incompatible with conan < 0.25

22.169 0.26.1 (05-September-2017)

- Feature: added apple-clang 9.0 to default settings.
- Fix: **`conan copy`** command now supports symlinks.
- Fix: fixed removal of "export_source" folder when files have no permissions
- Bugfix: fixed parsing of *conanbuildinfo.txt* with package names containing dots.

22.170 0.26.0 (31-August-2017)

- Feature: **`conan profile`** command has implemented `update`, `new`, `remove` subcommands, with detect``, to allow creation, edition and management of profiles.
- Feature: **`conan package_files`** command now can call recipe `package()` method if build_folder`` or source_folder`` arguments are defined
- Feature: graph loading algorithm improved to avoid repeating nodes. Results in much faster times for dense graphs, and avoids duplications of private requirements.
- Feature: authentication based on environment variables. Allows very long processes without tokens being expired.
- Feature: Definition of Visual Studio runtime setting `MD` or `MDd` is now automatic based on build type, not necessary to default in profile.
- Feature: Capturing `SystemExit` to return user error codes to the system with `sys.exit(code)`

- Feature: Added SKIP_RPATH argument to cmake `conan_basic_setup()` function
- Feature: Optimized uploads, now uploads will be skipped if there are no changes, irrespective of timestamp
- Feature: Automatic detection of VS 15-2017, via both a `vs150comntools` variable, and using `vswhere.exe`
- Feature: Added NO_OUTPUT_DIRS argument to cmake `conan_basic_setup()` function
- Feature: Add support for Chocolatey system package manager for Windows.
- Feature: Improved in conan user home and path storage configuration, better error checks.
- Feature: `export` command is now able to export recipes without name or version, specifying the full reference.
- Feature: Added new default settings, Arduino, gcc-7.2
- Feature: Add conan settings to cmake generated file
- Feature: new `tools.replace_prefix_in_pc_file()` function to help with .pc files.
- Feature: Adding support for system package tool `pkgutil` on Solaris
- Feature: **`conan remote update`** now allows **`--insert`** argument to change remote order
- Feature: Add `verbose` definition to `CMake` helper.
- Fix: **`conan package`** working locally failed if not specified build_folder
- Fix: Search when using wildcards for version like `Pkg/*@user/channel`
- Fix: Change current working directory to the conanfile.py one before loading it, so relative python imports or code work.
- Fix: `package_files` command now works with `short_paths` too.
- Fix: adding missing require of tested package in test_package/conanfile build() method
- Fix: path joining in `vcvars_command` for custom VS paths defined via env-vars
- Fix: better managing string escaping in CMake variables
- Fix: `ExecutablePath` assignment has been removed from the `visual_studio` generator.
- Fix: removing `export_source` folder containing exported code, fix issues with read-only files and keeps cache consistency better.
- Fix: Accept 100 return code from yum check-update
- Fix: importing *.so files from the **`conan new`** generated test templates
- Fix: progress bars display when download/uploads are not multipart (reported size 0)
- Bugfix: fixed wrong OSX `DYLD_LIBRARY_PATH` variable for virtual environments
- Bugfix: `FileCopier` had a bug that affected `self.copy()` commands, changing base reference directory.

22.171 0.25.1 (20-July-2017)

- Bugfix: Build requires are now applied correctly to test_package projects.
- Fix: Fixed search command to print an error when –table parameter is used without a reference.
- Fix: install() method of the CMake() helper, allows parallel building, change build folder and custom parameters.
- Fix: Controlled errors in migration, print warning if conan is not able to remove a package directory.

22.172 0.25.0 (19-July-2017)

Note: This release introduces a new layout for the local cache, with dedicated `export_source` folder to store the source code exported with `exports_sources` feature, which is much cleaner than the old `.c_src` subfolder. A migration is included to remove from the local cache packages with the old layout.

- Feature: new **`conan create`** command that supersedes *test_package* for creating and testing package. It works even without the test_package folder, and have improved management for user, channel. The test_package recipe no longer defines `requires`
- Feature: new **`conan get`** command that display (with syntax highlight) package recipes, and any other file from conan: recipes, conaninfo.txt, manifests, etc.
- Feature: new **`conan alias`** command that creates a special package recipe, that works like an **alias** or a **proxy** to other package, allowing easy definition and transparent management of "using the latest minor" and similar policies. Those special alias packages do not appear in the dependency graph.
- Feature: new **`conan search --table=file.html`** command that will output an html file with a graphical representation of available binaries
- Feature: created **default profile**, that replace the `[settings_default]` in **conan.conf** and augments it, allowing to define more things like env-vars, options, build_requires, etc.
- Feature: new `self.user_info` member that can be used in `package_info()` to define custom user variables, that will be translated to general purpose variables by generators.
- Feature: **`conan remove`** learned the **`--outdated`** argument, to remove those binary packages that are outdated from the recipe, both from local cache and remotes
- Feature: **`conan search`** learned the **`--outdated`** argument, to show only those binary packages that are outdated from the recipe, both from local cache and remotes
- Feature: Automatic management `CMAKE_TOOLCHAIN_FILE` in `CMake` helper for cross-building.
- Feature: created `conan_api`, a python API interface to conan functionality.
- Feature: new `cmake.install()` method of `CMake` helper.
- Feature: `short_paths` feature now applies also to `exports_sources`
- Feature: `SystemPackageTool` now supports **FreeBSD** system packages
- Feature: `build_requires` now manage options too, also default options in package recipes
- Feature: **`conan build`** learned new **`--package_folder`** argument, useful if the build system perform the packaging
- Feature: `CMake` helper now defines by default `CMAKE_INSTALL_PREFIX` pointing to the current package_folder, so `cmake.install()` can transparently execute the packaging.
- Feature: improved command UX with cwd`` arguments to allow define the current directory for the command
- Feature: improved `VisualStudioBuildEnvironment`
- Feature: transfers now show size (MB, KB) of download/uploaded files, and current status of transfer.
- Feature: **`conan new`** now has arguments to generate CI scripts for Gitlab CI.
- Feature: Added `MinRelSize` and `RelWithDebInfo` management in `CMake` helper.
- Fix: make `mkdir`, `rmdir`, `relative_dirs` available for import from **`conans`** module.
- Fix: improved detection of Visual Studio default under cygwin environment.

- Fix: `package_files` now allows symlinks
- Fix: Windows installer now includes conan_build_info tool.
- Fix: appending environment variables instead of overwriting them when they come from different origins: upstream dependencies and profiles.
- Fix: made opt-in the check of package integrity before uploads, it was taking too much time, and provide little value for most users.
- Fix: Package recipe linter removed some false positives
- Fix: default settings from conan.conf do not fail for constrained settings in recipes.
- Fix: Allowing to define package remote with **`conan remote add_ref`** before download/upload.
- Fix: removed duplicated BUILD_SHARED_LIBS in test_package
- Fix: add "rhel" to list of distros using yum.
- Bugfix: allowing relative paths in `exports` and `exports_sources` fields
- Bugfix: allow custom user generators with underscore

22.173 0.24.0 (15-June-2017)

- Feature: **`conan new`** new arguments to generate **Travis-CI** and **Appveyor** files for Continuous Integration
- Feature: Profile files with `include()` and variable declaration
- Feature: Added `RelWithDebInfo/MinRelSize` to cmake generators
- Feature: Improved linter, removing false positives due to dynamic conanfile attributes
- Feature: Added `tools.ftp_download()` function for FTP retrieval
- Feature: Managing symlinks between folders.
- Feature: **`conan remote add`** command learned new insert`` option to add remotes in specific order.
- Feature: support multi-config in the `SCons` generator
- Feature: support for gcc 7.1+ detection
- Feature: `tools` now are using global `requests` and `output` instances. Proxies will work for `tools.download()`
- Feature: json`` parameter added to **`conan info`** command to create a JSON with the `build_order`.
- Fix: update default repos, now pointing to Bintray.
- Fix: printing `outdated from recipe` also for remotes
- Fix: Fix required slash in `configure_dir` of `AutoToolsBuildEnvironment`
- Fix: command `new` with very short names, now errors earlier.
- Fix: better error detection for incorrect `Conanfile.py` letter case.
- Fix: Improved some cmake robustness using quotes to avoid cmake errors
- BugFix: Fixed incorrect firing of building due to build`` patterns error
- BugFix: Fixed bug with options incorrectly applied to `build_requires` and crashing
- Refactor: internal refactorings toward having a python api to conan functionality

22.174 0.23.1 (05-June-2017)

- BugFix: Fixed bug while packaging symlinked folders in build folder, and target not being packaged.
- Relaxed OSX requirement of pyopenssl to <18

22.175 0.23.0 (01-June-2017)

- Feature: new `build_requires` field and `build_requirements()` in package recipes
- Feature: improved commands (source, build, package, package_files) and workflows for local development of packages in user folders.
- Feature: implemented `no_copy_source` attribute in recipes to avoid the copy of source code from "source" to "build folder". Created new `self.source_folder`, `self.build_folder`, `self.package_folder` for recipes to use.
- Feature: improved `qmake` generator with multi-config support, resource directories
- Feature: improved exception capture and formatting for all recipe user methods exceptions
- Feature: new `tools.sha256()` method
- Feature: folder symlinks working now for packages and upload/download
- Feature: added `set_find_paths()` to `cmake-multi`, to set CMake FindXXX.cmake paths. This will work only for single-config build-systems.
- Feature: using environment variables for `configure()`, `requirements()` and `test()` methods
- Feature: added a `pylintrc` environment variable in `conan.conf` to define a PYLINTRC file with custom style definitions (like indents).
- Feature: fixed `vcvars` architecture setting
- Fix: Make `cacert.pem` folder use CONAN_USER_HOME if existing
- Fix: fixed `options=a=b` option definition
- Fix: `package_files` command allows force`` argument to overwrite existing instead of failing
- BugFix: Package names with underscore when parsing `conanbuildinfo.txt`

22.176 0.22.3 (03-May-2017)

- Fix: Fixed CMake generator (in targets mode) with linker/exe flags like –framework XXX containing spaces.

