FIFTH EDITION · IRIS V14

Iris

The Professional Guide to the Iris Web Framework for Go



WRITTEN BY

Gerasimos Maropoulos

iris-go.com
hellenic.dev

- Creator of Iris
- Creator of Plexon AI
- Founder of Hellenic Development

About this book

A complete, hands-on guide to building production web applications and REST APIs with Iris v14: the builder, routing and controllers, request and response handling, dependency injection, error handling, security, testing, deployment and performance — every example verified against the framework source.

Edition	Fifth Edition
Version	Iris v14
Author	Gerasimos Maropoulos
Website	https://iris-go.com
Source	https://github.com/kataras/iris
License	BSD 3-Clause License
Built	November 2026

> Upgrading from an earlier major version? Follow the migration guide at https://github.com/kataras/iris/blob/main/MIGRATION.md.

Contents

Preface 8

1 Getting Started 11

Background 11
Installation 12
Your First Application 12
Reading Parameters 14
Running the Server 14
A First Taste of the Production Stack 15
Project Layout 17
Summary 17
Further Reading 18

2 Core Concepts 19

Background 19
The Big Picture 20
The Application 20
The Context 22
The Router 24
Handlers and the Middleware Pipeline 26
The Request Lifecycle 27
Where Dependency Injection Sits 29
Summary 30
Further Reading 31

3 Building Applications with iris.NewBuilder 32

Background 32
Why the Builder 33
The Step Chain 34
Step Reference 35
A Complete Application 44
Testing the Built Application 49
When to Use iris.New Instead 50
Summary 50
Further Reading 51

4 Routing & Controllers 52

Background 52
Route Registration 53
Path Parameters 54
Route Groups 57
Controllers 58
Guards and Positional Middleware 60
Error-Returning Handlers 61
Subdomains 62
Route Names and Introspection 63
Generic MVC Controllers 64
Summary 64
Further Reading 65

5 Reading Requests 66

Background 66
Request Metadata 67
Path and Query Parameters 69
Binding JSON Requests 73
The Context Read Family 75
Pagination Input 78
Forms and File Uploads 81
Cookies 82
Body Size Limits 83
The RequestValidator Hook 84
Summary 84
Further Reading 85

6 Response Handling 86

Background 86
The Response Helpers 87
Context Writers 91
Status Codes 93
Response Headers 94
Redirects 95
File Responses 95
Streaming 96
Server-Sent Events 98
Content Negotiation 105
The Response Error Handler 105
Summary 106
Further Reading 107

7 Middleware 109

Background 109
Handlers and the Chain 110
The Registration Family 111
Guards and Per-Route Middleware 113
Writing Custom Middleware 114
Conditional Middleware 116
Error-Returning Middleware 116
What the Builder Already Registers 118
Ordering Rules 119
Summary 121
Further Reading 121

8 Dependency Injection 123

Background 123
Why Constructor Injection 124
A Complete Wired Application 125
Container Mechanics 127
Interface Resolution 129
Services and the Builder 130
Controllers and Dependencies 131
Dependency-Injected Handlers 132
Request-Scoped Dependencies 133
Startup Hooks and Shutdown Closers 134
Fail Fast at Startup 135
Testing with Fakes 136
Summary 137
Further Reading 138

9 Error Handling 139

Background 139
The Philosophy: One Place for Errors 140
The Wire Format 142
Canonical Error Codes 142
Sending Errors Directly 144
The Central Error Map 146
Forwarding Errors from Handlers 149
Field Validation 152
Private Errors 154
Database Errors and Resolvers 155
Handlers That Return Errors 156
HTML and RFC 7807 Rendering 157
Logging, Collection and Error IDs 158
Router-Level Defaults 160
Summary 162
Further Reading 162

10 Authentication & Authorization 164

Background 164
Choosing an Approach 165
Basic Authentication 166
API Keys for Machine Callers 168
The Auth SDK 171
The JWT Middleware 178
Role and Permission Guards 180
Session-Based Authentication 181
Identity Server and OAuth2 185
Summary 186
Further Reading 187

11 Configuration Management 188

Background 188
The Options Struct 189
Merge Semantics 190
Defaults and EditOptions 191
YAML Files and the SERVER_CONFIG Variable 191
Router Options 193
Context Options 194
Embedding Options in Your Own Config 196
Multi-Environment Setups 199
Server Options 200
Summary 203
Further Reading 203

12 Built-in Middleware 205

Background 205
The Middleware Catalog 206
Registration Levels 208
What the Builder Installs for You 208
CORS 210
Rate Limiting 212
Recovery 213
Compression 214
Access Logging 214
Body Limits 216
Cache Headers 216
Sessions 217
JWT 219
Basic Authentication 220
API Keys 221
Request IDs 222
Internationalization 222
Views 223
WebSockets 224
Monitoring and Profiling 225
Utility Middleware 226
Auditing Requests and Responses 228
Geolocation 229
Captcha Verification 230
Serving gRPC on the Same Port 231
Adapting net/http Middleware 232
Community Middleware 233
Summary 234
Further Reading 234

13 Database Integration 235

Background 235
The Repository Seam 236
Wiring a Store Through Services 241
PostgreSQL with kataras/pg 243
The Standard Library: database/sql 245
MongoDB, Redis and ORMs 247
Column Types That Survive JSON 247
Transactions Below the Handler 248
Mapping Database Errors to HTTP 249
Summary 250
Further Reading 251

14 Caching Strategies 252

Background 252
Three Layers of Caching 253
The Cache Middleware 254
Caching a Group of Endpoints 257
Conditional Requests by Hand 258
Server-Side Response Caching 260
Application-Level Caching 264
Distributed Caching 267
Cache Invalidation 267
Summary 268
Further Reading 269

15 Internationalization (i18n) 270

Background 270
Package Overview 271
Loading Translations 271
The Request Handler 274
Translating in Handlers 275
Pluralization and Variables 276
Path and Subdomain Rewriting 278
Template Integration 278

Complete Example 279
Summary 281
Further Reading 281

16 Testing 282

Background 282
How Testing Works in Iris 283
Your First Test 284
Designing for Testability 285
The Application Under Test 286
End-to-End Tests 289
The Request and Assertion Chain 290
Fakes and One-Method Stubs 291
Testing Guards and Authentication 292
Testing Validation and the Error Map 293
Testing Middleware in Isolation 294
Unit Testing Domain Logic 295
httptest Options and Variants 296
Running the Suite 298
Summary 299
Further Reading 300

17 Monitoring & Observability 301

Background 301
Structured Access Logging 302
Request Correlation 306
Application Logging 308
The Monitor Dashboard 309
Profiling with pprof 311
Per-Request Cost Measurement 313
Server-Timing 314
Health Checks and Build Information 314
Third-Party Metrics and Tracing 316
Summary 317
Further Reading 317

18 Security Best Practices 319

Background 319
Secure by Default 320
Transport Security 321
Bounding the Request 322
Input Validation 325
Keeping Internals Private 327
Sessions and Cookies 328
Authentication Hardening 330
CORS 332
CSRF Protection 333
Static File Safety 334
Security Headers 335
Secrets Hygiene 336
Production Checklist 337
Summary 338
Further Reading 338

19 Deployment & DevOps 340

Background 340
Building for Production 341
Serving: Listen and Run 342
Graceful Shutdown 345
Containerizing with Docker 348
Running Behind a Reverse Proxy 349
Health Checks and Kubernetes 350
Environment Configuration 352
Running Under systemd 352
Continuous Integration 353
Summary 354
Further Reading 354

20 Performance Optimization 355

Background 355
Measure First 356
What Iris Already Does 359
Response Efficiency 361
Request Efficiency 363
Caching 365
Database Efficiency 365
Concurrency in Handlers 366
Runtime Tuning 368
Summary 368
Further Reading 369

21 AI Backends 370

Background 370
One Handler, Both Shapes 371
The Part Vocabulary 372
Encoders 373
Reasoning Fenced in a Tag 374
Policy and Accounting 375
Relaying an Upstream 376
Typed Upstream Failures 378
Errors After the First Byte 379
A Gateway with Failover 381
What Breaks in Production 382
Summary 383
Further Reading 384

22 Real-World Examples 385

Background 385
Shortl: A URL Shortener 386
Stockroom: An Inventory API 392
Choosing Between the Two Shapes 413
Summary 414
Further Reading 414

23 Upgrading to v14 416

Background 416
The Canonical Migration Guide 416
Tips for a Smooth Upgrade 418
Starting Fresh 418
Summary 418
Further Reading 419

24 Advanced Topics 420

Background 420
Generic MVC Controllers 421
Custom Typed Contexts 429
Global Patches 431
Custom Path Macros 432
Wrapping the Router 433
Multi-Application Composition 435
Standard Library Interop 436
Built-in Controllers 438
Summary 440
Further Reading 441

Epilogue 442

Preface

Iris is a web framework for Go, built for writing HTTP servers, REST APIs and backend services that are fast to run and pleasant to maintain. This book is its professional guide: a complete path from your first route to a tested, production-shaped application, written for Iris v14 and verified against the framework's source code.

> **Ask AI about Iris, any time.** Iris has a free, live AI wiki at codewiki.google/github.com/kataras/iris, generated by Google directly from the framework's source code and kept in sync with it (each page shows the exact commit it reflects). It organizes the codebase into browsable topics with architecture diagrams, and its built-in chat answers questions about Iris in plain language, grounded in the actual code. Keep it open next to this book: when you wonder "how do I..." mid-chapter, ask there and jump back.
>
> If you would rather have the assistant on your own machine, Plexon AI (plexon.ai) is a desktop assistant for Windows, macOS and Linux, built by the author of Iris. It ships an Iris skill built in, covering the same ground as this book in a form written for an agent to read, and its Software Developer persona turns it on.

Who this book is for

You should be comfortable with Go (syntax, packages, interfaces, the standard library basics) and with HTTP concepts such as methods, headers and status codes. No prior Iris experience is assumed. If you have built services with the standard library's `net/http` or another framework, you will move quickly; the book takes care to explain not just what each API does but why it is shaped the way it is.

How the book is organized

The chapters build on one another and are best read in order the first time; afterwards each stands alone as a reference.

- **Chapters 1–3** are the foundation: installation, the framework's core concepts, and `iris.NewBuilder`, the recommended way to assemble a production application.
- **Chapters 4–7** cover the request path: routing and controllers, reading requests, writing responses, and the middleware pipeline.
- **Chapters 8–11** cover application structure: dependency injection, error handling, authentication and authorization, and configuration.
- **Chapters 12–15** tour the built-in middleware, database integration, caching and internationalization.
- **Chapters 16–20** are about running in production: testing, observability, security, deployment and performance.
- **Chapters 21–24** close with AI backends, complete applications, the upgrade path to v14, and advanced framework extension points. An epilogue sends you off with the habits worth keeping and where to go next.

Conventions

Each chapter opens with a **Background** section that explains the underlying concepts in plain language: what an HTTP header actually is, what a database transaction guarantees, what authentication means. No prior specialty knowledge is assumed. Each chapter closes with a **Further Reading** list of canonical references (MDN, the relevant RFCs, official Go documentation, OWASP) for going deeper. Readers who already know a chapter's fundamentals can skip its Background entirely.

Code examples use full import paths (`github.com/kataras/iris/v14/...`) and are complete enough to adapt directly; where an example belongs to a larger program, the chapter says so. Shell commands are written for a POSIX shell and work in PowerShell unless noted. The examples target Go 1.26 or newer.

Two companion documents in the framework repository pair well with this book: the hands-on TUTORIAL.md, which builds one small service end to end, and the migration guide for readers upgrading an application from an earlier major version. This book itself covers v14 only.

About Iris

Iris has been developed in the open since 2016 by Gerasimos Maropoulos (@kataras) together with hundreds of contributors. The framework lives at github.com/kataras/iris and its website is iris-go.com. Issues, discussions and contributions are welcome there. This book is maintained separately from the framework, and corrections to it are welcome the same way.

CHAPTER 1

Getting Started

Iris is a web framework for the Go programming language, built for HTTP APIs and web applications. It gives you a fast router with typed path parameters, a single request/response context with safe accessors, dependency injection for wiring services into handlers, and a set of production middleware (recovery, compression, CORS, access logging) that assemble into one coherent stack. The module path is `github.com/kataras/iris/v14` and it requires Go 1.27 or newer. This chapter installs Iris, builds a first application, and previews the production pattern the rest of the book expands on.

Background

If Iris is your first web framework, or Go your first server-side language, this section explains the moving parts before you install anything. If you have built HTTP services before, skip to Installation.

What a server does. A web application is a program that never exits: it listens on a network port (an address such as `:8080`), waits for clients (browsers, mobile apps, `curl`, other services) to connect, and answers each request they send. One request in, one response out; HTTP is that simple loop repeated millions of times.

What a request and a response look like. Both are short structured messages. A request carries a method (a verb such as `GET` for reading or `POST` for submitting data), a path (`/orders/42`), headers (key-value metadata such as `Accept: application/json`), and sometimes a body (the submitted data itself). The response carries a status code (a three-digit result: `200` success, `404` not found, `500` server error), its own headers, and a body: the HTML, JSON or file the client asked for. Every feature in this book manipulates some part of these two messages.

What the framework does for you. Go's standard library can already serve HTTP. A framework adds the layer you would otherwise write by hand for every project: routing (deciding which of your functions runs for `GET /orders/42` versus `POST /orders`), parsing (turning raw request text

into typed Go values you can trust), responding (turning your Go values back into correct HTTP with the right status code and content type), and the shared machinery around them: logging, crash recovery, security headers. The problem that disappears from your code is plumbing: you write the part that is unique to your application, and the framework supplies the part that is the same in every application.

What a Go module is. Go installs dependencies per project, not system-wide. A module is a folder with a `go.mod` file that names your project and pins the exact version of every library it uses. `go mod init myapp` creates that file; `go get` adds a dependency to it and downloads the code. There is no separate package manager to install and no virtual environment to activate. The `go` tool you already have does all of it.

The examples below use `curl`, a command-line tool (preinstalled on most systems) that sends an HTTP request to a URL and prints the response, so you can poke at your server without a browser.

Installation

You need Go 1.27+ and basic familiarity with the language (the Go Tour covers everything this book assumes). Create a module and add Iris to it:

```sh
mkdir myapp && cd myapp
go mod init myapp
go get github.com/kataras/iris/v14@latest
```

That is the entire setup. Iris is a regular Go module dependency: no code generators, no CLI tools, no configuration files required to start.

Your First Application

Create `main.go`:

```go
package main

import "github.com/kataras/iris/v14"

func main() {
    app := iris.New()

    app.Get("/", func(ctx iris.Context) {
        ctx.WriteString("Hello, Iris!")
```

```
    })

    app.Get("/greet/{name:string}", func(ctx iris.Context) {
        name := ctx.Params().GetString("name")
        ctx.JSON(iris.Map{"message": "hello, " + name})
    })

    app.Get("/orders/{id:int}", func(ctx iris.Context) {
        id := ctx.Params().GetInt("id")
        ctx.JSON(iris.Map{"order": id, "status": "pending"})
    })

    app.Listen(":8080")
}
```

Run it and try the routes:

```sh
go run main.go

curl http://localhost:8080/            # Hello, Iris!
curl http://localhost:8080/greet/iris  # {"message":"hello, iris"}
curl http://localhost:8080/orders/42   # {"order":42,"status":"pending"}
curl http://localhost:8080/orders/abc  # 404: "abc" is not an int
```

Three things are happening here.

`iris.New()` returns an `*iris.Application` with sensible defaults. Route registration methods (`Get`, `Post`, `Put`, `Delete`, `Patch` and the rest) hang directly off the application, and each takes a path pattern plus one or more handlers.

Every handler receives an `iris.Context`, the single object you use to read the request and write the response. `ctx.WriteString` writes a plain string body; `ctx.JSON` serializes any Go value (here `iris.Map`, a shorthand for `map[string]any`) and sets the content type.

The path patterns carry typed macros. `{name:string}` matches any single path segment; `{id:int}` matches only integers, so a request for `/orders/abc` never reaches your handler. The most useful macros:

Macro	Matches
`{name:string}`	any single path segment
`{id:int}`	integers
`{id:uuid}`	UUID v1/v4
`{ok:bool}`	true/false, 1/0
`{email:email}`	e-mail addresses
`{path:path}`	the rest of the path, any depth

Chapter 4 covers the full macro language, route grouping with `app.Router` and the controller pattern.

Reading Parameters

Iris exposes request input through typed accessors with safe defaults. There are no `(value, err)` pairs to unpack for simple reads: a missing string is `""`, a missing or malformed number is `-1`.

Path parameters come from `ctx.Params()`:

```go
app.Get("/users/{id:int}", func(ctx iris.Context) {
    id := ctx.Params().GetInt("id") // already validated by the macro.
    ctx.JSON(iris.Map{"user_id": id})
})
```

Query string values come from `ctx.Query()` the same way:

```go
app.Get("/search", func(ctx iris.Context) {
    q := ctx.Query().GetString("q")

    page := ctx.Query().GetInt("page")
    if page < 1 {
        page = 1 // GetInt returns -1 when the key is absent.
    }

    ctx.JSON(iris.Map{"query": q, "page": page})
})
```

Because the macro already validated `{id:int}`, the handler can trust the value. When you need richer input handling (JSON bodies, forms, headers, validation), Chapter 5 covers it in depth.

Running the Server

`app.Listen(addr)` starts an HTTP server on the given address and blocks until the server stops. On SIGINT/SIGTERM it stops accepting new connections and drains in-flight requests gracefully; you do not write any shutdown code for the common case. (SIGINT and SIGTERM are the shutdown signals your process receives when you press Ctrl+C or when a container platform stops it.) The process exits with status 0 once those requests are answered, or after 15 seconds if they are not.

If the server cannot start, because the port is taken or a route pattern is malformed, Iris logs the failure and the process exits with status 1. That is why the call takes no error check: a graceful stop returns nil once the shutdown has finished, which is the drain and then any cleanup registered to run with it, and a failed start never returns at all. After a signal the process exits at that same moment, so do not count on statements placed after `Listen` to run. Chapter 19 covers cleanup on shutdown and the opt-out for a process that runs more than one server.

When you need more than an address, pass `iris.ServerOptions` to `app.Run`. TLS is the everyday reason. TLS (Transport Layer Security) is the encryption layer that turns `http://` into `https://`: it keeps traffic private and proves to clients that they are talking to your server, not an impostor. It requires a certificate: a file, signed by a trusted authority, that ties your domain name to your server. Let's Encrypt is a free certificate authority that issues and renews such certificates automatically:

```go
// Manual certificates:
app.Run(iris.ServerOptions{
    Addr:     ":443",
    CertFile: "cert.pem",
    KeyFile:  "key.pem",
})

// Automatic certificates from Let's Encrypt:
app.Run(iris.ServerOptions{
    Addr:        ":443",
    Letsencrypt: true,
    Hosts:       []string{"example.com"},
})
```

`app.Listen(":8080")` is `app.Run` with a small set of defaults filled in for you: a TCP network, a three-minute keep-alive, socket sharding and the application's log writer. The options struct also covers TLS, timeouts and tunneling; Chapter 19 walks through the production serving choices.

A First Taste of the Production Stack

`iris.New()` plus hand-registered routes is fine for small programs and for learning. For real services, Iris ships a builder that assembles the recommended stack: a compiler-enforced chain of steps that configures CORS, compression, logging, a health endpoint, centralized error handling and dependency injection, then mounts your API controllers.

Three of those terms get full chapters later but deserve one line now. Middleware is code that runs before or after every handler: the place for logging, panic recovery and security checks, written once instead of in each handler. CORS (Cross-Origin Resource Sharing) is the browser

permission system that decides which web pages served from other domains may call your API. Dependency injection means your constructors declare what they need (a database, a logger) and the framework supplies it, instead of your code reaching for global variables.

A controller in this style is a plain struct with a constructor and a `Routing(r iris.Router)` method. Here is the smallest complete example:

```go
package main

import (
    "github.com/kataras/iris/v14"
)

// GreetAPI is a controller: a constructor that declares its dependencies
// (none yet) and a Routing method that registers its routes.
type GreetAPI struct{}

func NewGreetAPI() *GreetAPI { return &GreetAPI{} }

func (api *GreetAPI) Routing(r iris.Router) {
    r.Get("/{name:string}", api.greet)
}

func (api *GreetAPI) greet(ctx iris.Context) {
    name := ctx.Params().GetString("name")
    ctx.OK(iris.Map{"message": "hello, " + name}, nil)
}

func main() {
    app := iris.NewBuilder().
        Prefix("/api").
        AllowOrigin("none").
        Compression(true).
        LogRequests(false).
        Health(true, "development", "you@example.com").
        Services().
        RouterMiddlewares().
        Middlewares().
        API("/greet", NewGreetAPI).
        Build()

    app.Listen(":8080")
}
```

```sh
curl http://localhost:8080/api/greet/iris  # {"message":"hello, iris"}
curl http://localhost:8080/health          # health report, at the root
```

Each step returns the interface of the next, so the compiler enforces the order; the two middleware steps stay in the chain even when you pass them nothing. `RouterMiddlewares()` installs recovery and the framework's error handling for you. The `ctx.OK(value, err)` helper renders the value as 200 JSON, and hands any non-nil error to a central error map you configure once at startup, which is what keeps production handlers this short.

As the application grows, the same chain absorbs the real work: `Services(...)` registers repositories and services for injection into constructors like `NewGreetAPI`, and `Errors(...)` maps your domain errors to HTTP status codes in one place. Two references take you there:

- `TUTORIAL.md` at the repository root builds a complete, tested service on this stack step by step (DI, error mapping, pagination, httptest).
- Chapter 3 documents every builder step in depth.

Project Layout

Start small: one `main.go` is a perfectly good layout for the first day. When the program grows past a few handlers, split it into the shape the tutorial uses, three layers with one direction of dependency:

```
myapp/
├── go.mod
├── main.go            # configuration, wiring, startup
├── main_test.go       # HTTP tests against the in-memory application
├── api/               # HTTP layer: controllers and thin handlers
│   └── greet.go
└── catalog/           # your domain package: entities, rules, storage
    ├── catalog.go
    └── service.go
```

Name the domain package after your domain (the tutorial's Bookshelf service calls it `library`). The rule that keeps the layout healthy: `api` imports the domain package, and the domain package imports nothing from HTTP. That separation is what makes the service testable with plain Go interfaces later, as Chapter 16 shows.

Summary

- Iris installs as a normal Go module: `go mod init`, then `go get github.com/kataras/iris/v14@latest`.
- `iris.New()` creates an application; `app.Get(path, handler)` and friends register routes; handlers receive an `iris.Context`.
- Path patterns carry typed macros such as `{name:string}` and `{id:int}` that validate before your handler runs.
- Read input with `ctx.Params().GetString/GetInt` and `ctx.Query().GetString/GetInt`; missing values return safe defaults (`""`, `-1`), not error tuples.

- `app.Listen(addr)` serves HTTP with graceful shutdown built in; `app.Run(iris.ServerOptions{...})` adds TLS, Let's Encrypt and server tuning.
- The recommended production stack is `iris.NewBuilder` with controllers and the response helpers; the framework's `TUTORIAL.md` is the full hands-on walkthrough and Chapter 3 covers the builder in depth.
- Organize growing code as `main.go` plus an `api` package and a domain package, with dependencies pointing only toward the domain.

Further Reading

- MDN: An overview of HTTP: the best first read on how the request/response cycle actually works.
- MDN: HTTP messages: the exact structure of requests and responses. Method, path, headers, status line, body.
- Managing dependencies (go.dev): the official guide to Go modules, `go get` and `go.mod`.
- A Tour of Go: the interactive introduction to the Go language this book assumes.
- Writing Web Applications (go.dev): serving HTTP with only the standard library; useful for seeing exactly what a framework saves you.

CHAPTER 2

Core Concepts

Every Iris program is built from four cooperating parts: an **Application** that owns the server lifecycle, a **Router** that maps requests to code, a pooled **Context** that carries one request from start to finish, and plain **handlers** chained into a middleware pipeline. This chapter builds the mental model of how those parts fit together, so that the rest of the book reads as detail rather than surprise. Everything shown here is the v14 API, verified against the framework source.

Background

Chapter 1 showed a request and a response from a distance. The parts of Iris this chapter describes exist to serve that exchange, so it is worth pinning down the vocabulary first.

The anatomy of an HTTP exchange. Each request names a method, a path, headers and possibly a body. The method is the verb: `GET` reads, `POST` creates or submits, `PUT` replaces, `PATCH` partially updates, `DELETE` removes. The path (`/users/42`) names the thing the verb acts on. Headers are key-value metadata lines: who is asking (`Authorization`), what format the client wants back (`Accept`), what it is sending (`Content-Type`). The body carries the payload itself: form fields, a JSON document, an uploaded file. The response mirrors this with a status code, headers and a body. Status codes come in classes: `2xx` means success, `3xx` redirects the client elsewhere, `4xx` means the client made a mistake (`404` unknown path, `400` malformed input), and `5xx` means your server failed (`500` is the generic failure). You will choose methods and status codes constantly, so this small vocabulary pays for itself quickly.

Router and handler. A handler is a function you write that serves one request: it reads what it needs and writes a response. The router is the dispatcher in front of all your handlers: it looks at each request's method and path and decides which handler runs. "Routing" is that decision; a "route" is one registered rule, such as `GET /users/{id}` runs `getUser`. Middleware (detailed later in this chapter) is a handler that runs before or after other handlers to do shared work such as logging or authentication.

Why object pooling. A busy server may handle tens of thousands of requests per second, and Go serves each one on its own goroutine, a lightweight thread that Go can run by the million. If the framework allocated a fresh context object for every request, it would create garbage at the same rate, and Go's garbage collector would burn CPU cleaning it up. Object pooling is the fix: keep a bin of used context objects, wipe one clean for each new request, and drop it back in the bin afterwards, the way a restaurant washes plates instead of buying new ones for every meal. The only cost is a rule you must respect: never keep a context after your handler returns, because the same object will be reused for another request. The Context section below restates that rule where it matters.

The Big Picture



A request enters through a standard `net/http` server, is bound to a `Context` taken from a pool, travels through the router's middleware and the matched route's handler chain, and leaves as a written response. The `Context` is then reset and returned to the pool. There is no reflection and no magic in the hot path: a handler is a plain function, a chain is a slice of functions, and `ctx.Next()` moves an index forward.

The Application

`iris.New` returns a `*iris.Application`, a concrete struct defined in `iris.go`. Its routing surface is not its own: the struct embeds the root router builder, so every routing, middleware, and dependency-injection method you call on `app` comes from the embedded `*router.RouterBuilder`.

```go
type Application struct {
```

```
    *router.RouterBuilder // routing, middleware, DI methods

    // unexported: options, context pool, logger, closers...
}
```

What the `Application` itself adds is lifecycle management:

Method	Purpose
`app.Listen(addr)`	Build and serve on a TCP address with sane defaults; a failed start is logged and exits with status 1
`app.Run(iris.ServerOptions{...})`	Build and serve with explicit server options; same exit rule as `Listen`
`app.Build()`	Construct the router and internal state manually
`app.Shutdown(ctx)`	Gracefully stop the server and registered closers
`app.Logger()`	The application's structured logger
`app.EditOptions(fn)`	Modify options before the application is built
`app.RegisterOnBuild(before, after)`	Hooks around the build step

`app.Listen(":8080")` is a shortcut for `app.Run` with a TCP network, keep-alive, and socket sharding preconfigured. After `Build`, the application is a standard `http.Handler`, which is why it composes cleanly with the Go ecosystem and with `httptest` (see Chapter 16).

A minimal but complete program:

```go
package main

import (
    "github.com/kataras/iris/v14"
)

func main() {
    app := iris.New(iris.Options{
        Name:     "hello",
        LogLevel: "debug",
    })

    app.Get("/", func(ctx iris.Context) {
        ctx.Text("Hello, Iris!")
    })

    app.Listen(":8080")
}
```

`iris.Options` configures the application (`Name`, `LogLevel`, `Host`, plus nested `Router` and `Context` option structs). A partial literal merges with the defaults: unset fields are filled in, never zeroed. Configuration is covered in depth in Chapter 11.

For production applications the recommended entry point is the compile-time checked `iris.NewBuilder` chain rather than bare `iris.New`; that is the subject of Chapter 3. Everything in this chapter applies to both, because the builder produces the same `*iris.Application`.

The Context

`iris.Context` is an alias for `*context.Context`, the object that represents one request-response cycle. Contexts are pooled: at the start of a request one is acquired from a `sync.Pool`, wired to the incoming `*http.Request` and a wrapped response writer, and after the last handler returns it is reset and released back to the pool. This is why you must not keep a reference to `ctx` in a goroutine that outlives the handler: by the time it runs, the same object may be serving another request.

It is a context.Context

`iris.Context` implements the standard `context.Context` interface (`Deadline`, `Done`, `Err`, `Value`). You pass it directly into services, database drivers, and HTTP clients, and request cancellation propagates all the way down:

```go
func (api *BooksAPI) list(ctx iris.Context) {
    // ctx satisfies context.Context: pass it straight down.
    books, err := api.svc.Books(ctx)
    ctx.List(books, err)
}
```

For background work that must survive the response, derive a detached context with `ctx.WithoutCancel()`: it is no longer canceled when the request ends, but keeps the request's values.

Method families at a glance

The Context API is large but regular. Almost everything falls into three families:

Family	Representative methods
Request reading	`Method`, `Path`, `Host`, `RemoteAddr`, `Header(name)`, `ContentType()`, `MediaType()`, `Query()`, `Params()`, `Body()`, `ReadJSON`, `ReadForm`, `ReadQuery`, `ReadBody`, `Cookie`, `CookieValue`
Response writing	`StatusCode`, `SetHeader`, `AddHeader`, `SetContentType`, `JSON`, `Text`, `HTML`, `Writef`, `Write`, `Redirect`, `SendFile`, `ServeFile`, `SetCookie`, `ResponseHeader`, `ResponseWriter()`
Flow control	`Next`, `Skip`, `StopExecution`, `IsStopped`, `StopWithStatus`, `StopWithError`, `StopWithJSON`, `StopWithText`, `Values()`, `SetErr`, `GetErr`

Two naming rules keep the request and response sides apart:

- `ctx.Header(name)` reads a **request** header. `ctx.SetHeader` writes a **response** header, and `ctx.ResponseHeader(name)` reads one back.
- `ctx.ContentType()` and `ctx.MediaType()` describe the **request** body. The response-side counterparts are `ctx.ResponseContentType()` and `ctx.ResponseMediaType()`.