22.177 0.22.2 (20-April-2017)

- Fix: Fixed regression with usernames starting with non-alphabetical characters, introduced by 0.22.0

22.178 0.22.1 (18-April-2017)

- Fix: "-" symbol available again in usernames.
- Fix: Added `future` requirement to solve an error with pyinstaller generating the Windows installer.

22.179 0.22.0 (18-April-2017)

- Feature: `[build_requires]` can now be declared in `profiles` and apply them to build packages. Those requirements are only installed if the package is required to build from sources, and do not affect its package ID hash, and it is not necessary to define them in the package recipe. Ideal for testing libraries, cross compiling toolchains (like Android), development tools, etc.
- Feature: Much improved support for cross-building. Support for cross-building to **Android** provided, with toolchains installable via `build_requires`.
- Feature: New `package_files` command, that is able to create binary packages directly from user files, without needing to define `build()` or `package()` methods in the the recipes.
- Feature: command **`conan new`** with a new bare`` option that will create a minimal package recipe, usable with the `package_files` command.
- Feature: Improved `CMake` helper, with `test()` method, automatic setting of BUILD_SHARED_LIBS, better management of variables, support for parallel compilation in MSVC (via /MP)
- Feature: new `tools.msvc_build_command()` helper that both sets the Visual vcvars and calls Visual to build the solution. Also `vcvars_command` is improved to return non-empty string even if vcvars is set, for easier concatenation.
- Feature: Added package recipe linter, warning for potential errors and also about Python 3 incompatibilities when running from Python 2. Enabled by default can be opt-out.
- Feature: Improvements in HTML output of **`conan info --graph`**.
- Feature: allow custom path to bash, as configuration and environment variable.
- Fix: Not issuing an unused variable warning in CMake for the CONAN_EXPORTED variable
- Fix: added new `mips` architectures and latest compiler versions to default settings.yml
- Fix: Unified username allowed patterns to those used in package references.
- Fix: hardcoded vs15 version in tools.vcvars
- BugFix: Clean crash and improved error messages when manifests mismatch exists in conan upload.

22.180 0.21.2 (04-April-2017)

- Bugfix: virtualenv generator quoting environment variables in Windows.

22.181 0.21.1 (23-March-2017)

- BugFix: Fixed missing dependencies in `AutoToolsBuildEnvironment`
- BugFix: Escaping single quotes in html graph of **`conan info --graph=file.html`**.
- BugFix: Fixed loading of auth plugins in conan_server
- BugFix: Fixed `visual_studio` generator creating XML with dots.

22.182 0.21.0 (21-March-2017)

- Feature: **`conan info --graph`** or graph=file.html`` will generate a dependency graph representation in dot or html formats.
- Feature: Added better support and tests for Solaris Sparc.
- Feature: custom authenticators are now possible in **`conan_server`**` with plugins.
- Feature: extended **`conan info`** command with path information and filter by packages.
- Feature: enabled conditional binary packages removal with **`conan remove`** with query syntax
- Feature: enabled generation and validation of manifests from *test_package*.
- Feature: allowing `options` definitions in profiles
- Feature: new `RunEnvironment` helper, that makes easier to run binaries from dependent packages
- Feature: new `virtualrunenv` generator that activates environment variable for execution of binaries from installed packages, without requiring `imports` of shared libraries.
- Feature: adding new version modes for ABI compatibility definition in `package_id()`.
- Feature: Extended **`conan new`** command with new option for `exports_sources` example recipe.
- Feature: `CMake` helper defining parallel builds for gcc-like compilers via jN``, allowing user definition with environment variable and in conan.conf.
- Feature: **`conan profile`**` command now show profiles in alphabetical order.
- Feature: extended `visual_studio` generator with more information and binary paths for execution with DLLs paths.
- Feature: Allowing relative paths with $PROFILE_DIR place holder in `profiles`
- Fix: using only file checksums to decide for modified recipe in remote, for possible concurrent builds & uploads.
- Fix: Improved build`` modes management, with better checks and allowing multiple definitions and mixtures of conditions
- Fix: Replaced warning for non-matching OS to one message stating the cross-build
- Fix: local **`conan source`**` command (working in user folder) now properly executes the equivalent of `exports` functionality

- Fix: Setting command line arguments to cmake command as CMake flags, while using the TARGETS approach. Otherwise, arch flags like -m32 -m64 for gcc were not applied.
- BugFix: fixed **`conan imports`** destination folder issue.
- BugFix: Allowing environment variables with spaces
- BugFix: fix for CMake with targets usage of multiple flags.
- BugFix: Fixed crash of `cmake_multi` generator for "multi-config" packages.

22.183 0.20.3 (06-March-2017)

- Fix: Added opt-out for `CMAKE_SYSTEM_NAME` automatically added when cross-building, causing users providing their own cross-build to fail
- BugFix: Corrected usage of `CONAN_CFLAGS` instead of `CONAN_C_FLAGS` in cmake targets

22.184 0.20.2 (02-March-2017)

- Fix: Regression of `visual_studio``generator using ``%(ExecutablePath)` instead of `$(ExecutablePath)`
- Fix: Regression for build=outdated –build=Pkg`` install pattern

22.185 0.20.1 (01-March-2017)

- Fix: Disabled the use of cached settings and options from installed `conaninfo.txt`
- Fix: Revert the use of quotes in `cmake` generator for flags.
- Fix: Allow comments in artifacts.properties
- Fix: Added missing commit for CMake new helpers

22.186 0.20.0 (27-February-2017)

NOTE: It is important that if you upgrade to this version, all the clients connected to the same remote, should upgrade too. Packages created with conan>=0.20.0 might not be usable with conan older conan clients.

- Feature: Largely improved management of **environment variables**, declaration in `package_info()`, definition in profiles, in command line, per package, propagation to consumers.
- Feature: New build helpers `AutotoolsBuildEnvironment`, `VisualStudioBuildEnvironment`, which deprecate `ConfigureEnvironment`, with much better usage of environment variables
- Feature: New `virtualbuildenv` generator that will generate a composable environment with build information from installed dependencies.
- Feature: New `build_id()` recipe method that allows to define logic to build once, and package multiple times without building. E.g.: build once both debug and release artifacts, then package separately.

- Feature: **Multi-config packages**. Now packages can provide multi-configuration packages, like both debug/release artifacts in the same package, with `self.cpp_info.debug.libs = [...]` syntax. Not restricted to debug/release, can be used for other purposes.
- Feature: new **`conan config`** command to manage, edit, display `conan.conf` entries
- Feature: *Improvements* to `CMake` build helper, now it has `configure()` and `build()` methods for common operations.
- Feature: Improvements to `SystemPackageTool` with detection of installed packages, improved implementation, installation of multi-name packages.
- Feature: Unzip with `tools.unzip` maintaining permissions (Linux, OSX)
- Feature: **`conan info`** command now allows profiles too
- Feature: new tools `unix_path()`, `escape_windows_cmd()`, `run_in_windows_bash()`, useful for autotools projects in Win/MinGW/Msys
- Feature: new context manager `tools.chdir`, to temporarily change directory.
- Feature: CMake using `CMAKE_SYSTEM_NAME` for cross-compiling.
- Feature: Artifactory build-info extraction from traces
- Feature: Attach custom headers to artifacts uploads with an *artifacts.properties* file.
- Feature: allow and copy symlinks while **`conan export`**
- Fix: removing quotes in some cmake variables that were generating incorrect builds
- Fix: providing better error messages for non existing binaries, with links to the docs
- Fix: improved error messages if `tools.patch` failed
- Fix: adding `resdirs` to `cpp_info` propagated information, and cmake variables, for directories containing resources and other data.
- Fix: printing error messages if a build`` policy doesn't match any package
- Fix: managing VS2017 by `tools`. Still the manual definition of `vs150comntools` required.
- Bug fix: crashes when not supported characters were dumped to terminal by logger
- Bug fix: wrong executable path in Visual Studio generator

22.187 0.19.3 (27-February-2017)

- Fix: backward compatibility for new environment variables. New features to be introduced in 0.20 will produce that `conaninfo.txt` will not be correctly parsed, and then package would be "missing". This will happen for packages created with 0.20, and consumed with older than 0.19.3

NOTE: It is important that you upgrade at least to this version if you are using remotes with packages that might be created with latest conan releases (like conan.io).

22.188 0.19.2 (15-February-2017)

- Bug fix: Fixed bug with remotes behind proxies
- Bug fix: Fixed bug with `exports_sources` feature and nested folders

22.189 0.19.1 (02-February-2017)

- Bug fix: Fixed issue with **`conan copy`** followed by **`conan upload`** due to the new `exports_sources` feature.