Path and query values come through typed accessors that return a value directly, with a type-specific default on failure instead of a tuple: `GetString` returns `""`, `GetInt` returns `-1`, `GetBool` returns `false`.

```go
package main

import (
    "github.com/kataras/iris/v14"
)

func search(ctx iris.Context) {
    term := ctx.Params().GetString("term") // "" when missing
    page := ctx.Query().GetInt("page")     // -1 when missing or invalid
    if page <= 0 {
        page = 1
    }

    ctx.SetHeader("X-Page-Size", "20") // response header
    ctx.JSON(iris.Map{"term": term, "page": page})
}

func main() {
    app := iris.New()
    app.Get("/search/{term}", search)
    app.Listen(":8080")
}
```

Request-scoped values

Middleware communicates with later handlers through `ctx.Values()`, a per-request key-value store, freed together with the context:

```go
func requireShelf(ctx iris.Context) {
    shelf := loadShelf(ctx) // ... 404 and return when not found
    ctx.Values().Set("shelf", shelf)
    ctx.Next()
}

func listBooks(ctx iris.Context) {
    // Type-safe retrieval with the generic method.
    shelf, ok := ctx.ValueAs[*Shelf]("shelf")
    if !ok {
        // The route is not behind requireShelf: a wiring bug, not a client one.
        errors.Internal.Log(ctx, "shelf value missing from the request store")
        return
    }
```

```
    ctx.List(shelf.Books, nil)
}
```

Reading request bodies (`ReadJSON`, `ReadForm`, and the `ctx.BindJSON[T]()` method with automatic validation) is covered in Chapter 5; response rendering in Chapter 6.

The Router

The router has two lives. At design time it is a **builder**: you register routes, groups, and middleware on it. At build time (`app.Build`, called by `Listen` / `Run`) it compiles everything into a **mux**: the internal `ServeMux` that performs the actual matching of method, subdomain, and path for every request. Registration cost therefore never shows up at serve time.

("Mux" is short for multiplexer: the single component every request enters, which fans each one out to the correct handler chain. Go's standard library uses the same word for its `http.ServeMux`.)

Routes

Routes are registered with HTTP-verb methods (`Get`, `Post`, `Put`, `Delete`, `Patch`, `Head`, `Options`), with `Handle` for explicit or multiple methods, with `Any` for all methods, or with `Resource` for `GET` + `HEAD`:

```go
app.Get("/about", about)
app.Post("/contact", contact)
app.Handle("GET,POST", "/form", form)
```

Path segments can be dynamic, typed parameters called macros:

```go
app.Get("/users/{id:uuid}", getUser)      // validated UUID
app.Get("/posts/{page:int}", listPosts)   // integer
app.Get("/files/{p:path}", serveTree)     // wildcard, may contain /
app.Get("/tags/{name}", tagged)           // untyped = string

// Macro functions refine a type:
app.Get("/products/{id:int min(1)}", product)
```

A parameter that fails its type check does not match the route; in builder-built applications a malformed typed parameter renders a `400 INVALID_ARGUMENT` response. The full macro catalog and custom macro registration live in Chapter 4.

Groups: Router and Child

`Router(relativePath, ...)` creates a child router mounted under a path prefix. A child inherits its parent's middleware, context options, and dependency container; new routes it registers land in the one shared route registry. Three call styles cover most needs:

```go
// 1. Plain group, optionally with middleware handlers.
api := app.Router("/api/v1")
api.Get("/health", health)

users := api.Router("/users") // nested: /api/v1/users
users.Get("/", listUsers)
users.Get("/{id:uuid}", getUser)

// 2. Function form: configure the group in one place.
app.Router("/admin", func(r iris.Router) {
    r.Use(requireAdmin)
    r.Get("/stats", stats)
})

// 3. Controller form: a constructor resolved by dependency injection
//    (see the last section of this chapter).
app.Router("/books", NewBooksAPI)
```

`Child(relativePath, handlers...)` is the underlying primitive that clones the builder for a sub-path; `Router` builds on it and additionally accepts functions and controllers. Subdomain groups exist too: `app.Subdomain("admin")` returns a router whose routes match only `admin.<host>` requests.

When nothing matches

If the mux finds no route, the request runs the no-route chain (default: `404 Not Found`). If the path exists but the method does not, the request runs the no-method chain (default: `405 Method Not Allowed`), but only when you have asked for it. `Router.FireMethodNotAllowed` defaults to false, and until you set it a wrong-method request lands in the no-route chain like any other miss. Both chains are replaceable:

```go
app.HandleNoRoute(func(ctx iris.Context) {
    errors.NotFound.Message(ctx, "no route matches this path")
})
app.HandleNoMethod(func(ctx iris.Context) {
    errors.MethodNotAllowed.Message(ctx, "this path does not accept that method")
})
```

Those defaults are the plain status text on an `iris.New()` application. An application built with `iris.NewBuilder()` fills both chains with `errors.Handler` from `rest/errors`, so a miss answers a JSON `NOT_FOUND` or `METHOD_NOT_ALLOWED` instead. It fills them when the application is built and only when they are empty, so on a builder application your own handlers have to be registered

from a `Configurator` passed to `iris.NewBuilder`, or on the unbuilt value `iris.Default()` returns; a `HandleNoRoute` called on the built application does not replace the JSON answer. Structured error responses via the `rest/errors` package are the topic of Chapter 9.

Handlers and the Middleware Pipeline

A handler is any `func(ctx iris.Context)`, the `iris.Handler` type. There is exactly one signature; middleware and final handlers are the same kind of function, distinguished only by position in the chain. A handler either finishes the response or calls `ctx.Next()` to hand over to the next handler. Code after `ctx.Next()` runs on the way back out, which is how timing, logging, and cleanup middleware work:

```go
package main

import (
    "time"

    "github.com/kataras/iris/v14"
    "github.com/kataras/iris/v14/rest/errors"
)

func timing(ctx iris.Context) {
    start := time.Now()
    ctx.Next() // run the rest of the chain
    ctx.Application().Logger().Debugf("%s %s took %s",
        ctx.Method(), ctx.Path(), time.Since(start))
}

func requireToken(ctx iris.Context) {
    if ctx.Header("Authorization") == "" {
        errors.Unauthenticated.Message(ctx, "missing Authorization header")
        return // no Next: the chain ends here
    }
    ctx.Next()
}

func main() {
    app := iris.New()
    app.Use(timing)
    app.Get("/private", requireToken, func(ctx iris.Context) {
        ctx.Text("secret")
    })
    app.Listen(":8080")
}
```

```
request --> timing --> requireToken --> handler
                                          |
response <-- timing <------------------ (return)
```

Not calling `ctx.Next()` ends the chain quietly; `ctx.StopExecution()` (and the `StopWith*` shortcuts) additionally marks the context stopped so that any later `Next` calls are ignored. `ctx.IsStopped()` reports that state.

Use, UseRouter, UseRelevant

Where you attach middleware determines which requests it sees. The three registration methods, verified against `core/router/router_builder.go`:

Method	On the root app	On a child router
`UseRouter`	Before routing: every request, including 404/405	Same, guarded by the child's path prefix (and subdomain)
`Use`	All matched routes, plus the 404 and 405 chains	Matched routes of that child and its children only
`UseRelevant`	Matched routes only, never 404/405	Same as `Use`

The root-versus-child difference in `Use` is the one that surprises people: `app.Use(m)` also runs `m` for requests that match no route at all, because the root's `Use` handlers are prepended to the no-route and no-method chains. On a child router, `Use` delegates to `UseRelevant`, so it affects only routes registered under that child. When you want root middleware that skips 404s, say so explicitly with `app.UseRelevant(m)`.

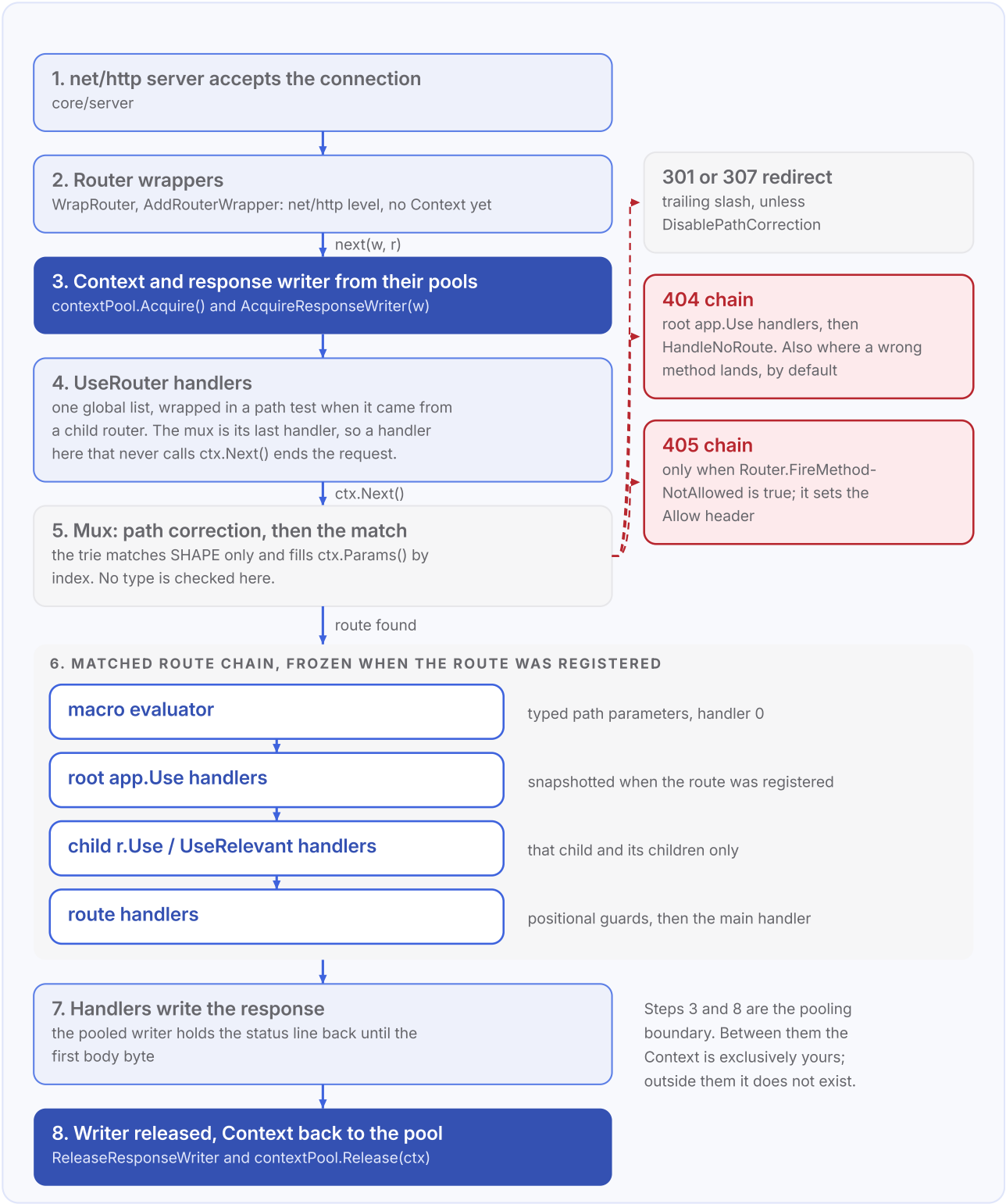
`UseRouter` runs even earlier: before the mux looks at the path. That makes it the right place for access logging, request IDs, rate limiting, and anything that must observe every request. From a child, `UseRouter` still registers globally but wraps the handler in a prefix check, so it fires only for requests under that child's path.

Two more variants exist: `UseOnce` and `UseRelevantOnce` insert a handler or replace an already-registered instance of it, which keeps middleware idempotent when setup code may run twice. And below all of this, `app.WrapRouter` wraps the entire router at the `net/http` level for integrations that need raw `http.Handler` semantics.

Order matters: a route's chain is assembled when the route is registered, so register middleware before the routes it should affect.

The Request Lifecycle

From accepted connection to released context:



A request passes the router wrappers, borrows a Context from the pool, runs the UseRouter chain into the mux, and either branches to a redirect or the 404 and 405 chains or enters the matched route's frozen chain before the Context is reset and returned to the pool.

Step 3 and step 8 are the pooling boundary. Between them the `Context` is exclusively yours; outside them it does not exist from your code's point of view. Step 5 is pure matching against the compiled mux: the trie compares segment count and static segments, and a typed

parameter such as `{id:int}` is *not* checked here. That check is a handler the router prepends to the matched route's own chain, which is why it appears as handler 0 of step 6. No route's handlers run before the router has decided, which is what makes the `UseRouter` versus `Use` distinction meaningful.

Graceful shutdown belongs to the same lifecycle at application scale: `app.Shutdown(ctx)` stops the server and runs everything registered with `app.AddCloser`. Deployment patterns around it are covered in Chapter 19.

Where Dependency Injection Sits

Handlers keep the single `func(ctx iris.Context)` signature; dependencies do not appear as handler parameters. Instead, constructors declare what they need, and the router resolves those constructors from registered services when routes are set up: at startup, never per request.

The application-level primitives:

```go
package main

import (
    "log"

    "github.com/kataras/iris/v14"
)

type Greeter struct{ Prefix string }

func main() {
    app := iris.New()

    if err := app.RegisterDependency(&Greeter{Prefix: "Hello, "}); err != nil {
        log.Fatal(err)
    }

    // HandlerFunc resolves the function's parameters from the
    // registered dependencies and returns a plain iris.Handler.
    app.Get("/{name}", app.HandlerFunc(func(g *Greeter) iris.Handler {
        return func(ctx iris.Context) {
            ctx.Text(g.Prefix + ctx.Params().GetString("name"))
        }
    }))

    app.Listen(":8080")
}
```

The same mechanism powers the controller style you saw in the router section: `app.Router("/books", NewBooksAPI)` calls the constructor with arguments resolved from the registered services, then invokes the returned struct's `Routing(r iris.Router)` method once to register its routes:

```go
type BooksAPI struct {
    svc *library.Service
}

func NewBooksAPI(svc *library.Service) *BooksAPI {
    return &BooksAPI{svc: svc}
}

func (api *BooksAPI) Routing(r iris.Router) {
    r.Get("/", api.list)
    r.Get("/{id:uuid}", api.get)
}
```

In builder-based applications the services are registered through the `Services(...)` step of the `iris.NewBuilder` chain, and any service implementing `Init(ctx context.Context) error` is initialized once inside `Build()`, after injection. The full resolution rules (constructors, struct field binding, per-router containers) are the subject of Chapter 8.

Summary

- The **Application** (`iris.go`) owns lifecycle: build, serve, shutdown. Its routing API comes from the embedded root `*router.RouterBuilder`, and after `Build` it is a standard `http.Handler`.
- The **Context** is pooled per request and implements `context.Context`, so it flows into services and database calls. Its methods split into request reading, response writing, and flow control; `Header` reads the request while `SetHeader` / `ResponseHeader` handle the response.
- The **Router** is a builder at design time and a compiled mux at serve time. Groups come from `Router` / `Child`, unmatched requests run the `HandleNoRoute` / `HandleNoMethod` chains.
- **Middleware** placement is semantic: `UseRouter` runs before routing for every request, root `Use` covers all routes plus $^{404}/_{405}$, child `Use` (and `UseRelevant`) covers matched routes only.
- **Dependency injection** happens at startup: constructors such as `NewBooksAPI(svc)` are resolved from registered services, and handlers keep their plain signature.

With the model in place, the next chapter introduces the recommended way to assemble production applications: the compile-time checked builder chain.

Further Reading

- MDN: HTTP request methods: what each verb means and which are safe to repeat.
- MDN: HTTP response status codes: the full status-code catalog with plain-English meanings.
- MDN: HTTP headers: reference for the metadata lines on both sides of the exchange.
- RFC 9110: HTTP Semantics: the authoritative specification behind methods, headers and status codes.
- net/http (pkg.go.dev): the standard library server Iris builds on; after `Build` the application is one of its `http.Handler`s.
- sync.Pool (pkg.go.dev): the Go primitive behind context pooling.

CHAPTER 3

Building Applications with iris.NewBuilder

`iris.NewBuilder()` is the recommended way to assemble an Iris v14 application. It is a fluent chain of configuration steps whose order the compiler enforces: each step returns the interface of the next, so CORS cannot come after logging and controllers cannot be registered before services. Along the way the builder installs the production plumbing you would otherwise wire by hand: recovery, structured error handling, compression, access logging, a health endpoint, dependency injection and graceful shutdown. This chapter walks through every step, backed by what the builder actually does in `iris_builder.go`, and ends with a complete application you can run and test.

Background

What "wiring" an application means. Every server program has two phases: a short setup phase and a long serving phase. In the setup phase you read configuration (the values that differ per deployment: addresses, credentials, allowed origins), construct your services and their dependencies, register routes and middleware, and decide what must happen at shutdown. Wiring is that setup phase, and it is where most production bugs are born, not because any single step is hard, but because the steps depend on each other and their order matters.

Cross-cutting concerns. Some behavior belongs to every request rather than to any one handler: logging what happened, recovering from crashes, compressing responses, checking which websites may call you, formatting errors consistently. These are called cross-cutting concerns because they cut across all your routes. Handled naively, each one is a line you must remember in every handler; handled as middleware wired once at startup, they disappear from your handlers entirely.

Startup and shutdown order. Resources start in dependency order (the database connection opens before the service that queries it) and must stop in a sensible order too: stop accepting new requests, let in-flight ones finish, then close what startup opened. Forgetting the shutdown half leaks connections and loses buffered logs. A good wiring layer pairs every "open" with its "close" at the moment you write it.

The builder pattern. A builder is an object you configure through a chain of method calls and finish with one final call (here `Build()`) that produces the assembled product. The chain reads top to bottom like a checklist, which is the point: the whole application's wiring is one visible expression instead of registrations scattered across files. Iris adds a stricter twist: each step returns a different type that offers only the next legal steps, so skipping a step or calling them out of order is not a bug you find under traffic. It is code that does not compile. The class of mistake this removes is exactly the class that is hardest to test for: middleware registered after the routes it should protect, recovery forgotten, shutdown code never registered.

Why the Builder

A production HTTP API needs more than routes: panics must become logged 500 responses instead of dropped connections, errors need one consistent JSON shape, responses should compress, requests should be logged, shutdown must close what startup opened, and middleware must run in the right order. None of that is hard individually; getting the order right every time is. The builder encodes the order once, in the framework, and gives you named steps for the decisions that are actually yours: which origins to allow, what to log, how to map your errors, which services to inject.

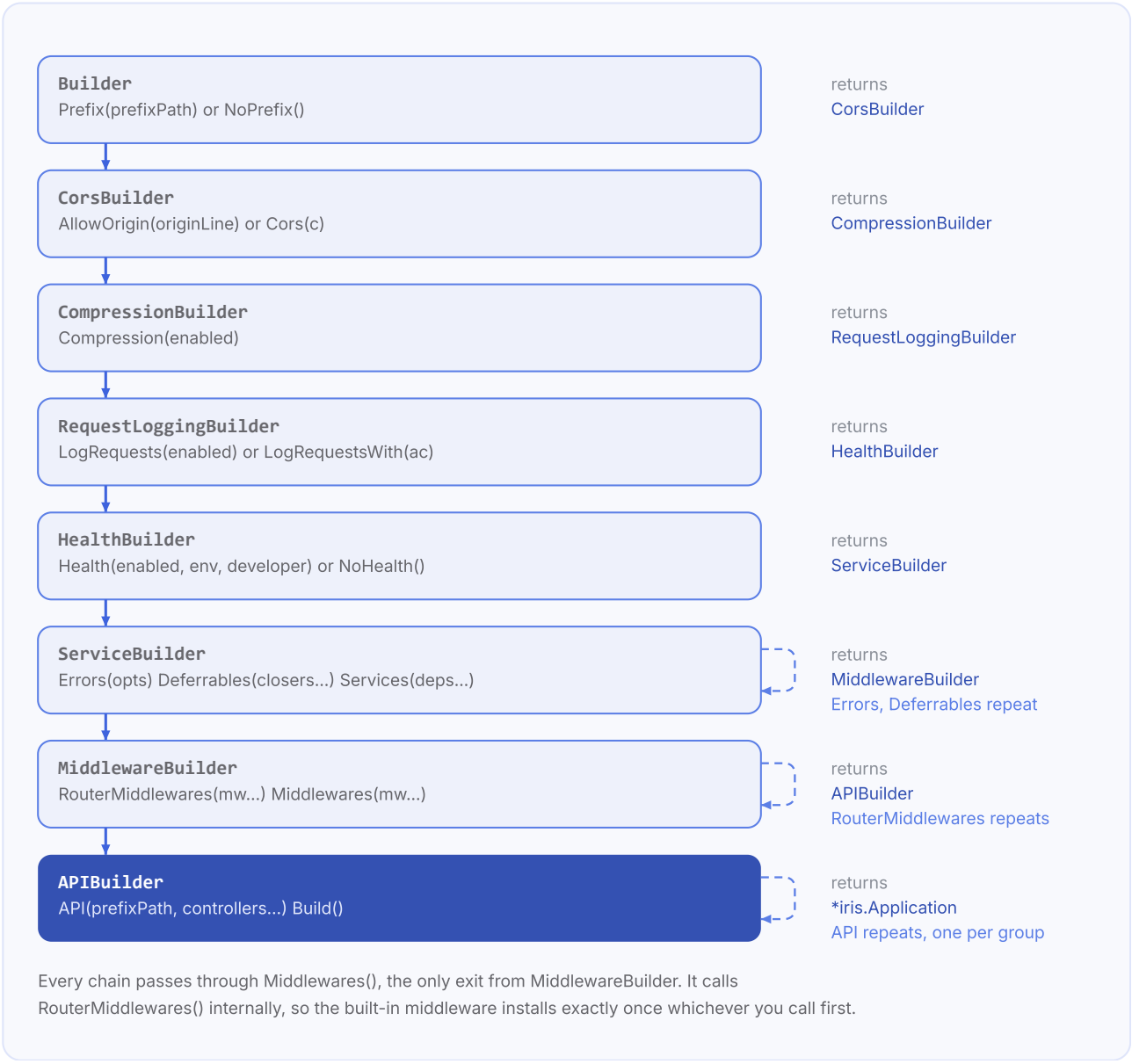
The result is that a whole application reads as a single expression:

```go
iris.NewBuilder().
    Prefix("/api").
    AllowOrigin("https://example.com").
    Compression(true).
    LogRequests(true).
    Health(true, "production", "team@example.com").
    Errors(errors.NewOptions().
        MapErrors(errors.NotFound, ErrNoteNotFound)).
    Deferrables(func() { log.Println("bye") }).
    Services(store, NewNoteService).
    RouterMiddlewares().
    Middlewares().
    API("/notes", NewNotesAPI).
    Build().
    Listen(":8080")
```

`iris.NewBuilder(configurators ...iris.Configurator)` creates the underlying application (any configurators you pass apply first) and registers the `*iris.Application` itself as a dependency, so controllers and custom components can receive it through injection.

The Step Chain

Each builder method returns a narrower interface that exposes only the next legal steps:



Each builder step returns the interface for the next one, so the compiler walks you through the chain in order; four steps return their own interface and may be repeated, and Build ends the chain with an application.

Three of the stages allow repetition. `Errors` and `Deferrables` return `ServiceBuilder` again, so you may call them in either order or several times before `Services` advances the chain. `RouterMiddlewares` returns `MiddlewareBuilder`, so it can be called repeatedly before `Middlewares` advances. `API` returns `APIBuilder`, so you mount as many groups as you need before the terminal `Build`.

The order is not decorative. Because each step returns a different type, skipping a stage is a compile error, not a runtime surprise. This is also why real applications contain the two seemingly empty calls:

```go
Services(store).
RouterMiddlewares(). // nothing custom, still required to move on
Middlewares().       // nothing custom, still required to move on
API("/notes", NewNotesAPI)
```

`Services` returns `MiddlewareBuilder`, and the only way out of `MiddlewareBuilder` is `Middlewares()`, so every chain passes through it even with no arguments. The calls are not no-ops either: the first of them triggers the builder's one-time middleware setup described in RouterMiddlewares and Middlewares. Strictly speaking `RouterMiddlewares()` could be omitted, because `Middlewares()` invokes it internally, but writing both keeps the two middleware layers visible at the call site.

One consequence of the changing return types: you cannot hold the chain in a single variable and reassign it across steps. Compute conditional values first, then build in one uninterrupted expression:

```go
origin := "https://example.com"
if env == "development" {
    origin = "*"
}

app := iris.NewBuilder().
    Prefix("/api").
    AllowOrigin(origin).
    Compression(env == "production").
    LogRequests(env != "production").
    Health(true, env, "team@example.com").
    Services().
    Middlewares().
    Build()
```

Step Reference

Prefix and NoPrefix

`Prefix("/api")` creates a child router with `app.Router("/api")` and points every later step at it: CORS, access logging, your middlewares and all `API()` groups mount under that prefix. Use it for versioning (`/api/v1`) or to keep the API apart from static content. `NoPrefix()` targets the application root instead; it is the same as `Prefix("")`.

The one deliberate exception is the health endpoint, which always registers at the root (see below).

AllowOrigin

CORS (Cross-Origin Resource Sharing) is the browser's permission system for cross-site calls: JavaScript served from `https://app.com` may not call an API on a different origin unless that API's responses explicitly allow it in headers. For some requests the browser first sends a preflight (an automatic `OPTIONS` request that asks "may I?") and proceeds only when the answer permits. The middleware writes those headers and answers preflights for you; your only decision is which origins to allow.

`AllowOrigin` configures the CORS middleware. The argument is trimmed and lowercased, then routed:

- `"none"`: no CORS middleware is installed at all.
- `"*"`: allow every origin (public, credential-free APIs only).
- `"https://app.com"`: exactly one origin.
- `"https://app.com,https://admin.app.com"`: a comma-separated list.

When enabled, the builder registers `cors.New().HandleErrorFunc(errors.FailedPrecondition.Err).AllowOrigin(originLine).Handler()` with `UseRouter`, so it runs before routing and handles preflight requests for every path under the prefix. Wiring `HandleErrorFunc` to the errors package means a disallowed origin is rejected with the same structured JSON error body as the rest of your API, instead of an anonymous plain-text response.

Cors

`AllowOrigin` covers the common case in one string. Everything else the middleware can do (credentials, exposed headers, max age, the origin extractor, a different error handler) needs the CORS value itself, which is what the alternative step at this stage takes:

```go
iris.NewBuilder().
    Prefix("/api").
    Cors(cors.New().
        HandleErrorFunc(errors.FailedPrecondition.Err).
        AllowOrigins("https://app.com", "https://*.admin.com").
        ExposeHeaders("X-Total-Count").
        AllowCredentials()).
```

```
    Compression(true).
    // ...
```

The two steps are the same call underneath: `AllowOrigin(line)` is the shorthand for `Cors(cors.New().HandleErrorFunc(errors.FailedPrecondition.Err).AllowOrigin(line))`. Both register through `UseRouter` at the same position in the chain, so preflights are answered before routing either way. Pass `nil` and no middleware is installed, the same as `AllowOrigin("none")`.

Your value is used exactly as given, with no defaults merged into it. That one difference matters: `cors.New()` on its own carries the middleware's default error handler, so a `Cors` step that omits `HandleErrorFunc` rejects disallowed origins with a bare 403 rather than the structured error body `AllowOrigin` installs. Set it yourself when you want the two to behave alike.

`AllowCredentials` is worth a note. A credentialed cross-origin request (one carrying cookies or an `Authorization` header) is only permitted when the response says `Access-Control-Allow-Credentials: true`, and browsers refuse that header when the allowed origin is the wildcard `*`. The middleware follows the browser rule: with a wildcard origin it suppresses the credentials header unless you called `AllowCredentials` explicitly. In practice a browser always sends `Origin` and the middleware echoes that origin back rather than `*`, so credentials are advertised normally; the wildcard case arises for requests that carry no `Origin` header at all, where `AllowCredentials` is the way to opt in.

Compression

Compression shrinks response bodies before they cross the network (JSON and HTML typically compress to a fraction of their size), trading a little CPU for less bandwidth and faster responses on slow links. Client and server agree on the algorithm through headers, which is the negotiation described below.

`Compression(true)` only records a flag; the actual `middleware/compress` handler is installed later, during the one-time middleware setup, and only if a handler named `iris.compression` is not already registered. The middleware negotiates the response encoding from the client's `Accept-Encoding` header (gzip, deflate, br, snappy, s2, zstd) and also decompresses request bodies that arrive with a `Content-Encoding` header, which is the "read" half of the parameter's `enableReadWriteCompression` name. Pass `false` while debugging if you want to read raw response bodies on the wire.

LogRequests and LogRequestsWith

`LogRequests(true)` is a shortcut for `LogRequestsWith(ac)` with the framework's tuned `middleware/accesslog` configuration:

- JSON lines, indented, with human-readable timestamps, written to `access_log.json` and mirrored to stdout.
- Per request: latency, status code, method, path, client IP, request and response bodies, and their byte sizes.
- Panics are logged too (`PanicLog`), with multi-line errors preserved.
- Synchronous writes (`Async = false`), so nothing is lost on crash.

The handler registers with `UseRouter` on the builder's router, and its `Close` method is added as an application closer, so buffered output is flushed on shutdown. `LogRequests(false)` installs nothing.

When the defaults do not fit, build your own access log and pass it at the same position; a `nil` value is ignored:

```go
ac := accesslog.File("access_log.json")
ac.RequestBody = false // do not log request payloads
ac.AddFields(func(ctx iris.Context, f *accesslog.Fields) {
    f.Set("tenant", ctx.Header("X-Tenant-ID"))
})

// ... .Compression(true).LogRequestsWith(ac). ...
```

Either way, add `access_log.json` to `.gitignore`.

Health and NoHealth

A health endpoint is a URL whose only job is to answer "is this process alive, and which build is it?". Monitoring systems poll it and alert, restart or reroute traffic when it stops answering.

`Health(true, env, developer)` registers `GET /health` at the application root, intentionally outside the prefix: load balancers, Kubernetes probes and uptime monitors ask for `/health`, not `/api/v1/health`. The handler comes from `middleware/modrevision` and responds with plain text containing the server name and environment (`Iris Server (production)`), the build revision and build time when the binary carries module build info, the developer line, detected cloud container information, and a final `OK`.

The `developer` string does double duty: the builder remembers it and, during the one-time middleware setup, installs a handler that sets a `Server: Iris` header on every response plus an `X-Developer` header when the string is non-empty. That makes it trivial to tell from any response which team owns a deployment. `NoHealth()` skips the endpoint and leaves the developer empty, so no `X-Developer` header is emitted.

Errors

`Errors(*errors.Options)` accepts the central error map from `github.com/kataras/iris/v14/rest/errors`. The builder stores the options and, during the one-time middleware setup, registers `opts.Handler()` with `UseRouter`; the handler attaches your options to each request context, so every error helper in the framework consults your map when it renders a response.

`errors.NewOptions()` starts from useful defaults, sending `database/sql.ErrNoRows` to `NOT_FOUND` and the `kataras/jwt` token errors to `UNAUTHENTICATED`. You extend it with your own mappings:

```go
Errors(errors.NewOptions().
    MapErrors(errors.NotFound, ErrNoteNotFound).
    MapErrors(errors.PermissionDenied, ErrArchiveLocked).
    MapErrorFunc(errors.Internal, func(err error) error {
        return errors.Private(err) // hide details from clients
    }))
```

`MapErrors` matches with `errors.Is` semantics, `MapErrorFunc` and the context-aware `MapErrorFuncCtx` transform errors, and `MapErrorTypes` matches by type. The default rendering is a structured JSON object; chain `SetRenderer(errors.Problem)` for RFC 7807 Problem Details output. Chapter 9 covers the error system in full.