22.190 0.19.0 (31-January-2017)

- Feature: `exports_sources` allows to snapshot sources (like `exports`) but retrieve them strictly when necessary, to build from sources. This can largely improve install times for package recipes containing sources
- Feature: new configurable **tracer** able to create structured logs of conan actions: commands, API calls, etc
- Feature: new logger for `self.run` actions, able to log information from builds and other commands to files, that can afterwards be packaged together with the binaries.
- Feature: support for **Solaris SunOS**
- Feature: `Version` helper improved with `patch, pre, build` capabilities to handle `1.3.4-alpha2+build1` versions
- Feature: compress level of tgz is now configurable via `CONAN_COMPRESSION_LEVEL` environment variable, default 9. Reducing it can lead to faster compression times, at the expense of slightly bigger archives
- Feature: Add **powershell** support for virtualenv generator in Windows
- Feature: Improved `system_requirements()` raising errors when failing, retrying if not successful, being able to execute in user space for local recipes
- Feature: new cmake helper macro `conan_target_link_libraries()`.
- Feature: new cmake `CONAN_EXPORTED` variable, can be used in CMakeLists.txt to differentiate building in the local conan cache as package and building in user space
- Fix: improving the caching of options from **`conan install`** in conaninfo.txt and precedence.
- Fix: conan definition of cmake output dirs has been disabled for `cmake_multi` generator
- Fix: `imports()` now uses environment variables at "conan install" (but not at "conan imports" yet)
- Fix: `conan_info()` method has been renamed to `package_id()`. Backward compatibility is maintained, but it is strongly encouraged to use the new name.
- Fix: `conan_find_libraries` now use the NO_CMAKE_FIND_ROOT_PATH parameter for avoiding issue while cross-compiling
- Fix: disallowing duplicate URLs in remotes, better error management
- Fix: improved error message for wildcard uploads not matching any package
- Fix: remove deprecated `platform.linux_distribution()`, using new "distro" package
- Bugfix: fixed management of `VerifySSL` parameter for remotes
- Bugfix: fixed misdetection of compiler version in conanbuildinfo.cmake for apple-clang

- Bugfix: fixed trailing slash in remotes URLs producing crashes
- Refactor: A big refactor has been do to `options`. Nested options are no longer supported, and `option.suboption` will be managed as a single string option.

This has been a huge release with contributors of 11 developers. Thanks very much to all of them!

22.191 0.18.1 (11-January-2017)

- Bug Fix: Handling of transitive private dependencies in modern cmake targets
- Bug Fix: Missing quotes in CMake macro for modern cmake targets
- Bug Fix: Handling LINK_FLAGS in cmake modern targets
- Bug Fix: Environment variables no propagating to test project with test_package command

22.192 0.18.0 (3-January-2017)

- Feature: uploads and downloads with **retries** on failures. This helps to avoid having to fully rebuild on CI when a network transfer fails
- Feature: added **SCons** generator
- Feature: support for **Python 3.6**, with several fixes. Added Python 3.6 to CI.
- Feature: show package dates in **`conan info`** command
- Feature: new `cmake_multi` generator for multi-configuration IDEs like Visual Studio and Xcode
- Feature: support for **Visual Studio 2017**, VS-15
- Feature: **FreeBSD** now passes test suite
- Feature: **`conan upload`** showing error messages or URL of remote
- Feature: **wildcard or pattern upload**. Useful to upload multiple packages to a remote.
- Feature: allow defining **settings as environment variables**. Useful for use cases like dockerized builds.
- Feature: improved help`` messages
- Feature: cmake helper tools to launch conan directly from cmake
- Added **code coverage** for code repository
- Fix: conan.io badges when containing dash
- Fix: manifests errors due to generated .pyc files
- Bug Fix: unicode error messages crashes
- Bug Fix: duplicated build of same binary package for private dependencies
- Bug Fix: duplicated requirement if using version-ranges and `requirements()` method.

22.193 0.17.2 (21-December-2016)

- Bug Fix: ConfigureEnvironment helper ignoring libcxx setting. #791

22.194 0.17.1 (15-December-2016)

- Bug Fix: conan install –all generating corrupted packages. Thanks to @yogeva
- Improved case sensitive folder management.
- Fix: appveyor links in README.

22.195 0.17.0 (13-December-2016)

- Feature: support for **modern cmake** with cmake `INTERFACE IMPORTED` targets defined per package
- Feature: support for more advanced queries in search.
- Feature: new `profile list|show` command, able to list or show details of profiles
- Feature: adding preliminary support for **FreeBSD**
- Feature: added new `description` field, to document package contents.
- Feature: generation of **imports manifest** and **`conan imports --undo`** functionality to remove imported files
- Feature: optional SSL certificate verification for remotes, to allow self signed certificates
- Feature: allowing custom paths in profiles, so profiles can be easily shared in teams, just inside the source repository or elsewhere.
- Feature: fields `user` and `channel` now available in conan recipes. That allows to declare requirements for the same user/channel as the current package.
- Feature: improved conan.io package web, adding description.
- Fix: allow to modify cmake generator in `CMake` helper class.
- Fix: added `strip` parameter to `tools.patch()` utility
- Fix: removed unused dependency to Boto
- Fix: wrong line endings in Windows for conan.conf
- Fix: proper automatic use of `txt` and `env` generators in *test_package*
- Bug fix: solved problem when uploading python packages that generated .pyc at execution
- Bug fix: crash when duplicate requires were declared in conanfile
- Bug fix: crash with existing imported files with symlinks
- Bug fix: options missing in “copy install command to clipboard” in web

22.196 0.16.1 (05-December-2016)

- Solved bug with *test_package* with arguments, like scopes.

22.197 0.16.0 (19-November-2016)

Upgrade: The build=outdated`` feature had a change in the hash computation, it might report outdated binaries from recipes. You can re-build the binaries or ignore it (if you haven't changed your recipes without re-generating binaries)

- Feature: **version ranges**. Conan now supports defining requirements with version range expressions like `Pkg/[>1.2,<1.9||1.0.1]@user/channel`. Check the *version ranges reference* for details
- Feature: decoupled `imports` from normal install. Now **`conan install --no-imports`** skips the imports section.
- Feature: new **`conan imports`** command that will execute the imports section without running install
- Feature: **overriding settings per package**. Now it is possible to specify individual settings for each package. This can be specified both in the command line and in `profiles`
- Feature: **environment variables** definition in the command line, global and per package. This allows to define specific environment variables as the compiler (CC, CXX) for a specific package. These environment variables can also be defined in `profiles`. Check *profiles reference*
- Feature: Now conan files copies handle **symlinks**, so files are not duplicated. This will save some space and improve download speed in some large packages. To enable it, use `self.copy(..., links=True)`
- Fix: Enabling correct use of **MSYS** in Windows, by using the Windows `C:/...` path instead of the MSYS ones
- Fix: Several fixes in **`conan search`**, both local and in remotes
- Fix: Manifests line endings and order fix, and hash computation fixed (it had wrong ordering)
- Fix: Removed http->https redirection in conan_server that produced some issues for SSL reversed proxies
- Fix: Taking into account "ANY" definition of settings and options
- Fix: Improved some error messages and failures to encode OS errors with unicode characters
- Update: added new arch `ppc64` to default settings
- Update: updated python-requests library version
- Fix: Using `generator()` instead of compiler to decide on cmake multi-configuration for Ninja+cl builds
- Improved and completed documentation

22.198 0.15.0 (08-November-2016)

Upgrade: If you were using the `short_paths` feature in Windows for packages with long paths, please reset your local cache. You could manually remove packages or just run **`conan remove "*"`**

- Feature: New build=outdated`` functionality, that allows to build the binary packages for those dependencies whose recipe has been changed, or if the binary is not existing. Each binary package stores a hash of the recipe to know if they have to be regenerated (are outdated). This information is also provided in the **`conan search <ref>`** command. Useful for package creators and CI.
- Feature: Extended the `short_paths` feature for Windows path limit to the `package` folder, so package with very long paths, typically in headers in nested folder hierarchies are supported.

- Feature: New `tool.build_sln_command()` helper to `build()` Microsoft Visual Studio solution (.sln) projects
- Feature: Extended the `source` and `package` command, so together with `build` they can be fully executed in a user folder, as a convenience for package creation and testing.
- Feature: Extending the scope of `tools.pythonpath` to work in local commands too
- Improved the parsing of `profiles` and better error messages
- Not adding `-s` compiler flag for clang, as it doesn't use it.
- Automatic generation of *conanenv.txt* in local cache, warnings if using local commands and no `conanbuildinfo.txt` and no *conanenv.txt* are present to cache the information form install
- Fix: Fixed bug when using empty initial requirements (`requires = ""`)
- Fix: Added `glob` hidden import to pyinstaller
- Fix: Fixed minor bugs with `short_paths` as local search not listing packages
- Fix: Fixed problem with virtual envs in Windows with paths separator (using / instead of)
- Fix: Fixed parsing of conanbuildinfo.txt, so the root folder for each dependency is available in local commands too
- Fix: Fixed bug in *test_package* with the test project using the `requirements()` method.