Deferrables

`Deferrables(closers ...any)` registers shutdown work. Every argument is forwarded to `app.AddCloser`, which accepts exactly these shapes and panics at registration time on anything else:

- `func()`
- `func() error` (the error is preserved and joined into the shutdown error with `errors.Join`)
- `func(context.Context)`
- `func(context.Context) error`
- `io.Closer` (anything with `Close() error`)
- the context-aware `Closer` from `core/server/closer` (`CloseWithContext(context.Context) error`)

Closers run in registration order when the application shuts down through `app.Shutdown(ctx)`. The built-in interrupt handler drains in-flight requests but does not run them by itself, so register the shutdown explicitly if you want closers on Ctrl+C:

```go
iris.RegisterOnInterrupt(app.Shutdown)
```

Registered this way, `app.Shutdown` receives a context whose deadline is the `ShutdownOnInterruptTimeout` of the running server: 15 seconds from `iris.Timeout()`, and when no `Timeout` is set at all. The drain and every closer that takes a context get that same deadline, counted from the signal, so a drain that uses all of it leaves the closers a context that has expired already. Zero or a negative value means no deadline. For a budget of your own, register a function that calls `app.Shutdown` with its own context, as Chapter 19 does.

The server's own shutdown is prepended to the closer list, so the requests in flight drain first and the remaining closers run after them. `Listen` and `Run` return once `app.Shutdown` has run the whole list, not when the server alone has drained, so a `main` that ends with `Listen` lets every closer finish. For the same reason a closer must not wait for `Listen` or `Run` to return: each would be waiting for the other, and the shutdown blocks for good. After a signal the process exits when the interrupt functions are done, which is also when `Listen` returns: statements placed after it may not run.

```go
Deferrables(
    func() { worker.Stop() },
    func() error { return cache.Flush() },
    logFile, // io.Closer
)
```

Services and the Initializer Hook

`Services(deps ...any)` feeds the dependency injection container that later constructs your middlewares and controllers. Each argument is routed by its shape:

Argument shape	What the builder does
`nil`	Skipped.
`func()` or `func() error`	Added as a shutdown closer only; a function with no outputs cannot be a constructor.
Value with `Close()` or `Close() error`	Registered as a dependency and as a shutdown closer.
Value implementing `iris.Worker`	Registered as a dependency, started after the initializers; at shutdown it is stopped and waited for before its own `Close`, in registration order with the other closers.
`iris.Configurator`	Applied to the application immediately; not a dependency.
Anything else	Registered as a dependency: values by their type, functions as constructors.

Constructor functions are called lazily with their parameters resolved from previously registered dependencies, which is why order matters inside a `Services` call:

```go
Services(
    store,          // *NoteStore value (has Close: also a closer)
    NewNoteService, // func(*NoteStore) *NoteService
)
```

If registration fails, `Services` panics with the underlying error. That is deliberate: a broken dependency graph should stop the process at startup, not surface as confusing behavior under traffic.

Services also get a startup hook. Any registered singleton (a value you passed or a constructor's output) that implements `iris.Initializer` runs exactly once inside `Build()`, after dependency injection completes, in registration order:

```go
type Initializer interface {
    Init(ctx context.Context) error
}
```

Use it for cache warm-up, connection pinging or background pool setup. An `Init` error aborts the build, making the interface the startup counterpart of `Deferrables`.

Long-running work has its own interface. A scheduler, a queue consumer or a cache refresher passed to `Services` as an `iris.Worker` runs inside the same chain, instead of in a goroutine started next to it:

```go
type Worker interface {
    Run(ctx context.Context) error
    Stop(ctx context.Context) error
}
```

`Run` starts in its own goroutine after every `Initializer` has returned, with a context that is cancelled at shutdown. An error it returns before that is logged with the worker's type name and nothing else: a failed cache refresher should not take down a server that can still answer every request which does not need it.

`Stop` runs once, in registration order with the closers, and `Shutdown` then waits for `Run` to return, so a worker is really finished when `Shutdown` reports success rather than still draining while the process exits from under it. A value that is both a `Worker` and a closer is stopped, waited for, and only then closed, because closing the connection a running `Run` is still reading from is the crash that ordering avoids.

Both halves are bounded. They get the `Shutdown` context when that one carries a deadline, and a ten second budget when it does not, so a hanging `Stop` cannot hold the process open. That budget is *per worker*, not per shutdown, because the closers run one after another: pass a

deadline to `app.Shutdown` to bound the whole chain instead.

A `Worker` is a dependency like any other value, so the same scheduler can be injected into a controller that wants to trigger it. Only a `Worker` passed to `Services` as a value is started; one returned by a constructor is injected but never run, so hand the scheduler itself to `Services`, not its constructor.

RouterMiddlewares and Middlewares

Two of the built-ins are installed by `NewBuilder` itself, before the first step runs:

- the macro (path parameter) error handler, `errors.DefaultPathParameterTypeErrorHandler`, so a request like `/notes/abc` against `{id:int}` renders `400 INVALID_ARGUMENT` ("invalid path parameter") instead of a bare 404, and
- the context error handler, `errors.DefaultContextErrorHandler`, so failures inside response writers such as `ctx.JSON` render structured errors.

They go in that early for two reasons. `Prefix` creates the child router the moment it is called and copies the parent's context options into it, and a route keeps the macro error handler that was set when it was registered. The macros themselves are not copied: a child router shares the ones of its application. Both handlers stay in the builder's own application, so an `iris.New()` application in the same process is not affected by them.

These two methods separate the middleware layers, and the first call to either performs the builder's one-time setup, guarded by a `sync.Once`. That setup installs the pre-routing chain, in execution order:

1. Your `Errors` options handler, if you provided one. It goes first so everything after it, the CORS rejection included, is written through the collector, renderer and id generator you configured.
2. The access log, when `LogRequests` asked for one. It logs when `ctx.Next` returns rather than from a `defer`, so it has to sit outside recovery: otherwise a panicking request is logged nowhere.
3. The recovery middleware with `errors.Internal.LogErr` as its error handler: a panic becomes a logged, structured 500 response. Skipped when a handler named `iris.recovery` or `iris.errors.recover` is already registered.
4. CORS, below recovery so a panic inside its error handler is caught, and below the access log so a rejection is recorded.
5. The `Server: Iris` header on every response, plus `X-Developer` when `Health` received a developer string.

6. `compress.Safe`, when `Compression(true)` was chosen. `Safe` rather than `Handler`: an unbounded decompressor turns a one-megabyte gzip body into a gigabyte of memory.
7. `errors.Handler` from `rest/errors`, the last of the built-in router middleware and after compression, so what it writes is compressed like any other body. A failure status that reaches the client with nothing written, from a route or from any middleware registered after this point, answers the JSON envelope of Chapter 9 instead of an empty body: `ctx.StopWithStatus(403)`, `ctx.NotFound()`, the 401 of the `jwt` middleware.
8. The rate-limit exceed handler, which is a binding rather than a middleware in the chain: responses from `middleware/rate` render as the errors package's JSON instead of empty 429s.

When the application is built, the builder also registers `errors.Handler` as the 404 and the 405 handler, so an unknown path answers `NOT_FOUND` as JSON and not the text `Not Found`. It fills those two slots only when they are empty: a `HandleNoRoute` or `HandleNoMethod` of your own, registered from a `Configurator` passed to `NewBuilder`, wins. Two rejections are answered before `errors.Handler` runs and keep their bare status: one from a handler a `Configurator` registered with `UseRouter`, and the 415 of the compression middleware for a request body that is not what its `Content-Encoding` says.

One consequence of that order is worth stating plainly: a panic inside the access log or inside your own `Errors` options handler is above recovery, so it is not recovered. Both are framework-adjacent code you register once; keep them simple.

After the built-ins, your own entries register:

- `RouterMiddlewares(...)` uses `UseRouter`: the handlers run before routing, for every request that reaches the builder's router, including requests that end in 404. Use it for request IDs, security headers, metrics and global rate limiting.
- `Middlewares(...)` uses `Use`: the handlers run only for matched routes. Use it for authentication and anything that assumes a real route.

Both accept plain `iris.Handler` values as well as constructor functions whose parameters resolve from registered services:

```go
RouterMiddlewares(func(ctx iris.Context) {
    ctx.SetHeader("X-Frame-Options", "DENY")
    ctx.Next()
}).
Middlewares(func(auth *AuthService) iris.Handler {
    return func(ctx iris.Context) {
        user, err := auth.UserFromRequest(ctx)
        if err != nil {
            errors.Unauthenticated.Err(ctx, err)
            return
```

```
        }
        ctx.Values().Set("user", user)
        ctx.Next()
    }
})
```

`Middlewares` calls `RouterMiddlewares()` internally before doing its own work, so the built-ins install exactly once regardless of which method you call first or how many times.

API

`API(prefixPath string, controllers ...any)` mounts a route group under the builder's prefix by delegating to `router.Router(prefixPath, ...)`. A controller is anything with a `Routing(r iris.Router)` method; you usually pass its constructor and let injection supply the dependencies, as the complete example below shows. Plain handlers listed before a controller act as middleware for that group only:

```go
API("/notes", NewNotesAPI).
API("/admin", basicauth.Default(map[string]string{
    "admin": adminPassword,
}), NewAdminAPI).
```

Call `API` as many times as you have groups. Controllers, grouping and the helper controllers (`iris.Middleware`, `iris.Dependencies` and friends) are the subject of Chapter 4.

Build

`Build()` finalizes the application: it runs the `iris.Initializer` hooks of your services (once), then compiles the router and middleware stack. On any misconfiguration it panics; like the `Services` panic, this is intentional so that a broken application never starts serving.

It returns the `*iris.Application`, which you can serve or test:

```go
app := builder.Build()

// Plain HTTP.
app.Listen(":8080")

// Or with explicit server options (TLS shown; see Chapter 19).
app.Run(iris.ServerOptions{
    Addr:     ":443",
    CertFile: "cert.pem",
    KeyFile:  "key.pem",
})
```

A Complete Application

The following program is a small but production-shaped notes API. It is a single file here for readability; the layering (config, domain, storage, controllers, wiring) is exactly what you would split into packages as the application grows. The code compiles and runs as shown.

GO

```go
package main

import (
    "context"
    "fmt"
    "log"
    "os"
    "sync"

    "github.com/kataras/iris/v14"
    "github.com/kataras/iris/v14/middleware/basicauth"
    "github.com/kataras/iris/v14/rest/errors"
)

// Config carries the deployment-specific values. iris.BaseConfiguration
// brings Host, Port, AllowOrigin, EnableCompression, RequestLog and the
// framework options, so cfg.Addr() feeds Listen and there is no Addr field
// of our own to keep in step. Environment variables keep the example short;
// Chapter 11 covers configuration in depth, including loading this same
// struct from a YAML document with iris.LoadConfiguration.
type Config struct {
    iris.BaseConfiguration `yaml:",inline"`

    Env       string `yaml:"Env"`
    Developer string `yaml:"Developer"`
    AdminUser string `yaml:"AdminUser"`
    AdminPass string `yaml:"-"`
}

func loadConfig() (Config, error) {
    cfg := Config{
        BaseConfiguration: iris.BaseConfiguration{
            Port:              8080,
            AllowOrigin:       "none",
            EnableCompression: true,
        },
        Env:       "development",
        Developer: "notes-team",
        AdminUser: "admin",
        AdminPass: os.Getenv("ADMIN_PASSWORD"),
    }

    // HOST, PORT, HTTP_PLATFORM_PORT and ALLOW_ORIGIN override the
    // defaults above. iris.LoadConfiguration calls this for you when the
    // configuration arrives as a YAML document.
    cfg.BindEnv()

    if cfg.AdminPass == "" {
        return cfg, fmt.Errorf("ADMIN_PASSWORD is required")
    }

    return cfg, nil
}

// ErrNoteNotFound is a sentinel error; the builder's Errors step maps
// it to a 404 response.
```

```
var ErrNoteNotFound = fmt.Errorf("note not found")

type Note struct {
    ID   int    `json:"id"`
    Text string `json:"text"`
}

// NoteInput is the request payload. Implementing Validate makes
// ctx.BindJSON run it automatically after decoding.
type NoteInput struct {
    Text string `json:"text"`
}

func (in NoteInput) Validate() error {
    if in.Text == "" {
        return fmt.Errorf("text: must not be empty")
    }
    return nil
}
```

The storage type demonstrates two builder behaviors at once: it has a `Close` method, so `Services` registers it as a shutdown closer automatically, and it implements `iris.Initializer`, so `Init` runs once inside `Build()`.

```
type NoteStore struct {
    mu    sync.Mutex
    seq   int
    notes []Note
}

func NewNoteStore() *NoteStore { return &NoteStore{} }

func (s *NoteStore) Init(ctx context.Context) error {
    log.Println("notes: store ready") // ping a real database here.
    return nil
}

func (s *NoteStore) Close() error {
    log.Println("notes: store closed")
    return nil
}

func (s *NoteStore) List(ctx context.Context) ([]Note, error) {
    s.mu.Lock()
    defer s.mu.Unlock()
    return append([]Note(nil), s.notes...), nil
}

func (s *NoteStore) Create(ctx context.Context, in NoteInput) (Note, error) {
    s.mu.Lock()
    defer s.mu.Unlock()

    s.seq++
    note := Note{ID: s.seq, Text: in.Text}
    s.notes = append(s.notes, note)
    return note, nil
}

func (s *NoteStore) Get(ctx context.Context, id int) (Note, error) {
```

```
    s.mu.Lock()
    defer s.mu.Unlock()

    for _, note := range s.notes {
        if note.ID == id {
            return note, nil
        }
    }
    return Note{}, fmt.Errorf("%w: %d", ErrNoteNotFound, id)
}
```

Two controllers: the public notes API and an admin API mounted behind basic authentication. (Basic authentication is the simplest HTTP login scheme: the client repeats a username and password in the `Authorization` header of every request. Served over TLS, it is a reasonable fit for small internal admin endpoints.)