22.199 0.14.1 (20-October-2016)

- Fixed bug with *short_paths* feature in windows.
- Improved error messages for non-valid *profile* test files.
- Remove downloaded tgz package files from remotes after decompress them.
- Fixes bug with *install –all* and short_paths

22.200 0.14.0 (20-October-2016)

- Feature: Added profiles, as user predefined settings and environment variables (as CC and CXX for compiler paths). They are stored in files in the conan cache, so they can be easily edited, added, and shared. Use them with **`conan install --profile=name`**
- Feature: `short_paths` feature for Windows now also handle long paths for the final package, in case that a user library has a very long final name, with nested subfolders.
- Feature: Added `tools.cpu_count()` as a helper to retrieve the number of cores, so it can be used in concurrent builds
- Feature: Detects cycles in the dependency graph, and raise error instead of exhausting recursion limits
- Feature: Conan learned the werror`` option that will raise error and stop installation under some cases treated as warnings otherwise: Duplicated dependencies, or dependencies conflicts
- Feature: New `env` generator that generates a text file with the environment variables defined by dependencies, so it can be stored. Such file is parsed by **`conan build`** to be able to use such environment variables for `self.deps_env_info` too, in the same way it uses the `txt` generator to load variables for `self.deps_cpp_info`.
- Fix: Do not print progress bars when output is a file
- Fix: Improved the local conan search, using options too in the query **`conan search -q option=value`**

- Fix: Boto dependency updated to 2.43.0 (necessary for ArchLinux)
- Fix: Simplified the **conan package** command, removing unused and confusing options, and more informative messages about errors and utility of this command.
- Fix: More fixes and improvements on `ConfigureEnvironment`, mainly for Windows
- Fix: Conan now does not generate a `conanbuildinfo.txt` file when doing **conan install <PkgRef>**.
- Bug fix: Files of a package recipe are "touched" to update their timestamps to current time when retrieved, otherwise some build systems as Ninja can have problems with them.
- Bug fix: `qmake` generator now uses quotes to handle paths with spaces
- Bug fix: Fixed `OSInfo` to return the short distro name instead of the long one.
- Bug fix: fixed transitivity of `private` dependencies

22.201 0.13.3 (13-October-2016)

This minor solves some problems with `ConfigureEnvironment`, mainly for Windows, but also fixes other things:

- Fixed concatenation problems in Windows for several environment variables. Fixed problems with path with spaces
- A batch file is created in Windows to be called, as `if defined` structures doesn't seem to work in the command line.
- The `vcvars_command` from `tools` now checks the Visual Studio environment variable, if it is already set, it will check it with the current project settings, throwing an error if not matching, returning an empty command if matches.
- Added a `compile_flags` property to `ConfigureEnvironment`, to be passed in the command line to the compiler, but not as environment variables
- Added `defines` to environment for nix systems, it was not being handled before
- Added new tests, compiling simple projects and diamond dependencies with cmake, cl (msvc), gcc (gcc in linux, mingw in win) and clang (OSX), for a better coverage of the `ConfigureEnvironment` functionality.
- Fixed wrong `CPP_INCLUDE_PATH`, it is now `CPLUS_INCLUDE_PATH`

22.202 0.13.0 (03-October-2016)

IMPORTANT UPGRADE ISSUE: There was a small error in the computation of binary packages IDs, that has been addressed by conan 0.13. It affects to third level (and higher) binary packages, i.e. A and B in A->B->C->D, which binaries **must** be regenerated for the new hashes. If you don't plan to provide support for older conan releases (<=0.12), which would be reasonable, you should remove all binaries first (**conan remove -p**, works both locally and remotely), then re-build your binaries.

Features:

- Streaming from/to disk for all uploads/downloads. Previously, this was done for memory, but conan started to have issues for huge packages (>many hundreds MBs), that sometimes could be alleviated using Python 64 bits distros. This issues should be alleviated now
- New security system that allows capturing and checking the package recipes and binaries manifests into user folders (project or any other folder). That ensures that packages cannot be replaced, hacked, forged, changed or wrongly edited, either locally or in any remote server, without notice.

- Possible to handle and reuse python code in recipes. Actually, conan can be used as a package manager for python, by adding the package path to `env_info.PYTHONPATH`. Useful if you want to reuse common python code between different package recipes.
- Avoiding re-compress the tgz for packages after uploads if it didn't change.
- New command **`conan source`** that executes the `source()` method of a given conanfile. Very useful for CI, if desired to run in parallel the construction of different binaries.
- New propagation of `cpp_info`, so it now allows for capturing binary packages libraries with new `collect_libs()` helper, and access to created binaries to compute the `package_info()` in general.
- Command *test_package* now allows the update`` option, to automatically update dependencies.
- Added new architectures for `ppc64le` and detection for `AArch64`
- New methods for defining requires effect over binary packages ID (hash) in `conan_info()`
- Many bugs fixes: error in `tools.download` with python 3, restore correct prompt in virtualenvs, bug if removing an option in `config_options()`, setup.py bug…

This release has contributions from @tru, @raulbocanegra, @tivek, @mathieu, and the feedback of many other conan users, thanks very much to all of them!

22.203 0.12.0 (13-September-2016)

- Major changes to **search** api and commands. Decoupled the search of package recipes, from the search of binary packages.
- Fixed bug that didn't allow to `export` or `upload` packages with settings restrictions if the restrictions didn't match the host settings
- Allowing disabling color output with `CONAN_COLOR_DISPLAY=0` environment variable, or to configure color schema for light console backgrounds with `CONAN_COLOR_DARK=1` environment variable
- Imports can use absolute paths, and files copied from local conan cache to those paths will not be removed when **`conan install`**. Can be used as a way to install machine-wise things (outside conan local cache)
- More robust handling of failing transfers (network disconnect), and inconsistent status after such
- Large internal refactor for storage managers. Improved implementations and decoupling between server and client
- Fixed slow **`conan remove`** for caches with many packages due to slow deletion of empty folders
- Always allowing explicit options scopes, `-o Package:option=value` as well as the implicit `-o option=value` for current `Package`, for consistency
- Fixed some bugs in client-server auth process.
- Allow to extract `.tar` files in `tools.unzip()`
- Some helpers for `conan_info()`, as `self.info.requires.clear()` and removal of settings and options

22.204 0.11.1 (31-August-2016)

- New error reporting for failures in conanfiles, including line number and offending line, much easier for package creators
- Removed message requesting to create an account in **conan.io** for other remotes
- Removed localhost:9300 remote that was added by default mostly for demo purposes. Clarified in docs.
- Fixed usernames case-sensitivity in conan_server, due to ConfigParser it was forcing lowercase
- Handling unicode characters in remote responses, fixed crash
- Added new compilers gcc 6.2, clang 8.0 to the default `settings.yml`
- Bumped cryptography, boto and other conan dependencies, mostly for ArchLinux compatibility and new OSX security changes

22.205 0.11.0 (3-August-2016)

- New solution for the path length limit in Windows, more robust and complete. Package conanfile.py just have to declare an attribute `short_paths=True` and everything will be managed. The old approach is deprecated and totally removed, so no shorts_paths.conf file is necessary. It should fix also the issues with uploads/retrievals.
- New `virtualenv` generator that generates `activate` and `deactivate` scripts that set environment variables in the current shell. It is very useful, for example to install tools (like CMake, MinGW) with conan packages, so multiple versions can be installed in the same machine, and switch between them just by activating such virtual environments. Packages for MinGW and CMake are already available as a demo
- ConfigureEnvironment takes into account environment variables, defined in packages in new `env_info`, which is similar to `cpp_info` but for environment information (like paths).
- New per-package `build_policy`, which can be set to `always` or `missing`, so it is not necessary to create packages or specify the build`` parameter in command line. Useful for example in header only libraries or to create packages that always get the latest code from a branch in a github repository.
- Command **`conan test_package`** now executes by default a **`conan export`** with smarter package reference deduction. It is introduced as opt-out behavior.
- Conan :command`export` command avoids copying *test_package/build* temporary files in case of `export=*`
- Now, `package_info()` allows absolute paths in `includedir`, `libdirs` and `bindirs`, so wrapper packages can be defined that use system or manually installed libraries.
- LDFLAGS in `ConfigureEnvironment` management of OSX frameworks.
- Options allow the `ANY` value, so such option would accept any value. For example a commit of a git repository, useful to create packages that can build any specific commit of a git repo.
- Added gcc 5.4 to the default settings, as well as MinGW options (Exceptions, threads…)
- Command **`conan info`** learned a new option to output the packages from a project dependency tree that should be rebuilt in case of a modification of a certain package. It outputs a machine readable **ordered** list of packages to be built in that order. Useful for CI systems.
- Better management of incomplete, dirty or failed `source` directories (e.g. in case of a user interrupting with Ctrl+C a git clone inside the `source()` method.
- Added tools for easier detection of different OS versions and distributions, as well as command wrappers to install system packages (apt, yum). They use `sudo` via a new environment variable CONAN_SYSREQUIRES_SUDO, so using sudo is opt-in/out, for users with different sudo needs. Useful for `system_requirements()`

- Deprecated the `config()` method (still works, for backwards compatibility), but has been replaced by a `config_options()` to modify options based on settings, and a `configure()` method for most use cases. This removes a nasty behavior of having the `config()` method called twice with side effects.
- Now, running a **`conan install MyLib/0.1@user/channel`** to directly install packages without any consuming project, is also able to generate files with the `-g` option. Useful for installing tool packages (MinGW, CMake) and generate `virtualenvs`.
- Many small fixes and improvements: detect compiler bug in Py3, search was crashing for remotes, conan new failed if the package name had a dash, etc.
- Improved some internal duplications of code, refactored many tests.

This has been a big release. Practically 100% of the released features are thanks to active users feedback and contributions. Thanks very much again to all of them!

22.206 0.10.0 (29-June-2016)

- **conan new** command, that creates conan package conanfile.py templates, with a *test_package* package test (-t option), also for header only packages (-i option)
- Definition of **scopes**. There is a default **dev** scope for the user project, but any other scope (test, profile...) can be defined and used in packages. They can be used to fire extra processes (as running tests), but they do not affect the package binaries, and are not included in the package IDs (hash).
- Definition of **dev_requires**. Those are requirements that are only retrieved when the package is in **dev** scope, otherwise they are not. They do not affect the binary packages. Typical use cases would be test libraries or build scripts.
- Allow **shorter paths** for specific packages, which can be necessary to build packages with very long path names (e.g. Qt) in Windows.
- Support for bzip2 and gzip decompression in `tools`
- Added `package_folder` attribute to conanfile, so the `package()` method can for example call `cmake install` to create the package.
- Added `CONAN_CMAKE_GENERATOR` environment variable that allows to override the `CMake` default generator. That can be useful to build with Ninja instead of the default Unix Makefiles
- Improved `ConfigureEnvironment` with include paths in CFLAGS and CPPFLAGS, and fixed bug.
- New **`conan user --clean`** option, to completely remove all user data for all remotes.
- Allowed to raise `Exceptions` in `config()` method, so it is easier for package creators to raise under non-supported configurations
- Fixed many small bugs and other small improvements

As always, thanks very much to all contributors and users providing feedback.

22.207 0.9.2 (11-May-2016)

- **Fixed download bug** that made it specially slow to download, even crash. Thanks to github @melmdk for fixing it.
- **Fixed cmake check of CLang**, it was being skipped
- **Improved performance**. Check for updates has been removed from install, made it opt-in in `conan info` command, as it was very slow, seriously affecting performance of large projects.
- Improved internal representation of graph, also improves performance for large projects.
- Fixed bug in `conan install --update`.

22.208 0.9 (3-May-2016)

- **Python 3** "experimental" support. Now the main conan codebase is Python 2 and 3 compatible. Python 2 still the reference platform, Python 3 stable support in next releases.
- Create and share your **own custom generators for any build system or tool**. With "generator packages", you can write a generator just as any other package, upload it, modify and version it, etc. Require them by reference, as any other package, and pull it into your projects dynamically.
- **Premake4** initial experimental support via a generator package.
- Very large **re-write of the documentation**. New "creating packages" sections with in-source and out-source explicit examples. Please read it! :)
- Improved `conan test`. Renamed `test` to *test_package* both for the command and the folder, but backwards compatibility remains. Custom folder name also possible. **Adapted test layout** might require minor changes to your package test, automatic warnings added for your convenience.
- Upgraded pyinstaller to generate binary OS installers from 2.X to 3.1
- `conan search` now has command line options:, less verbose, verbose, extra verbose
- Added variable with full list of dependencies in conanbuildinfo.cmake
- Several minor bugfixes (check github issues)
- Improved `conan user` to manage user login to multiple remotes

22.209 0.8.4 (28-Mar-2016)

- Fixed linker problems with the new apple-clang 7.3 due to libraries with no timestamp set.
- Added apple-clang 7.3 to default settings
- Fixed default libcxx for apple-clang in auto detection of base conan.conf

22.210 0.8 (15-Mar-2016)

- New **conan remote** command to manage remotes. Redesigned remotes architecture, now allows to work with several remotes in a more consistent, powerful and "git-like" way. New remotes registry keeps track of the remote of every installed package, and this information is shown in **conan info** command too. Also, it keeps different user logins for different remotes, to improve support in corporate environments running in-house servers.
- New **update** functionality. Now it is possible to **conan install --update** to update packages that became obsolete because new ones were uploaded to the corresponding remote. Conan commands as install and info show information about the status of the local packages compared with the remote ones. In this way, using latest versions during development is much more natural.
- Added new **compiler.libcxx** setting in order to support the different c++ standard libraries. It can take libstdc++, libstdc++11 or libc++ values to take into account different standard libraries for modern gcc and clang compilers. It is also possible to remove not needed settings, like this one in pure C projects, with the new syntax: `del self.settings.compiler.libcxx`
- Conan **virtual environment**: Define a custom conan directory with **CONAN_USER_HOME** env variable, and have a per project or per workspace storage for your dependencies. So you can isolate your dependencies and even bundle them within your project, by just setting the CONAN_USER_HOME variable to your `<project>/deps` folder, for example. This also improves support for continuous integration CI systems, in which many builds from different users could be run in parallel.
- Better conanfile download method. More stable and now checks (opt-out) the **ssl certificates**.
- Lots of improvements: Increased library name length limit, Improved and cleaner output messages.
- Fixed several minor bugs: removing empty folders, case sensitive exports, arm settings detection.
- Introduced the concept of **"package recipe"** that refers to conanfile.py and exported files.
- Improved settings display in web, with new "copy install command to clipboard" to assist in installing packages discovered in web.
- The macOS installer, problematic with latest macOS releases, has been deprecated in favor of homebrew and pip install procedures.

22.211 0.7 (5-Feb-2016)

- Custom conanfile names are allowed for developing. With file`` option you can define the file you want to use, allowing for `.conaninfo.txt` or having multiple `conanfile_dev.py`, `conanfile_test.py` besides the standard `conanfile.py` which is used for sharing the package. Inheritance is allowed, e.g. `conanfile_dev.py` might extend/inherit from `conanfile.py`.
- New **conan copy** command that can be used to copy/rename packages, promote them between channels, forking other users packages.
- New all`` and package`` options for **conan install** that allows to download one, several, or all package configurations for a given reference.
- Added `patch()` tool to easily patch sources if necessary.
- New **qmake** and **qbs** generators
- Upload of conanfile **exported** files is also **tgz'd**, allowing fast upload/downloads of full sources if desired, avoiding retrieval of sources from externals sources.
- **conan info** command improved showing info of current project too
- Output of `run()` can be redirected to buffer string for processing, or even removed.

- Added **proxy** configuration to conan.conf for users behinds proxies.
- Large improvements in commands output, prefixed with package reference, and much clear.
- Updated settings for more versions of gcc and new arm architectures
- Treat dependencies includes as SYSTEM in cmake, so no warnings are raised
- Deleting source folder after **`conan export`** so no manual removal is needed
- Normalizing to CRLF generated user files in Win
- Better detection and checks for compilers as VS, apple-clang
- Fixed CMAKE_SHARED_LINKER_FLAGS typo in cmake files
- Large internal refactor in generators

22.212 0.6 (11-Jan-2016)

- New cmake variables in cmake generator to make FindPackage work better thanks to the underlaying FindLibrary. Now many FindXXX.cmake work "as-is" and the package creator does not have to create a custom override, and consumers can use packages transparently with the originals FindXXX.cmakes
- New "conan info" command that shows the full dependency graph and details (license, author, url, dependants, dependencies) for each dependency.
- New environment helper with a ConfigureEnvironment class, that is able to translate conan information to autotools configure environment definition
- Relative importing from conanfiles now is possible. So if you have common functionality between different packages, you can reuse those python files by importing them from the conanfile.py. Note that export="..." might be necessary, as packages as to be self-contained.
- Added YouCompleteMe generator for vim auto-completion of dependencies.
- New "conanfile_directory" property that points to the file in which the conanfile.py is located. This helps if using the conanfile.py "build" method to build your own project as a project, not a package, to be able to use any workflow, out-of-source builds, etc.
- Many edits and improvements in help, docs, output messages for many commands.
- All cmake syntax in modern lowercase
- Fixed several minor bugs: gcc detection failure when gcc not installed, missing import, copying source->build failing when symlinks

22.213 0.5 (18-Dec-2015)

- New cmake functionality allows package creators to provide cmake finders, so that package consumers can use their CMakeLists.txt with typical FindXXX.cmake files, without any change to them. CMake CONAN_CMAKE_MODULES_PATH added, so that package creators can provide any additional cmake scripts for consumers.
- Now it is possible to generate out-of-source and multiple configuration installations for the same project, so you can switch between them without having to **`conan install`** again. Check *the new workflows*
- New qmake generator (thanks @dragly)

- Improved removal/deletion of folders with shutil.rmtree, so **`conan remove`** commands and other processes requiring deletion of folders do not fail due to permissions and require manual deletion. This is an improvement, especially in Win.
- Created `pip` package, so conan can be installed via: `pip install conan`
- Released `pyinstaller` code for the creation of binaries from conan python source code. Distros package creators can create packages for the conan apps easily from those binaries.
- Added md5, sha1, sha256 helpers in `tools`, so external downloads from `conanfile.py` files `source()` can be checked.
- Added latest gcc versions to default `settings.yml`
- Added CI support for conan development: travis-ci, appveyor
- Improved human-readability for download progress, help messages.
- Minor bug fixes

CHAPTER

TWENTYTHREE

CONAN MIGRATION GUIDE TO 2.0

Conan 2.0-alpha is already released, you can install the latest Conan Alpha version from PyPI doing:

```
$ pip install conan --pre
```

If you want to migrate to 2.0, there are several things you will need to change:

- The **recipes** have to be updated to be compatible with Conan 2.0. There are 2.0 features ported to Conan 1.X so you can get a compatible recipe with 2.0 using Conan 1.X.
- The **conan commands** have also changed, but there are no "compatible" commands introduced in Conan 1.X. We will review the more relevant changes.
- **General changes** not related to the recipes nor the Conan commands, "build profiles", lowercase references... etc.

23.1 Migrating the recipes

We introduced changes to `Conan 1.X` versions so you can start migrating your recipes to do a smooth transition to `Conan 2.0`.

23.1.1 Python import statements

- All the imports from the `conans` package have to be replaced. The Conan 2.0 ones are in the `conan` package. Note the plural.
- The "tools" functions are now organized in different packages, you can check the *complete reference here*.

Listing 1: **From:**

```
from conans import ConanFile, tools
```

Listing 2: **To:**

```
from conan import ConanFile
from conan.tools.files import save, load
from conan.tools.gnu import AutotoolsToolchain, AutotoolsDeps
from conan.tools.microsoft import unix_path, VCVars, is_msvc
from conan.errors import ConanInvalidConfiguration
from conan.errors import ConanException
...
```

23.1.2 Requirements

- Use `self.test_requires()` to define test requirements instead of the legacy `self.build_requires(..., force_host_context)`.
- Use `self.tool_requires()` to define the legacy build_requires.