GO

```
type NotesAPI struct {
    store *NoteStore
}

func NewNotesAPI(store *NoteStore) *NotesAPI {
    return &NotesAPI{store: store}
}

func (api *NotesAPI) Routing(r iris.Router) {
    r.Get("/", api.list)
    r.Post("/", api.create)
    r.Get("/{id:int}", api.get)
}

func (api *NotesAPI) list(ctx iris.Context) {
    notes, err := api.store.List(ctx)
    ctx.List(notes, err) // an empty store renders [], never null.
}

func (api *NotesAPI) create(ctx iris.Context) {
    input, ok := ctx.BindJSON[NoteInput]()
    if !ok {
        return // 400 already written: bad JSON or failed Validate.
    }

    note, err := api.store.Create(ctx, input)
    ctx.Created(note, err)
}

func (api *NotesAPI) get(ctx iris.Context) {
    note, err := api.store.Get(ctx, ctx.Params().GetInt("id"))
    ctx.OK(note, err) // ErrNoteNotFound renders as 404.
}

type AdminAPI struct {
    store *NoteStore
}

func NewAdminAPI(store *NoteStore) *AdminAPI {
    return &AdminAPI{store: store}
}
```

```
func (api *AdminAPI) Routing(r iris.Router) {
    r.Get("/stats", api.stats)
}

func (api *AdminAPI) stats(ctx iris.Context) {
    notes, err := api.store.List(ctx)
    if err != nil {
        errors.HandleError(ctx, err)
        return
    }
    ctx.JSON(iris.Map{"notes": len(notes)})
}
```

Finally the wiring. `newApp` takes its storage as a parameter, so tests can build the identical application around any store they like:

GO

```
// newApp assembles the application. Storage arrives as a parameter:
// main passes the real store, tests pass whatever they need.
func newApp(cfg Config, store *NoteStore) *iris.Application {
    return iris.NewBuilder(cfg.Iris).
        Prefix("/api").
        AllowOrigin(cfg.AllowOrigin).
        Compression(cfg.EnableCompression).
        LogRequests(cfg.Env == "development").
        Health(true, cfg.Env, cfg.Developer).
        Errors(errors.NewOptions().
            MapErrors(errors.NotFound, ErrNoteNotFound)).
        Deferrables(func() { log.Println("notes: goodbye") }).
        Services(store).
        RouterMiddlewares().
        Middlewares().
        API("/notes", NewNotesAPI).
        API("/admin", basicauth.Default(map[string]string{
            cfg.AdminUser: cfg.AdminPass,
        }), NewAdminAPI).
        Build()
}

func main() {
    cfg, err := loadConfig()
    if err != nil {
        log.Fatal(err)
    }

    app := newApp(cfg, NewNoteStore())
    app.Listen(cfg.Addr())
}
```

Run it and exercise the behaviors the builder gave you for free:

SH

```
ADMIN_PASSWORD=change-me go run .



curl -X POST localhost:8080/api/notes \
  -H 'Content-Type: application/json' -d '{"text":"hello"}'
```

```
curl localhost:8080/api/notes/42


curl localhost:8080/api/notes/abc


curl localhost:8080/api/admin/stats
curl -u admin:change-me localhost:8080/api/admin/stats


curl localhost:8080/health
```

Testing the Built Application

`Build()` returns a fully assembled `*iris.Application`, and `httptest.New` drives it in memory: real router, real middleware, real error map, no network. Because `newApp` accepts its dependencies, a test constructs the exact production application around a test store:

GO

```go
package main

import (
    "testing"

    "github.com/kataras/iris/v14"
    "github.com/kataras/iris/v14/httptest"
)

func testConfig() Config {
    return Config{
        BaseConfiguration: iris.BaseConfiguration{AllowOrigin: "none"},
        Env:               "test",
        Developer:         "tests",
        AdminUser:         "admin",
        AdminPass:         "test-password",
    }
}

func TestNotesAPI(t *testing.T) {
    app := newApp(testConfig(), NewNoteStore())
    e := httptest.New(t, app)

    e.POST("/api/notes").WithJSON(NoteInput{Text: "hello"}).
        Expect().Status(httptest.StatusCreated).
        JSON().Object().HasValue("id", 1)

    e.GET("/api/notes/42").Expect().Status(httptest.StatusNotFound)

    e.GET("/api/admin/stats").WithBasicAuth("admin", "test-password").
        Expect().Status(httptest.StatusOK).
        JSON().Object().HasValue("notes", 1)
}
```

Chapter 16 expands on this pattern.

When to Use iris.New Instead

`iris.New()` creates a bare application: no recovery, no error handling, nothing installed for you. It remains the right tool for small programs and for setups where you want to control every registration yourself:

```go
app := iris.New()
app.UseRouter(recovery.Handler)
app.Get("/", func(ctx iris.Context) {
    ctx.JSON(iris.Map{"message": "hello"})
})
app.Listen(":8080")
```

For quick prototypes there is also `iris.Default()`, which is itself implemented as a builder chain, a good indication of how idiomatic the builder is:

```go
iris.NewBuilder().
    NoPrefix().
    AllowOrigin("*").
    Compression(true).
    LogRequests(false).
    NoHealth().
    Services().
    RouterMiddlewares(requestid.New()).
    Middlewares()
```

plus debug-level logging. Note the missing `Build()`: `iris.Default()` hands back an application that has not been built, so routes registered after the call are still served, exactly as with `iris.New()`. For anything you intend to deploy, start from `iris.NewBuilder()`.

Summary

- `iris.NewBuilder()` assembles a production-ready application in one expression; each step returns the next step's interface, so the order is checked by the compiler.
- The chain is `Prefix` / `NoPrefix`, `AllowOrigin` (`"none"` disables) or `Cors` for a fully configured CORS value, `Compression`, `LogRequests` / `LogRequestsWith`, `Health` / `NoHealth` (always `GET /health` at the root), then `Errors`, `Deferrables` and `Services`, then `RouterMiddlewares` / `Middlewares`, then `API` groups and `Build`.

- The first middleware call installs the built-ins once: path parameter error handling, your error options, recovery, the context error handler, `Server` / `X-Developer` headers, compression, `errors.Handler` and the rate-limit exceed handler. An unknown path, a wrong method and a failure status with no body all answer the `rest/errors` JSON envelope. Empty `RouterMiddlewares().Middlewares()` calls exist because the type chain must pass through them.
- `Services` routes arguments by shape: zero-output functions become closers, values with `Close` become dependency plus closer, configurators apply immediately, `nil` is skipped, and registration failures panic at startup.
- Services implementing `iris.Initializer` run `Init(ctx)` once inside `Build()`, after injection, in registration order; an error aborts the build. `Deferrables` is the mirror image at shutdown.
- `Build()` panics on misconfiguration by design, and returns the `*iris.Application` for `Listen`, `Run` or `httptest`.

Further Reading

- MDN: Cross-Origin Resource Sharing (CORS): how browsers police cross-site calls and what a preflight request is.
- MDN: Compression in HTTP: how clients and servers negotiate response encoding with `Accept-Encoding` and `Content-Encoding`.
- MDN: HTTP authentication: the Basic scheme used by the admin API example, and its relatives.
- RFC 7807: Problem Details for HTTP APIs: the standard error-body format the `errors.Problem` renderer emits.
- context (pkg.go.dev): the standard package behind `Init(ctx)`, `Shutdown(ctx)` and cancellation throughout the book.

CHAPTER 4

Routing & Controllers

An Iris route maps an HTTP method and a URL path to a chain of handlers. This chapter covers the registration API, typed path parameters, grouping with child routers, the constructor-based controller pattern that scales from a single file to a full service, error-returning handlers through `rest/errors`, subdomains, and route naming and introspection.

Background

The shape of a URL. A URL such as `https://example.com/api/v1/users/42?page=2&sort=name` splits into parts the router cares about differently. The scheme (`https`) and host (`example.com`) select the server. The path (`/api/v1/users/42`) is a list of segments separated by slashes and names the resource being acted on. The query string (everything after the `?`, here `page=2&sort=name`) is a set of optional key-value pairs, conventionally used for filtering, paging and sorting rather than for identity. Routing decides on method and path only; the query string never chooses the handler, it only tunes what the handler does.

Static and dynamic segments. `/users/42` and `/users/97` should run the same code, so a route pattern mixes fixed segments (`users`) with placeholders (`{id}`) that match whatever appears in that position. The placeholder's value is handed to your handler as a parameter.

What typed parameters buy you. Without them, every handler opens with the same ritual: pull the raw string out of the path, convert it to a number or date, check the conversion error, answer 400 or 404, and only then do the real work. A typed parameter such as `{id:int}` moves that ritual into the route table: the router validates and converts before any handler of yours runs, so a handler that runs can trust its input completely. Multiplied across every parameter of every route, the win is a whole class of "forgot to check" bugs that can no longer be written, and validation rules that sit in one visible place (the route) instead of scattered through function bodies.

What grouping and controllers organize. Real APIs have shared structure: everything under `/api/v1` shares that prefix, everything under `/admin` requires an administrator, every route about books needs the book service. A group captures the path-level sharing once (a common prefix plus common middleware), so individual routes stay one line each. A controller goes a step further and organizes the code: one struct owns a family of related handlers together with the dependencies they share, and declares its routes in one method. Groups organize paths; controllers organize code around those paths.

Route Registration

Every router (the application itself or any child created with `Router`) exposes one function per HTTP method. A handler is a `func(ctx iris.Context)` (the `iris.Handler` type); when you pass more than one, they run in order and the last is the main handler.

```go
package main

import "github.com/kataras/iris/v14"

func main() {
    app := iris.New()

    app.Get("/", index)                    // GET /
    app.Post("/users", createUser)         // POST /users
    app.Put("/users/{id:int}", updateUser) // PUT /users/42
    app.Delete("/users/{id:int}", deleteUser)

    // One handler for several methods, space or comma separated.
    app.Handle("GET POST", "/contact", contact)

    // Every HTTP method.
    app.Any("/ping", ping)

    app.Listen(":8080")
}

func index(ctx iris.Context) {
    ctx.JSON(iris.Map{"message": "hello"})
}
```

The method functions (`Get`, `Post`, `Put`, `Delete`, `Patch`, `Head`, `Options`, `Connect`, `Trace`, `Query`) return a `*Route`, so registration can be chained with route configuration such as `SetName`. `Handle` and `Any` create one route per method and return a `*RouteRegistry` holding all of them.

`Query` is HTTP QUERY (RFC 10008): a safe, idempotent read that carries a body, the natural method for a search or a filtered list. `iris.MethodQuery` is its name (`"QUERY"`) wherever a method string is expected, as in `app.Handle("POST QUERY", "/list", list)` , and `Any` includes it. When the router redirects a request to its trailing-slash twin, POST, PUT, PATCH and QUERY get a 307 so the body survives the redirect.

There is no optional-parameter syntax. When a path segment may be absent, register two routes that share the same handler:

```go
app.Get("/search", search)
app.Get("/search/{term}", search)
```

Path Parameters

A dynamic path segment is written as `{name}` or `{name:type}` . The type is a *macro*: it validates the raw segment before your handler runs and converts it to a Go value. Without a type, `{name}` means `{name:string}` .

One type in the table deserves a definition: a UUID (universally unique identifier) is a 128-bit identifier written as 36 hexadecimal characters, such as `9f86d081-8292-4c4c-812f-0a1b2c3d4e5f` . Because a freshly generated UUID is unique for all practical purposes, services use them as record IDs that any machine can mint without coordinating with a central counter, which is why they appear in this book's examples wherever a database ID crosses a URL.

Built-in Parameter Types

Type	Syntax	Go type	Matches
`string`	`{name}` or `{name:string}`	`string`	any single path segment (default)
`int`	`{id:int}`	`int`	integers (alias: `number`)
`int8` .. `int64`	`{n:int64}`	`int8` .. `int64`	sized integers (`int64` alias: `long`)
`uint` .. `uint64`	`{n:uint64}`	`uint` .. `uint64`	unsigned integers
`bool`	`{v:bool}`	`bool`	`strconv.ParseBool` values: 1, 0, t, f, true, false, ... (alias: `boolean`)
`alphabetical`	`{name:alphabetical}`	`string`	letters and spaces only
`file`	`{name:file}`	`string`	letters, digits, underscore, dash, dot; no spaces
`path`	`{rest:path}`	`string`	multi-segment wildcard; must be the last segment

`uuid`	`{id:uuid}`	`uuid.UUID`	any RFC 9562 UUID, whatever its version (alias kept for compatibility: `uuidv4`)
`mail`	`{addr:mail}`	`string`	e-mail address, syntax check only
`email`	`{addr:email}`	`string`	e-mail address, syntax plus MX domain lookup
`date`	`{d:date}`	`time.Time`	date in YYYY/MM/DD format
`weekday`	`{day:weekday}`	`time.Weekday`	day name or 0-6

Constraints

Constraints follow the type, separated by spaces. A trailing `else N` sets the HTTP status code the router responds with when validation fails.

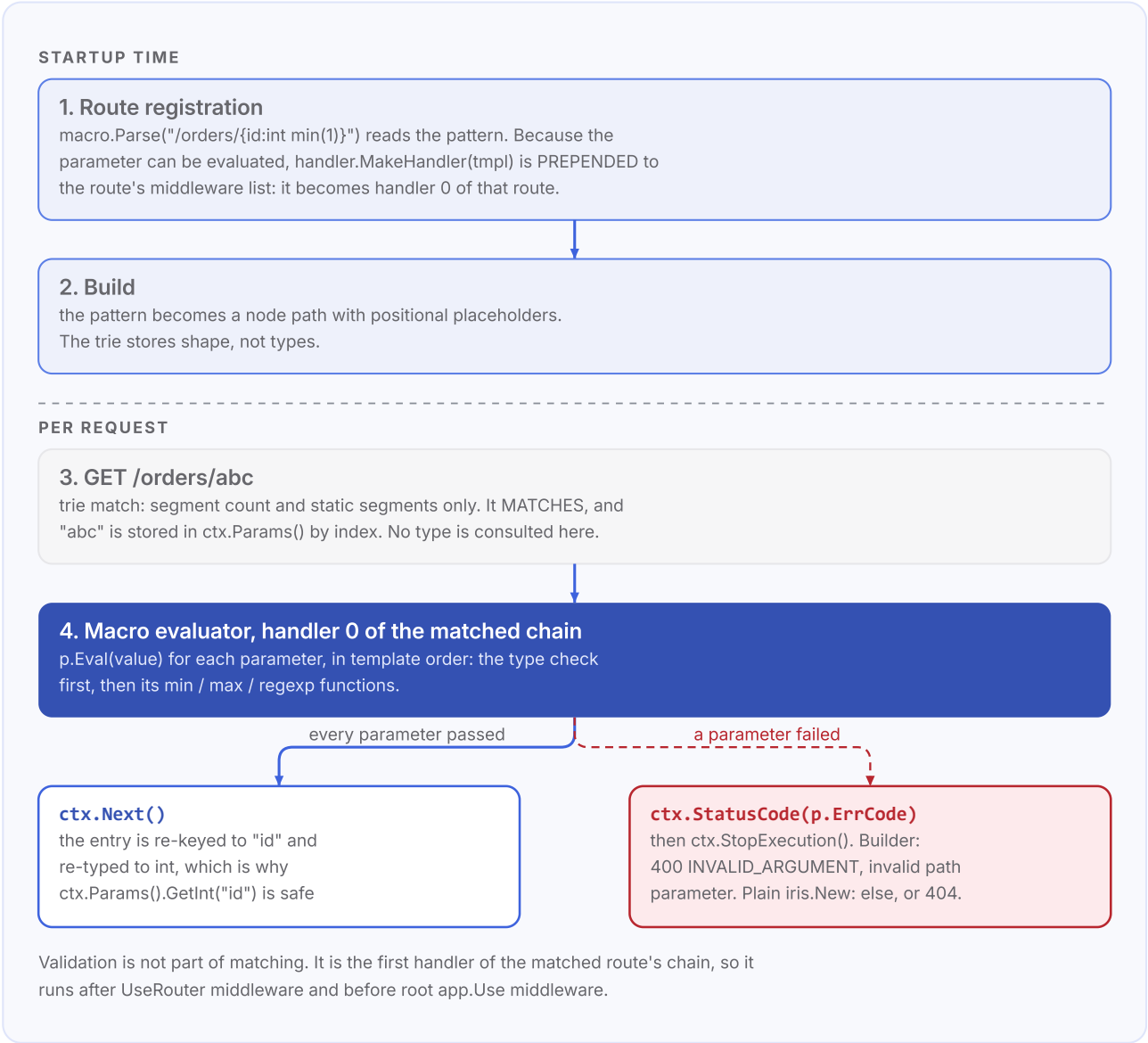
Constraint	Applies to	Example	Meaning
`min(n)`	string, numbers	`{name:string min(3)}`	minimum length (string) or value (numbers)
`max(n)`	string, numbers	`{name:string max(20)}`	maximum length (string) or value (numbers)
`range(a,b)`	numbers	`{age:int range(1,120)}`	value between a and b, inclusive
`regexp(expr)`	string	`{code:string regexp(^[A-Z]{3}$)}`	regular expression match
`prefix(s)`	string	`{name:string prefix(user_)}`	must start with s
`suffix(s)`	string	`{name:string suffix(.pdf)}`	must end with s
`contains(s)`	string	`{name:string contains(admin)}`	must contain s
`eq(v)`	string	`{state:string eq(active)}`	exact equality
`eqor(v1,v2,...)`	string	`{flag:string eqor(on,off)}`	equals any listed value
`else N`	all	`{id:int else 400}`	status code on failure

The `string` macro is the master type: when another type has no function of the requested name, the string version is used. It compares against the value's text, so `{id:int prefix(4)}` and `{id:uuid prefix(a6)}` work as they read.

```go
app.Get("/page/{page:int min(1) max(100) else 400}", pageHandler)
app.Get("/users/{name:string min(3) max(20) else 400}", profileHandler)
app.Get("/files/{filepath:path}", fileHandler)
app.Get("/shelves/{id:uuid}", shelfHandler)
```

When a typed parameter fails validation and no `else` code is given, the request never reaches your handler. Applications assembled with `iris.NewBuilder` (whose `RouterMiddlewares` step installs the macro error handler) render a 400 `INVALID_ARGUMENT` response with the message “invalid path parameter”.

Where that check happens is worth being precise about, because it is not where most readers assume. The router’s trie matches *shape*: segment count and static segments. `/orders/abc` does match the route `/orders/{id:int}`. The type check is a handler that route registration prepends to the route’s own chain, so it runs as that route’s handler zero: after any `UseRouter` middleware, and before the root `app.Use` middleware that would otherwise see the request.



The macro handler is prepended to the route at registration, the trie matches shape only, and the macro evaluator runs as handler zero of the matched chain, so a bad value is rejected before any handler of yours runs.

Accessing Parameters

Typed accessors return the value directly, with no `(value, error)` pairs. Each has a fixed failure default:

```go
func handler(ctx iris.Context) {
    name := ctx.Params().GetString("name")   // "" if missing
    id := ctx.Params().GetInt("id")          // -1 on failure
    uid := ctx.Params().GetUint64("id")      // 0 on failure
    active := ctx.Params().GetBool("active") // false on failure
    when := ctx.Params().GetDate("d")        // zero time.Time on failure
    sid := ctx.Params().GetUUID("sid")       // zero uuid.UUID on failure

    if v, ok := ctx.Params().Lookup("name"); ok {
        _ = v // the evaluated value as any
    }
}
```

Custom Macros

Add functions to an existing type or register a new type. Each application has its own macros, copied from the framework's defaults when it is created, so a registration stays in the application it was made on. Both must happen before the routes that use them are registered:

```go
// A new constraint on the string type:
// usage: /status/{status:string equals(active)}
app.Macros().Get("string").RegisterFunc("equals",
    func(expected string) func(string) bool {
        return func(paramValue string) bool {
            return paramValue == expected
        }
    })

// A completely new parameter type:
// usage: /events/{when:datetime}
app.Macros().Register("datetime", "", time.Time{}, false, false,
    func(paramValue string) (any, bool) {
        t, err := time.Parse("2006-01-02T15:04:05", paramValue)
        if err != nil {
            return nil, false
        }
        return t, true
    })
```

Route Groups

`Router(relativePath, ...)` creates a child router that inherits its parent's prefix and middleware. Groups nest to any depth, and any `iris.Handler` passed alongside the path becomes middleware of the group's matched routes.

```go
package main
```

```
import "github.com/kataras/iris/v14"

func main() {
    app := iris.New()

    api := app.Router("/api")
    api.Get("/status", status)  // GET /api/status

    v1 := api.Router("/v1")
    v1.Get("/users", listUsers) // GET /api/v1/users

    // Closure form: configure the child router in place.
    app.Router("/reports", func(r iris.Router) {
        r.Use(requireAuth)
        r.Get("/", listReports)
        r.Get("/{id:int}", getReport)
    })

    // Handlers passed to Router become the group's middleware.
    admin := app.Router("/admin", requireAuth, requireAdmin)
    admin.Get("/stats", stats)

    app.Listen(":8080")
}
```

Three registration points control where middleware runs; they matter most on groups:

- `UseRouter(handlers...)` runs before the routing decision itself: on every request, including not-found (404) and method-not-allowed (405).
- `Use(handlers...)` on the root application also covers 404 and 405 responses; on a child router it runs only for that group's matched routes.
- Handlers listed per route run last, in the order given, before the main handler.

The middleware contract is plain: call `ctx.Next()` to continue the chain, or write a response and return to stop it. The built-in middleware catalogue is covered in Chapter 12.

Controllers

The primary way to organize routes in Iris is a small convention rather than a base class: a controller is a plain struct with a constructor that declares its dependencies and a `Routing(r iris.Router)` method that declares its routes. The router calls `Routing` once at startup.

GO

```
// api/shelves.go
package api

import (
    "github.com/kataras/iris/v14"
```

```
    "bookshelf/library"
)

// ShelvesAPI groups the /shelves handlers around their dependencies.
type ShelvesAPI struct {
    svc *library.Service
}

// NewShelvesAPI is the controller's dependency declaration: the DI
// container calls it with the registered *library.Service.
func NewShelvesAPI(svc *library.Service) *ShelvesAPI {
    return &ShelvesAPI{svc: svc}
}

// Routing is the controller's route table.
func (api *ShelvesAPI) Routing(r iris.Router) {
    r.Get("/", api.list)
    r.Post("/", api.create)
    r.Get("/{id:uuid}", api.get)
    r.Delete("/{id:uuid}", api.delete)

    // A shelf's books live in their own nested controller; its
    // constructor is resolved by the container exactly like this one.
    r.Router("/{shelf_id:uuid}/books", NewBooksAPI)
}

func (api *ShelvesAPI) list(ctx iris.Context) {
    shelves, err := api.svc.Shelves(ctx)
    ctx.OK(shelves, err)
}

func (api *ShelvesAPI) get(ctx iris.Context) {
    id := ctx.Params().GetUUID("id")

    shelf, err := api.svc.Shelf(ctx, id)
    ctx.OK(shelf, err)
}
```

The `ctx.OK(value, err)` helpers render the value or hand the error to the central error pipeline; they are covered in Chapter 6 and Chapter 9.

Registering Controllers

Pass the constructor to `Router`. Its parameters are resolved from the dependencies you registered:

```
app := iris.New()
app.RegisterDependency(library.NewMemStore(), library.NewService)
app.Router("/shelves", api.NewShelvesAPI)
```

GO

In a builder-assembled application the `API` step does the same after `Services` has populated the container. This is the recommended setup, shown in Chapter 3:

```
iris.NewBuilder().
```

GO

```
    // ... Services(repo, library.NewService), middleware steps ...
    API("/shelves", api.NewShelvesAPI).
    Build().
    Listen(":8080")
```

Accepted Controller Shapes

`Router(path, c ...any)` accepts, in any combination:

- a `func(r iris.Router)` closure;
- an `iris.Handler` (or a slice of handlers), registered as the group's middleware;
- a pointer to a struct implementing `Routing(r iris.Router)`, whose exported fields are bound from the registered dependencies;
- a constructor function whose parameters are resolved from the dependencies and which returns such a struct, the `NewXxxAPI` form used above;
- the helpers `iris.Middleware(handlers...)` and `iris.Dependencies(deps...)`, which add group middleware and controller-only dependency overrides positionally:

```go
app.Router("/api",
    iris.Middleware(authMiddleware), // group middleware
    iris.Dependencies(specialDep),   // extra deps for what follows
    api.NewShelvesAPI,
    api.NewAdminAPI,
)
```

Guards and Positional Middleware

A guard is middleware with a yes/no job: it checks one precondition (the record exists, the user is allowed to write) and either lets the chain continue or ends the request with the right error. Putting the check in a guard means the handlers behind it never restate it.

Because guards are methods on the controller struct, they see the same injected dependencies as the handlers. Two idioms cover most cases: a *group guard* registered with `r.Use` runs before every route of the controller, and a *positional guard* placed between path and handler protects individual routes.

```go
// api/books.go
package api

import (
    "github.com/kataras/iris/v14"
    "github.com/kataras/iris/v14/rest/errors"

    "bookshelf/library"
```

```
)

// shelfContextKey caches the guard-loaded shelf on the request.
const shelfContextKey = "bookshelf.shelf"

// BooksAPI serves /shelves/{shelf_id:uuid}/books.
type BooksAPI struct {
    svc *library.Service
}

func NewBooksAPI(svc *library.Service) *BooksAPI {
    return &BooksAPI{svc: svc}
}

func (api *BooksAPI) Routing(r iris.Router) {
    // Group guard: every book route needs an existing shelf.
    r.Use(api.requireShelf)

    r.Get("/", api.list)
    // Positional guard: writes also need an unlocked shelf.
    r.Post("/", api.requireUnlocked, api.create)
    r.Put("/{id:uuid}", api.requireUnlocked, api.update)
    r.Delete("/{id:uuid}", api.requireUnlocked, api.delete)
}

// requireShelf loads the shelf once, replies 404 when it does not
// exist and caches it for the handlers that follow.
func (api *BooksAPI) requireShelf(ctx iris.Context) {
    shelfID := ctx.Params().GetUUID("shelf_id")

    shelf, err := api.svc.Shelf(ctx, shelfID)
    if errors.HandleError(ctx, err) {
        return // mapped 404, request stopped.
    }

    ctx.Values().Set(shelfContextKey, shelf)
    ctx.Next()
}

// requireUnlocked rejects writes against a locked shelf.
func (api *BooksAPI) requireUnlocked(ctx iris.Context) {
    shelf, _ := ctx.ValueAs[library.Shelf](shelfContextKey)
    if shelf.Locked {
        errors.PermissionDenied.Detail(ctx, "shelf is locked",
            "unlock the shelf before modifying its books")
        return
    }
    ctx.Next()
}
```

The handlers then read the cached shelf with `ctx.ValueAs[library.Shelf](shelfContextKey)` instead of loading it again. The full controller, with its list, create, update and delete handlers, is developed step by step in the framework's `TUTORIAL.md`.

Error-Returning Handlers

Plain handlers return nothing; error handling happens inside them. The `rest/errors` package offers an alternative: `errors.NewRouter` wraps any `iris.Router` so that routes accept `func(ctx iris.Context) error`. A non-nil return value flows through the central error map: `sql.ErrNoRows` becomes 404 by default, and your own mappings come from `errors.NewOptions()` as shown in Chapter 9.

```go
package main

import (
    "github.com/kataras/iris/v14"
    "github.com/kataras/iris/v14/rest/errors"
)

func main() {
    app := iris.New()

    r := errors.NewRouter(app.Router("/api"))

    r.UseRouter(func(ctx iris.Context) error {
        // Runs before routing, even for 404/405 requests.
        // Return nil to continue; a non-nil error is rendered
        // by the error pipeline and stops the request.
        return nil
    })

    r.Get("/books/{id:uuid}", func(ctx iris.Context) error {
        book, err := findBook(ctx.Params().GetString("id"))
        if err != nil {
            return err // mapped centrally.
        }
        return ctx.JSON(book)
    })

    app.Listen(":8080")
}
```

The wrapper mirrors the core router: `Get`, `Post`, `Put`, `Delete` and the rest register error-returning functions, `Use` and `UseRouter` accept error-returning middleware, and `Subdomain` returns another wrapped router. The embedded core router remains accessible for anything else.

Subdomains

A subdomain is the label in front of your domain: in `admin.example.com` it is `admin`. DNS usually points every subdomain at the same server (a wildcard record), and the server tells them apart by the `Host` header each request carries, which lets one Iris application serve `example.com` and `admin.example.com` with entirely different route tables.

`Subdomain` is available on every router and returns a child router bound to a host label instead of a path prefix. Calling it on a child prepends the label, so subdomains nest naturally.

```go
admin := app.Subdomain("admin")
admin.Get("/", dashboard) // admin.example.com/

// Nested: registers "panel.admin."
panel := app.Subdomain("admin").Subdomain("panel")
panel.Get("/", panelIndex) // panel.admin.example.com/

// Wildcard: any subdomain not matched by a static one.
wild := app.WildcardSubdomain()
wild.Get("/", func(ctx iris.Context) {
    ctx.Writef("subdomain: %s", ctx.Subdomain())
})
```

`WildcardSubdomain` is a method of the root application. Matching enforces dot boundaries: for a registered `admin` subdomain, requests to `admin` and `staff.admin` match, while `badmin` does not.

Route Names and Introspection

Every route gets a unique-per-application name, defaulting to method + subdomain + path (for example `"GET/users/{id:uuid}"`). Override it with `SetName`, look routes up by name, and build concrete paths from a route's template:

```go
route := app.Get("/users/{id:uuid}", getUser)
route.SetName("user.get")

r := app.Routes().GetByName("user.get")
path := r.ResolvePath("42") // "/users/42"
```

`None` registers an *offline* route under the virtual method `iris.MethodNone`; it is not reachable over HTTP but can be inspected and toggled online by swapping its exported `Method` field. Before the server starts, `app.RemoveRoute(name)` deletes a route by name and reports whether it was found.

The router is built once, when the server starts, so by default none of this reaches a running server: a removed route keeps answering, and a route registered late is not served and logs a warning. To change routes while the server runs, set `Options.Router.EnableDynamicHandler` before the first build and call `app.BuildRouter()` after each change. Chapter 11 lists what the option leaves fixed and what it costs.

Registration order rarely matters, because routes are sorted at build time: static paths sort before parameterized paths of equal depth, and more specific parameter types (`int`, `uuid`, ...) are evaluated before the `string` and `path` fallbacks when several routes share a position. Parameterized routes of equal priority keep their registration order.

Generic MVC Controllers

Two acronyms first. MVC (Model-View-Controller) is a long-standing way to organize applications: the model holds data and rules, the view renders output, and the controller connects requests to both. In an HTTP API, "controller" is the request-handling struct. CRUD names the four operations nearly every resource needs: create, read, update, delete.

For classic resource CRUD, the `controller/mvc` package generates the route table for you: a controller implements a generic interface and `mvc.UseController` mounts it, with `With` nesting further controllers under the same path.

```go
import "github.com/kataras/iris/v14/controller/mvc"

app.Router("/stores", mvc.UseController[*StoreController]().
    With("/books", mvc.UseController[*BookController]()))
```

This chapter's `Routing` convention remains the general-purpose tool; the generic MVC controllers and their extensions are covered in Chapter 24.

Summary

- Method functions (`Get`, `Post`, ...) return a `*Route`; `Handle` and `Any` register one route per method and return a `*RouteRegistry`.
- Path parameters are typed macros such as `{id:uuid}` and `{n:int min(1)}`, validated before the handler runs, with `else N` controlling the failure status and `ctx.Params().GetX` reading the values.
- `Router("/prefix")` groups routes; handlers passed to it become group middleware, and closures configure the child in place.
- A controller is a struct with a `NewXxxAPI` constructor resolved by dependency injection and a `Routing(r iris.Router)` route table; controllers nest through `r.Router(path, NewChildAPI)`.
- Guards are controller methods: `r.Use` for the whole group, a positional handler for single routes.

- `errors.NewRouter` wraps any router so handlers return `error` into the central error pipeline.
- `Subdomain` and `WildcardSubdomain` route by host label; names, offline routes and `ResolvePath` cover introspection needs.

Further Reading

- MDN: URIs. The anatomy of URLs: scheme, host, path, query and fragment.
- RFC 3986: Uniform Resource Identifier — Generic Syntax: the specification that defines path segments and query strings precisely.
- RFC 9110: HTTP Semantics: what each method means on a resource, the semantics your route tables encode.
- net/url (pkg.go.dev): how Go itself represents and parses URLs.
- uuid (pkg.go.dev): the standard library's UUID package (Go 1.27), used by this chapter's examples and by the `:uuid` route macro.
- RFC 9562: Universally Unique IDentifiers: the UUID formats and versions the `:uuid` macro accepts.

CHAPTER 5

Reading Requests

Every handler receives an `iris.Context`, and half of its surface exists to answer one question: what did the client send? This chapter covers that half: request metadata, typed path and query accessors, body decoding with the recommended `ctx.BindJSON` path and the lower-level `ctx.Read*` family, pagination input, forms and file uploads, cookies, body size limits, and the application-level validation hook. Writing responses is the subject of Chapter 6.