Listing 3: **From:**

```
from conans import ConanFile

class Pkg(Conanfile):

      ...

      def build_requirements(self):
          self.build_requires("nasm/2.15.05")
          self.build_requires("gtest/0.1", force_host_context=True)
```

Listing 4: **To:**

```
from conan import ConanFile

class Pkg(Conanfile):

      ...

      def build_requirements(self):
          self.tool_requires("nasm/2.15.05")
          self.test_requires("gtest/0.1")
```

23.1.3 Settings

- Do not use dictionary expressions in your recipe `settings` definition (like `settings = {"os": ["Windows", "Linux"]}`. This way of limiting supported configurations by one recipe will be removed. Use the `validate()` method instead to raise `ConanInvalidConfiguration` if strictly necessary to fail fast for unsupported configurations.

```
from conan import ConanFile

class Pkg(Conanfile):

      settings = "os", "arch", "compiler"

      ...

      def validate(self):
          if self.settings.os == "Macos":
              raise ConanInvalidConfiguration("Macos not supported")
```

- In Conan 2, removing a setting, for example, `del self.settings.compiler.libcxx` in the `configure()` method, will raise an exception if the setting doesn't exist. It has to be protected with try/except:

```
def configure(self):
    try:
       # In windows, with msvc, the compiler.libcxx doesn't exist, so it will raise.
       del self.settings.compiler.libcxx
    except Exception:
       pass
```

23.1.4 Options

The definition of the `default_options` attribute has changed when referring to a dependency. It is related to the *unified patterns in the command line*.

Listing 5: **From:**

```
from conans import ConanFile

class Pkg(Conanfile):
    default_options = {"pkg:some_option": "value"}
```

Listing 6: **To:**

```
from conan import ConanFile

class Pkg(Conanfile):
    # "pkg/*:some_option" or ""pkg/1.0:some_option" or "pkg*:some_option" would be valid
    default_options = {"pkg/*:some_option": "value"}
```

23.1.5 The layout() method

The layout method is not mandatory but very recommended to:

- Give better support for `editable` packages.
- Work with local commands, `conan install` + `conan source` + `conan build`.

If your recipe is using CMake, you might want to use the `cmake_layout(self)`:

```
from conan import ConanFile
from conan.tools.cmake import cmake_layout

class Pkg(Conanfile):

    def layout(self):
        cmake_layout(self)
```

A typical anti-pattern in the recipes that can be solved with a `layout()` declaration would be:

Listing 7: **From:**

```
from conans import ConanFile, tools

class Pkg(Conanfile):
```

(continues on next page)

(continued from previous page)

```
    @property
    def _source_subfolder(self):
        return "source_subfolder"

    def source(self):
        tools.get(**self.conan_data["sources"][self.version],
                  destination=self._source_subfolder, strip_root=True)
```

Listing 8: **To:**

```
from conan import ConanFile
from conan.tools.layout import basic_layout
from conan.tools.files import get

class Pkg(Conanfile):

    @property
    def layout(self):
        basic_layout(self, src_folder="source")

    def source(self):
        get(self, **self.conan_data["sources"][self.version], strip_root=True)
```

Declaring the layout, the variables `self.source_folder`, `self.build_folder` will point to the correct folder, both in the cache or locally when using local methods, it is always recommended to use these when performing disk operations (read, write, copy, etc).

If you are using `editables`, the external template files are going to be removed. Use the `layout()` method definition instead.

Read more about the *layout feature* and the *reference of the layout() method*.

Adjusting the cpp_info objects

You can adjust the cpp_info in the `layout` method too, not only for a package in the cache, that was typically done in the `package_info()` method using the `self.cpp_info`, but for editable packages (to reuse a conan package that is being developed in a local directory):

```
def layout(self):

    # This will be automatically copied to self.cpp_info
    # This information is relative to the self.package_folder
    self.cpp.package.includedirs.append("other_includes")

    # This information is relative to the self.build_folder
    self.cpp.build.libdirs = ["."]
    self.cpp.build.bindirs = ["bin"]

    # This information is relative to the self.source_folder
    self.cpp.source.includedirs = ["."]
```

23.1.6 The scm attribute

The `scm` attribute won't exist in Conan 2.0. You have to start using the `export()` and `source()` methods to mimic the same behavior:

- The `export()` method is responsible for capturing the "coordinates" of the current URL and commit. The new `conan.tools.scm.Git` can be used for this (do not use the legacy `Git` helper but this one)
- The `export()` method, after capturing the coordinates, can store them in the `conandata.yml` using the `update_conandata()` helper function
- The `source()` method can use the information in `self.conan_data` coming from exported `conandata.yml` file to do a clone and checkout of the matching code. The new `conan.tools.scm.Git` can be used for this purpose.

Listing 9: **From:**

```
from conans import ConanFile, tools

class Pkg(Conanfile):

    scm = {
         "type": "git",
         "url": "auto",
         "revision": "auto",
    }
```

Listing 10: **To:**

```
from conan import ConanFile
from conan.tools.scm import Git
from conan.tools.files import load, update_conandata

class Pkg(Conanfile):

   def export(self):
       git = Git(self, self.recipe_folder)
       scm_url, scm_commit = git.get_url_and_commit()
       update_conandata(self, {"sources": {"commit": scm_commit, "url": scm_url}})

   def source(self):
       git = Git(self)
       sources = self.conan_data["sources"]
       git.clone(url=sources["url"], target=".")
       git.checkout(commit=sources["commit"])
```

Please **check the full example** on the *conan.tools.scm.Git section*.

23.1.7 The generate() method

This is a key method to understand how Conan 2.0 works. This method is called during the "install" process, before calling the "build()" method. All information needed to build the current package has to be calculated and written in disk (in the `self.generators_folder`) by the `generate()` method. That information is about:

- The dependencies of the recipe: Typically called "generators".
- The configuration (settings, options...): Typically called "toolchains".

The goal of the `generate()` method is to have a very simple build process (the more dummy, the better), calling the build system passing some files or arguments and activating some environment launchers.

This improves a lot the local development, a simple `conan install` will generate everything we need to build our project in the IDE or just call the build system. This example is using the `CMake` integration, but if you use other build systems, even a custom one, remember you should generate everything needed in the `generate()` method:

```
from conan import ConanFile
from conan.tools.cmake import CMakeToolchain, CMakeDeps, CMake, cmake_layout


class Pkg(ConanFile):
    ...
    requires = "foo/1.0", "bar/1.0"

    def layout(self):
        cmake_layout(self)

    def generate(self):
        # This generates "conan_toolchain.cmake" in self.generators_folder
        tc = CMakeToolchain(self)
        tc.variables["MYVAR"] = "1"
        tc.preprocessor_definitions["MYDEFINE"] = "2"
        tc.generate()

        # This generates "foo-config.cmake" and "bar-config.cmake" in self.generators_
↪folder
        deps = CMakeDeps(self)
        deps.generate()

    ...
```

If we are using that recipe for our project we can build it by typing:

```
$ conan install .
# This will generate the config files from the dependencies and the toolchain
$ cmake . -DCMAKE_TOOLCHAIN_FILE=./cmake-build-release/conan/conan_toolchain.cmake
$ cmake --build .
```

You can check all the generators and toolchains for different build systems in the *tools reference page*.

It is also very important to know that every access to the information from the `dependencies` must be done in the `generate()` method using the self.dependencies access. Do not use `self.deps_cpp_info`, `self.deps_env_info` or `self.deps_user_info`, these have been removed in 2.0.

Note: If you don't need to customize anything in a generator you can specify it in the `generators` attribute and skip

using the `generate()` method for that:

```
from conan import ConanFile
from conan.tools.cmake import CMake, cmake_layout


class Pkg(ConanFile):
    ...
    requires = "foo/1.0", "bar/1.0"
    generators = "CMakeToolchain", "CMakeDeps"
    ...
```

23.1.8 The build() method

There is nothing special in the `build()` method, just emphasize the concept of `dummy build` explained before.

23.1.9 The package() method

The `self.copy` has been replaced by the explicit tool *copy*.

Listing 11: **From:**

```
def package(self):
    ...
    self.copy("*.h", dst="include", src="src")
    self.copy("*.lib", dst="lib", keep_path=False)
    self.copy("*.dll", dst="bin", keep_path=False)
```

Listing 12: **To:**

```
from conan.tools.files import copy

def package(self):
    ...
    copy(self, "*.h", self.source_folder, join(self.package_folder, "include"), keep_
↪path=False)
    copy(self, "*.lib", self.build_folder, join(self.package_folder, "lib"), keep_
↪path=False)
    copy(self, "*.dll", self.build_folder, join(self.package_folder, "bin"), keep_
↪path=False)
```

23.1.10 The package_info() method

Removed cpp_info defaults in components

There are some defaults in the general `self.cpp_info` object:

```
self.cpp_info.includedirs => ["include"]
self.cpp_info.libdirs => ["lib"]
self.cpp_info.resdirs => ["res"]
self.cpp_info.bindirs => ["bin"]
self.cpp_info.builddirs => []
self.cpp_info.frameworkdirs => ["Frameworks"]
```

If you declare components, you need to explicitly specify these directories, because by default are empty:

```
def cpp_info(self):
    self.cpp_info.components["mycomponent"].includedirs = ["my_include"]
    self.cpp_info.components["myothercomponent"].bindirs = ["myother_bin"]
```

Note: Remember that now is possible to declare the `cpp_info` in the *layout() method* using the `self.cpp.package` instead of using `self.cpp_info` in the `package_info()`.

Removed self.user_info

Replaced by the `self.conf_info` object, much more versatile than the previous `self.user_info`. Check the complete usage of *self.conf_info*.

Example:

Listing 13: **From:**

```
import os
from conans import ConanFile

class Pkg(ConanFile):
    name = "pkg"
```

(continues on next page)

(continued from previous page)

```
    version = "1.0"

    def package_info(self):
        self.user_info.FOO = "bar"
```

Listing 14: **To:**

```
import os
from conans import ConanFile

class Pkg(ConanFile):
    name = "pkg"
    version = "1.0"

    def package_info(self):
        self.conf_info.define("user.myconf:foo", "bar")
```

In a consumer recipe:

```
import os
from conans import ConanFile

class Pkg(ConanFile):
    requires = "pkg/1.0"

    def generate(self):
       my_value = self.dependencies[pkg].conf_info.get("user.myconf:foo")
       ...
```

Note: The consumer recipes will have a `self.conf` object available with the aggregated configuration from all the recipes in the `build` context:

```
from conan import ConanFile

class Pkg(ConanFile):
    settings = "os", "compiler", "build_type", "arch"
    generators = "CMakeToolchain"
    build_requires = "android_ndk/1.0"

    def generate(self):
        self.output.info("NDK: %s" % self.conf.get("tools.android:ndk_path"))
```

Removed self.env_info

The attribute `self.env_info` has been replaced by:

- `self.buildenv_info`: For the dependant recipes, the environment variables will be present during the build process.
- `self.runenv_info`: For the dependant recipes, environment variables will be present during the runtime.

Read more about how to use them in the *environment management* of Conan 2.0.

Remember that if you want to pass general information to the dependant recipes, you should use the `self.conf_info` and not environment variables if they are not supposed to be reused as environment variables in the dependent recipes.

Removed self.cpp_info.builddirs

The default value (pointing to the package root folder) form `self.cpp_info.builddirs` has been removed. Also assigning it will be discouraged because it affects how *CMakeToolchain* and *CMakeDeps* locate executables, libraries, headers... from the right context (host vs build).

To be prepared for Conan 2.0:

- If you have *cmake modules* or *cmake config files* at the root of the package, it is strongly recommended to move them to a subfolder `cmake` and assing it: `self.cpp_info.builddirs = ["cmake"]`
- If you are not assigning any `self.cpp_info.builddirs` assign an empty list: `self.cpp_info.builddirs = []`.
- Instead of appending new values to the default list, assign it: `self.cpp_info.builddirs = ["cmake"]`

New properties model

Migrating legacy cpp_info attributes to set_property()

Migrating from *.names*, *.filenams* and *.build_modules* to `set_property()` is easy, but there are some details to take into account for properties like `cmake_target_name` and `cmake_file_name`. Let's see some examples.

Important: The 2 mechanisms are completely independent:

- Old way using `.names`, `.filenames` will work exclusively for legacy generators like `cmake_find_package`
- New properties, like `set_property("cmake_target_name")` will work exclusively for new generators like `CMakeDeps`. They have changed to be absolute, and that would break legacy generators.
- Recipes that want to provide support for both generators need to provide the 2 definitions in their `package_info()`

Migrating from .names to cmake_target_name

It is important to note that `cmake_target_name` is **not** going to take the same value as the `.names` attribute did. With the `.names` attribute, if you set a name for the target in CMake, Conan would automatically create a "namespaced" target name with that name. This code, for example:

```
def package_info(self):
    ...
    self.cpp_info.filenames["cmake_find_package"] = "myname"
    ...
```

Will create a CMake target named `myname::myname`.

The property `cmake_target_name` accepts **complete** target names. That means that the name you set with this property will be the one added to the CMake generated files without appending any more information to it. To translate the last example to the set_property model you should add the following declaration:

```
def package_info(self):
    ...
    self.cpp_info.set_property("cmake_target_name", "myname::myname")
    ...
```

Note that you can use whatever name you want, it can have a different namespace, like `mynamespace::myname` or use a name with no namespace at all.

Also, please note that you may want to have different target names for both config and module CMake generated files. For example, you have a package named `myssl` and you want to generate a `Findmyssl.cmake` module that declares the target `MySSL::SSL`, but for config mode you want to declare the target `MySSL` without namespaces. You can do that using the `cmake_module_target_name` property. Also, when setting this property, remember to set `cmake_find_mode` so that *CMakeDeps* generates those module files. Let's see an example:

```
class MySSL(ConanFile):
    name = "myssl"
    version = "1.0"
    ...
    def package_info(self):
        self.cpp_info.set_property("cmake_target_name", "MySSL")
        self.cpp_info.set_property("cmake_module_target_name", "MySSL::SSL")
        self.cpp_info.set_property("cmake_find_mode", "both")
    ...
```

Migrating from .filenames to cmake_file_name

To migrate from `.filenames` to names just use the same `.filenames` value for the property `cmake_file_name`. For example:

```
def package_info(self):
    ...
    self.cpp_info.filenames["cmake_find_package"] = "MyFileName"
    self.cpp_info.filenames["cmake_find_package_multi"] = "MyFileName"
    ...
```

Could be declared like this with `set_property()`:

```
def package_info(self):
    ...
    self.cpp_info.set_property("cmake_file_name", "MyFileName")
    ...
```

Please note that for the legacy `.names` and `.filenames` model, if `.filenames` is not declared but `.names` is, then Conan will automatically set the value of `.filenames` to the value of `.names`. So for example:

```
def package_info(self):
    ...
    self.cpp_info.names["cmake_find_package"] = "SomeName"
    self.cpp_info.names["cmake_find_package_multi"] = "SomeName"
    ...
```

This will use "SomeName" to compose the generated filenames. In this case you should set `cmake_file_name` to "SomeName":

```
def package_info(self):
    ...
    self.cpp_info.set_property("cmake_file_name", "SomeName")
    ...
```

Also, please note that you may want to use different file names for both config and module CMake generated files. If we take the previous example of the `myssl` and you want to generate a `FindMySSL.cmake` for module mode and `myssl-config.cmake` for config mode, you can set the `cmake_module_file_name` to set the value for the module file:

```
class MySSL(ConanFile):
    name = "myssl"
    version = "1.0"
    ...
    def package_info(self):
        self.cpp_info.set_property("cmake_file_name", "myssl")
        self.cpp_info.set_property("cmake_module_file_name", "MySSL")
        self.cpp_info.set_property("cmake_find_mode", "both")
    ...
```

You can read more about this properties in the *CMakeDeps* properties reference.

Migrating components information

As we said, all these properties but `cmake_file_name` and `cmake_module_file_name` have components support, so for example:

```
def package_info(self):
    ...
    self.cpp_info.components["mycomponent"].names["cmake_find_package"] = "mycomponent-
↪name"
    self.cpp_info.components["mycomponent"].names["cmake_find_package_multi"] =
↪"mycomponent-name"
    self.cpp_info.components["mycomponent"].names["pkg_config"] = "mypkg-config-name"
    self.cpp_info.components["mycomponent"].build_modules.append(os.path.join("lib",
```

(continues on next page)

(continued from previous page)

```
↪"mypkg_bm.cmake"))
    ...
```

Could be declared like this with the properties model:

```
def package_info(self):
    ...
    self.cpp_info.components["mycomponent"].set_property("cmake_target_name", "component_
↪namespace::mycomponent-name")
    self.cpp_info.components["mycomponent"].set_property("cmake_build_modules", [os.path.
↪join("lib", "mypkg_bm.cmake")])
    self.cpp_info.components["mycomponent"].set_property("pkg_config_name", "mypkg-
↪config-name")
    self.cpp_info.components["mycomponent"].set_property("custom_name", "mycomponent-name
↪", "custom_generator")
    ...
```

Please **note** that most of the legacy generators like *cmake*, *cmake_multi*, *cmake_find_package*, *cmake_find_package_multi* and *cmake_paths* do not listen to these properties at all, so if you want to maintain compatibility with consumers that use those generators and also that information for new generators like *CMakeDeps* you need both models living together in the same recipe.

See also:

Read *Using Components* and *package_info()* to learn more.

Using `.names`, `.filenames` and `.build_modules` will not work anymore for new generators, like *CMakeDeps* and *PkgConfigDeps*. They have a new way of setting this information using `set_property` and `get_property` methods of the `cpp_info` object (available since Conan 1.36).

```
def set_property(self, property_name, value)
def get_property(self, property_name):
```

New properties `cmake_target_name`, `cmake_file_name`, `cmake_module_target_name`, `cmake_module_file_name`, `pkg_config_name` and `cmake_build_modules` are defined to allow migrating `names`, `filenames` and `build_modules` properties to this model. In Conan 2.0 this will be the default way of setting these properties for all generators and also passing custom properties to generators.

Important: The 2 mechanisms are completely independent:

- Old way using `.names`, `.filenames` will work exclusively for legacy generators like `cmake_find_package`
- New properties, like `set_property("cmake_target_name")` will work exclusively for new generators like `CMakeDeps`. They have changed to be absolute, and that would break legacy generators.
- Recipes that want to provide support for both generators need to provide the 2 definitions in their `package_info()`

New properties defined for *CMake* generators family, used by *CMakeDeps* generator:

- **cmake_file_name** property will define in `CMakeDeps` the name of the generated config file (`xxx-config.cmake`)
- **cmake_target_name** property will define the absolute target name in `CMakeDeps`
- **cmake_module_file_name** property defines the generated filename for modules (`Findxxxx.cmake`)

- **cmake_module_target_name** defines the absolute target name for find modules.
- **cmake_build_modules** property replaces the `build_modules` property.
- **cmake_find_mode** will tell *CMakeDeps* to generate config files, modules files, both or none of them, depending on the value set (`config`, `module`, `both` or `none`)

Properties related to *pkg_config*, supported by both legacy *pkg_config* and new *PkgConfigDeps*:

- **pkg_config_name** property equivalent to the `names` attribute.
- **pkg_config_custom_content** property supported by both generators that will add user-defined content to the *.pc* files created by the generator
- **component_version** property supported by both generators that set a custom version to be used in the `Version` field belonging to the created `*.pc` file for that component.

Properties related to *pkg_config*, only supported by new *PkgConfigDeps*:

- **pkg_config_aliases** property sets some aliases of any package/component name for the `PkgConfigDeps` generator only, it doesn't work in `pkg_config`. This property only accepts list-like Python objects.

All of these properties, but `cmake_file_name` and `cmake_module_file_name` can be defined at the global `cpp_info` level or at the component level.

The *set/get_property* model is very useful if you are creating a custom generator. Using `set_property()` you can pass the parameters of your choice and read them using the `get_property()` method inside the generator.

```
def package_info(self):
    ...
    # you have created a custom generator that reads the 'custom_property' property and␣
↪you set here
    # the value to 'prop_value'
    self.cpp_info.components["mycomponent"].set_property("custom_property", "prop_value")
    ...
```

Please **check a detailed migration guide** in the *dedicated section*.

23.1.11 Removed imports() method

The `def imports(self)` method from the conanfile has been removed. If you need to import files from your dependencies you can do it in the `generate(self)` method with the new `copy` tool:

```
from conan.tools.files import copy

def generate(self):
    for dep in self.dependencies.values():
        copy(self, "*.dylib", dep.cpp_info.libdirs[0], self.build_folder)
        copy(self, "*.dll", dep.cpp_info.libdirs[0], self.build_folder)
```

23.1.12 Changes in the test_package recipe

In Conan 2.0, the `test_package/conanfile.py` needs to declare the requirement being tested explicitly. To be prepared you have to set the attribute `test_type="explicit"` (this will be ignored in 2.0) to make Conan activate the explicit mode, then declaring the requirement using the `self.tested_reference_str` that contains the reference being tested.

```
from conan import ConanFile

class MyTestPkg(ConanFile):
    test_type = "explicit"

    def requirements(self):
        # A regular requirement
        self.requires(self.tested_reference_str)

    def build_requirements(self):
        # If we want to test the package as a tool_require (formerly `test_type = "build_
↪requires"`)
        # Keep both "requires()" and "tool_requires()" if you want to test the same␣
↪package both as a regular
        # require and a tool_require (formerly `test_type = "build_requires", "requires"`)
        self.tool_requires(self.tested_reference_str)
```

23.1.13 Other recipe changes

The environment management

The environment management has changed quite a bit. In Conan 1.X the environment was managed by modifying the environment of Python (of the running process), often using the `environment_append` tool, which is not available in 2.0 anymore. In Conan 2.0, all the applied environment variables are managed by script files (sh, bat) that will be run just before calling the command specified in every `self.run("mycommand")`.

These "environment launchers" can be organized by scopes. Conan will aggregate all the launchers of the same scope in a single launcher called `conan<scope_name>.bat/sh`.

For example, if you need to call your build system, passing some environment variables:

```
from conan import ConanFile
from conan.tools.env import Environment

class MyTestPkg(ConanFile):
    ...
    def generate(self):
        env = Environment()
        env.define("foo", "var")
        # scope="build" is the default
        envvars = env.vars(self, scope="build")
        # This will generate a my_launcher.sh but also will create a "conan_build.sh"␣
↪calling "my_launcher.sh"
        envvars.save_script("my_launcher")
```

(continues on next page)

(continued from previous page)

```
    def build(self):
        # by default env="conanbuild"
        self.run("my_build_system.exe", env="conanbuild")
```

The resulting command executed in the build() method would be something like:

```
$ conan_build.sh && my_build_system.exe
```

So the environment variable `foo` declared in the `generate()` method will be automatically passed to the `my_build_system.exe`.

There are two generators managing the environment, the `VirtualBuildEnv` and the `VirtualRunEnv`. By default, these generators are automatically declared in Conan 2.0 but you have to explicitly declare them in Conan 1.X otherwise you can set `tools.env.virtualenv:auto_use=True` in the `global.conf`.

- **VirtualBuildEnv**: It will generate a *conanbuildenv* .bat or .sh script containing environment variables of the build time environment. That information is collected from the direct `tool_requires` in "build" context recipes from the `self.buildenv_info` definition plus the `self.runenv_info` of the transitive dependencies of those `tool_requires`.

  The scope used by the `VirtualBuildEnv` is `build` so, as explained before, it will be applied by default before calling any command.

  Check more details *here*.

- **VirtualRunEnv**: It will generate a *conanrunenv* .bat or .sh script containing environment variables of the run time environment. The launcher contains the runtime environment information, anything that is necessary for the environment to actually run the compiled executables and applications. The information is obtained from the `self.runenv_info` and also automatically deducted from the `self.cpp_info` definition of the package, to define `PATH`, `LD_LIBRARY_PATH`, `DYLD_LIBRARY_PATH`, and `DYLD_FRAMEWORK_PATH` environment variables.

  The scope used by the `VirtualRunEnv` is `run` so if you need that environment applied you need to specify it in the `self.run` command.

  An example of usage of the `conanrun` is the test_package of a recipe that builds a shared library:

  ```
  import os
  from conan import ConanFile
  from conan.tools.env import Environment

  class MyTestPkg(ConanFile):
      generators = "VirtualRunEnv"

      ...

      def test(self):
          my_app_path = os.path.join(self.cpp.build.bindirs[0], "my_app")
          # The default env is "conanbuild" but we want the runtime here to locate␣
  ↪the shared library
          self.run(my_app_path, env="conanrun")
  ```

  Check more details *here*.

Windows Subsystems

If you want to run commands inside a Windows subsystem (e.g bash from msys2) you have to set the `self.win_bash=True` in your recipe, instead of using the deprecated `self.run(..., win_bash=True)` from 1.X.

You need to configure how to run the commands with two config variables:

- **tools.microsoft.bash:subsystem**: Possible values: 'msys2', 'msys', 'cygwin', 'wsl' and 'sfu'
- **tools.microsoft.bash:path** (Default "bash"): Path to the shell executable.

Any command run with `self.run`, if `self.win_bash == True` will run the command inside the specified shell.

Symlinks

Conan won't alter any symlink while exporting or packaging files. If any manipulation to the symlinks is required, the package *conan.tools.files.symlinks* contains some tools to help with that.

23.2 Commands

There is no "compatible with 2.X" commands introduced in Conan 1.X. You will need to adapt to the new commands once you migrate to Conan 2.0

23.2.1 Most common commands overview

conan install

Almost the same command, the major change is the way to specify (or compete if not defined) the fields of the reference. Remember that in Conan 1.X you have to specify the build profile or activate the conf `core:default_build_profile=default`.

```
$ conan install . [--name=mylib] [--version=1.0] [-pr:b=build_profile] [-pr:h=host_
↪profile]
```

conan create

Same changes as *conan install*:

```
$ conan create . [--name=mylib] [--version=1.0] [-pr:b=build_profile] [-pr:h=host_
↪profile]
```

conan graph info

This is the substitute of the old "conan info". The syntax is very similar to `conan install` and `conan create`

```
$ conan graph info . [--name=mylib] [--version=1.0] [-pr:b=build_profile] [-pr:h=host_
↪profile]
```

conan search

The `conan search` will search, by default, in all the remotes (not in the local cache):

```
$ conan search "zlib*"

myremote:
  zlib
    zlib/1.2.11
conancenter:
  zlib-ng
    zlib-ng/2.0.2
    zlib-ng/2.0.5
    zlib-ng/2.0.6
  zlib
    zlib/1.2.11
    zlib/1.2.8
```

If you want to explore the local cache there is a command `conan list recipes <pattern>`.

23.2.2 Unified patterns in command arguments

The arguments in Conan 1.X where we specified recipe names require now a valid reference pattern. A valid reference pattern contains the `*` character or at least the `name/version` part of a reference (`name/version@user/channel`).

There are some examples:

- The `--build` argument when referring to a package:

Listing 15: **From:**

```
conan install . --build zlib
```

Listing 16: **To:**

```
conan install . --build zlib*
conan install . --build zlib/1.2.11
conan install . --build zlib/1.*
```

- The `--options` and `--settings` arguments when used scoped:

Listing 17: **From:**

```
conan install . -s zlib:arch=x86 -o zlib:shared=True
```

Listing 18: **To:**

```
conan install . -s zlib*:arch=x86 -o zlib*:shared=True
conan install . -s zlib/1.2.11@user/channel:arch=x86 -o zlib/1.2.11:shared=True
```

23.2.3 Removed "conan package"

The `conan package` command has been removed. If you are developing a recipe and want to test that the package method is correct, we recommend using the `conan export-pkg .` instead and exploring the package folder in the cache to check if everything is ok.

23.2.4 Removed "conan copy"

Do not use the `conan copy` command to change user/channel. Packages will be immutable, and this command will disappear in 2.0. Package promotions are generally done on the server-side, copying packages from one server repository to another repository.

23.2.5 Custom commands

You can build custom commands on top of the Conan Python API. WIP documentation.

23.3 General changes

23.3.1 Host and Build profiles

Use always *build and host profiles*.

Conan 1.x uses one profile by default, to start using two profiles, please do the following:

- Pass `-pr:b=default` in the command line to most commands.
- Or set the variable `core:default_build_profile=default` at the *global.conf* file to apply it always, automatically.

Do not use `os_build`, `arch_build` anywhere in your recipes or code.

- Revisions

Conan 2.0 uses *revisions* by default and the local cache 2.0 will store multiple recipe and package revisions for your Conan packages (Conan 1.X supports only one revision). To start working with revisions enabled in Conan 1.X, please enable them in your Conan configuration:

```
$ conan config set general.revisions_enabled=True
```

23.3.2 Lowercase references

Move all your packages to lowercase. Uppercase package names (or versions/user/channel) will not be allowed in 2.0.

23.3.3 Default Package ID mode

Work in progress

23.3.4 Compatible packages

Work in progress

23.3.5 Extensions

Work in progress

INDEX