Background

An HTTP request is a small, structured text message. Its first line names a method and a path: the method says what kind of action the client intends (`GET` reads, `POST` creates, `PUT` replaces, `DELETE` removes), and the path says which resource the action targets. Headers come next, then optionally a body. Data can arrive in four distinct places, and each place has a job:

- **The path** identifies the resource. In `/users/42`, the `42` is a path parameter (part of the URL itself), so it is always present and always text until you convert it.
- **The query string** is everything after the `?`. In `/search?q=iris&page=2`, `q` and `page` are query parameters: they refine a request (filters, sorting, page numbers) without changing which resource it targets.
- **Headers** are key-value metadata about the request: who the client is (`User-Agent`), what credentials it carries (`Authorization`), what format the body is in (`Content-Type`).
- **The body** is the payload itself: the JSON document to store, the form fields, the uploaded file. Only some methods carry one; a `GET` usually does not.

The `Content-Type` header tells the server how to interpret the body's bytes, using a media type (also called a MIME type) such as `application/json` or `text/html`. JSON is the dominant one for APIs: a plain-text format defined by RFC 8259, with exactly six shapes: objects (`{}`), arrays (`[]`), strings, numbers, booleans and `null`. Because it is plain text, every language can produce and read it, which is why it became the common tongue between web services.

HTML forms have their own two media types. A simple form posts `application/x-www-form-urlencoded`: field names and values joined into one `name=value&...` string, essentially a query string moved into the body. A form with a file input posts `multipart/form-data`: the body is split into parts separated by a boundary marker, each part carrying its own headers, so text fields and raw file bytes travel in one request.

One rule shapes everything in this chapter: never trust the client. Every piece of a request (path, query, headers, body, cookies, even the declared `Content-Type`) is chosen by the sender, and the sender is not always a well-behaved browser. Anyone can send any bytes with a one-line `curl` command, so client-side checks (a form's `required` attribute, a JavaScript validator) are conveniences, not guarantees. Server-side validation is the real gate, and it is what this chapter's reading helpers build in: `ctx.BindJSON` decodes *and* validates before your logic runs, the typed accessors refuse malformed numbers, and body limits stop a client from streaming gigabytes at your decoder. The problem that disappears from your code is manual string parsing and defensive `if` ladders: handlers receive typed, validated values or never run at all.

Request Metadata

The basics (method, path, host) are one call away:

```go
package main

import "github.com/kataras/iris/v14"

func main() {
    app := iris.New()
    app.Get("/info", info)
    app.Listen(":8080")
}

func info(ctx iris.Context) {
    ctx.JSON(iris.Map{
        "method":    ctx.Method(),         // "GET"
        "path":      ctx.Path(),           // "/info"
        "uri":       ctx.FullRequestURI(), // "http://localhost:8080/info"
        "scheme":    ctx.Scheme(),         // "http://"
        "host":      ctx.Host(),           // "localhost:8080"
        "hostname":  ctx.Hostname(),       // "localhost"
        "subdomain": ctx.Subdomain(),      // "" unless e.g. api.example.com
        "ip":        ctx.RemoteAddr(),
    })
}
```

When you need the raw `*http.Request`, `ctx.Request()` returns it.

The client address

`ctx.RemoteAddr()` returns the client IP. By default it parses only the transport-level `Request.RemoteAddr`: proxy headers such as `X-Forwarded-For` are ignored, because any client can forge them. Enable them only when your server sits behind a proxy you control:

```go
app := iris.New(iris.Options{
    Context: iris.ContextOptions{
        RemoteAddrHeaders: []string{"X-Forwarded-For", "X-Real-IP"},
    },
})
```

With `RemoteAddrHeaders` set, `RemoteAddr` reads the listed headers in order and each one from right to left. A proxy appends the address it received the request from, so the right end of `X-Forwarded-For` is what your own proxy wrote and the left end is whatever the client sent. The first public address is the result, without its port when it has one: `1.2.3.4, 8.8.4.4:1234` gives `8.8.4.4`. An address inside the ranges of `RemoteAddrPrivateSubnets` is taken for one of your proxies and skipped.

An entry that is not an address ends the reading of its header with no result: `unknown`, `unix:`, a host name, an address with a zone. It is never skipped, because the entry on its left may be the client's own writing. `RemoteAddr` then reads the next header, and when no header gives a result it returns the address of the connection.

`RemoteAddrHeadersForce` adds one step before that fallback: a header whose entries are all addresses, none of them public, gives its first address from the left. A header that ended at an entry that is not an address stays out of it, so `10.0.0.1, unknown` returns the connection address with the option on or off.

The related `ctx.Addr()` reads from the left instead and returns the first address it finds, even when it belongs to a private network. Never base security decisions solely on a client-reported address.

Request headers

`ctx.Header(name)` reads a **request** header. The response counterpart is `ctx.ResponseHeader(name)`, covered in Chapter 6:

```go
auth := ctx.Header("Authorization")
agent := ctx.Header("User-Agent")
```

One deliberate quirk: when the client repeats a header, `ctx.Header` returns all values joined by `", "` instead of only the first (which is what `http.Header.Get` would do). Code that compares the result against a secret therefore rejects a request carrying two copies of the header rather than silently honoring the first one.

Headers also bind to a struct through the `header` field tag:

```go
type ClientInfo struct {
    UserAgent string `header:"User-Agent"`
    RequestID string `header:"X-Request-ID"`
}

var info ClientInfo
if err := ctx.ReadHeader(&info); err != nil {
    errors.InvalidArgument.Err(ctx, err)
    return
}
```

Content type of the request

`ctx.ContentType()` parses the request's `Content-Type` header into a structured value; `ctx.MediaType()` returns just the media type string, and `ctx.Charset()` and `ctx.ContentLength()` complete the picture:

```go
ct := ctx.ContentType()   // ContentType{MediaType, Charset, Boundary, ...}
mt := ctx.MediaType()     // e.g. "application/json"
cs := ctx.Charset()       // e.g. "utf-8", "" when absent
n := ctx.ContentLength()  // int64, 0 when the header is missing
```

These are request-side methods. The response equivalents are `ctx.ResponseContentType()` and `ctx.ResponseMediaType()`.

Path and Query Parameters

Path parameters

Dynamic route segments (`{id:int64}`, `{slug}`; see Chapter 4 for the macro types) are read through `ctx.Params()`. Accessors return the value directly with a type-specific default on a missing key or failed conversion; there are no `(value, error)` tuples:

```go
app.Get("/users/{id:int64}/posts/{slug}", func(ctx iris.Context) {
    id := ctx.Params().GetInt64("id")      // int64, -1 on failure
    slug := ctx.Params().GetString("slug") // string, "" on failure
    // ...
})
```

Accessor	Failure default
`GetString`	`""`

`GetInt`, `GetInt64`	`-1`
`GetUint`, `GetUint64`	`0`
`GetFloat64`	`-1`
`GetBool`	`false`
`GetUUID`	zero `uuid.UUID` (equal to `uuid.Nil()`)
`GetDuration`	`0`
`GetTime`	zero `time.Time`
`GetIP`	zero `netip.Addr` (`IsValid()` false)

When the default is ambiguous (is `-1` a real value or a miss?), use `Lookup`, which reports presence, or `Exists`:

```go
if raw, ok := ctx.Params().Lookup("id"); ok {
    _ = raw // the stored value, typed by the route macro.
}
```

A request whose typed parameter does not match the macro (say `/users/abc` against `{id:int64}`, or a malformed `{id:uuid}`) never reaches your handler; applications built with `iris.NewBuilder` render a 400 `INVALID_ARGUMENT` ("invalid path parameter") response automatically.

The context has a generic method that writes the error response for you (404 when the parameter is missing, 400 when it is not of type `T`), so the handler only handles the happy path:

```go
app.Get("/bookmarks/{id:int64}", func(ctx iris.Context) {
    id, ok := ctx.BindParam[int64]("id")
    if !ok {
        return // error response already sent.
    }
    // ...
})
```

To bind all path parameters at once, `ctx.ReadParams(&dst)` decodes into a struct using the `param` field tag; `ctx.BindParams[S]()` does the same and writes the error response itself, returning `(S, bool)`.

`T` is the type the macro stored, not the type you would like: `{id:int64}` stores an `int64`, and `{id:uuid}` stores a parsed `uuid.UUID`, so a uuid parameter is read as `ctx.BindParam[uuid.UUID]("id")` or `ctx.Params().GetUUID("id")`. Asking for `string` there answers 400, because the

assertion fails. `ctx.Params().GetString("id")` still works and returns the canonical lowercase text, whatever spelling the client sent.

Query parameters

`ctx.Query()` returns a read-only, sorted view of the URL query with the same typed accessors and the same failure defaults as `Params()`:

```go
// GET /search?q=iris&page=2&active=true&tags=go&tags=web
q := ctx.Query().GetString("q")        // "iris"
page := ctx.Query().GetInt("page")     // 2, -1 if missing or invalid
active := ctx.Query().GetBool("active") // true, false if missing
price := ctx.Query().GetFloat64("max") // -1 if missing

exists := ctx.Query().Exists("q")      // true
val, ok := ctx.Query().Lookup("q")     // "iris", true
```

A few getters parse the text into a richer type and return its zero value on a miss. `GetDuration` reads `?ttl=1h30m` as 90 minutes, `GetTime` reads an RFC 3339 timestamp such as `?since=2026-10-10T09:00:00Z`, and `GetIP` turns `?addr=10.0.0.7` into a `netip.Addr` whose `IsValid()` is false when the value is not an address:

```go
ttl := ctx.Query().GetDuration("ttl")    // time.Duration, 0 if missing or invalid
since := ctx.Query().GetTime("since")    // time.Time, zero if missing or invalid
addr := ctx.Query().GetIP("addr")        // netip.Addr
ref := ctx.Query().GetUUID("ref")        // uuid.UUID, uuid.Nil() if missing or invalid
```

Repeated keys are available in two forms:

```go
tags := ctx.Query().GetSlice("tags")        // ["go", "web"]: the original
                                            // []string from the query map.
cols := ctx.Query().GetStringSlice("columns") // splits the joined value by
                                              // the configured separator.
```

`GetStringSlice` splits on `ContextOptions.URLParamSeparator` (default `","`), so `?columns=a,b,c` and `?columns=a&columns=b&columns=c` both produce three elements; `GetSlice` returns the raw multi-value list as the client sent it. There is no `URLParam` method family in v14. Use the `Query()` accessors or `ReadQuery`.

Struct binding uses the `url` tag, and `ReadURL` combines path parameters with the query in one call:

```go
type SearchFilter struct {
    UserID   int      `param:"id"`   // from the path.
    Keyword  string   `url:"q"`      // from the query.
    Tags     []string `url:"tags"`
```

```
    Page     int      `url:"page"`
}

app.Get("/users/{id:int}/search", func(ctx iris.Context) {
    var filter SearchFilter
    if err := ctx.ReadURL(&filter); err != nil {
        errors.InvalidArgument.Err(ctx, err)
        return
    }
    ctx.JSON(filter)
})
```

`ctx.ReadQuery(&dst)` does the same for the query alone. A field that has no `url` tag is bound there by its `form` tag, so a struct tagged `form` only serves a GET query and a POST form alike. `ctx.ReadBody` picks between the two: it reads the query on a GET and the form on a form post. For such a field the `form` name is the only query key (the Go field name no longer matches), `form:"-"` keeps it out of the query, and the tag's `required` option applies to a query that is not empty. A `url` tag still wins, and a field with neither tag binds by its field name.

For a date without a time, give the field the type `types.SimpleDate` (package `rest/types`). `ReadQuery`, `ReadParams` and `ReadURL` fill it from a `YYYY-MM-DD` value such as `?from=2026-01-01`, where a `time.Time` field expects a full RFC 3339 timestamp.

Strict query readers

The accessors above are lenient by design: `GetInt("page")` answers `-1` for a missing key *and* for `?page=banana`, and a filter built on them silently returns the wrong rows instead of telling the client it sent nonsense. Three readers make that a 400 instead:

```go
// GET /reports?from=2026-01-01&limit=50&detailed=yes
from, ok := ctx.QueryDate("from", iris.QueryRequired())
if !ok {
    return // 400 already written: missing, or not a date.
}

limit, ok := ctx.QueryInt("limit", iris.QueryDefault("25"))
if !ok {
    return
}

detailed, ok := ctx.QueryBool("detailed")
if !ok {
    return
}
```

The second result is the `ok` of the `Bind*` family: `false` means the response was already written and the handler returns. A present but unparsable value is always a 400 naming the key and the shape expected, never a silent zero. An absent value is the zero value unless an option says otherwise.

Option	Effect
`iris.QueryRequired()`	An absent or blank value is a 400 naming the key
`iris.QueryDefault(raw)`	The raw string to parse when the key is absent, stated in the client's own wire form (`"2020-01-02"`, `"25"`, `"true"`). It wins over `QueryRequired`
`iris.QueryZeroValues(v...)`	The placeholders older clients send for "unset" (`"0"`, `"zero"`, `"0000-00-00"`, `"null"`), each read exactly like an absent value

`QueryDate` parses `types.SimpleDate` (`2006-01-02`, or `2006-1-2`) and `QueryBool` accepts what `strconv.ParseBool` accepts plus `yes`, `no`, `on` and `off`, case-insensitively. Nothing is special-cased below the options: `?from=null` is a 400 unless `QueryZeroValues("null")` declared it, because what a raw value means is the endpoint's decision and not the parser's. An unparsable `QueryDefault` is a programmer error and panics on the first request rather than rendering as the client's fault.

An endpoint that answers to more than one spelling of the same parameter states them once, separated by a space, and each is tried in the order given until one carries a value:

```go
date, ok := ctx.QueryDate("date day from") // ?day=2026-01-02 answers here
if !ok {
    return
}
```

A URL query key cannot contain a space, so a key without one is a single key and reads exactly as it always did. `QueryInt` and `QueryBool` take the same. When no key carried a value the 400 names all of them; when one carried an unparsable value it names only that one, since the client did not send the others.

Binding JSON Requests

For JSON APIs the recommended entry point is the generic context method `ctx.BindJSON[T]()`. It decodes the body, runs the payload's own `Validate() error` method when the type implements it (the `iris.Validatable` interface), and on any failure writes a structured 4xx response and returns `false`. The handler simply returns. You never call `Validate` by hand.

The `rest/errors/validation` package provides small field builders for those `Validate` methods:

```go
package main

import (
    "github.com/kataras/iris/v14"
```

```
    "github.com/kataras/iris/v14/rest/errors/validation"
)

type CreateUserInput struct {
    Name  string `json:"name"`
    Email string `json:"email"`
    Age   int    `json:"age"`
}

// Implementing Validate makes ctx.BindJSON run it after decoding.
func (in CreateUserInput) Validate() error {
    return validation.Join(
        validation.String("name", in.Name).NotEmpty().Length(2, 100),
        validation.String("email", in.Email).Email(),
        validation.Number("age", in.Age).Positive(),
    )
}

func createUser(ctx iris.Context) {
    input, ok := ctx.BindJSON[CreateUserInput]()
    if !ok {
        return // 400 already written: malformed JSON or failed Validate.
    }

    // input is decoded AND valid here.
    ctx.Created(input.Name, nil) // see Chapter 6 for responses.
}

func main() {
    app := iris.New()
    app.Post("/users", createUser)
    app.Listen(":8080")
}
```

A failed `Validate` renders each field error individually:

JSON

```
{
    "http_error_code": {"canonical_name": "INVALID_ARGUMENT", "status": 400},
    "message": "validation failure",
    "detail": "fields were invalid",
    "validation": [
        {"field": "name", "value": "",
         "reason": "must not be empty, must be between 2 and 100 characters"},
        {"field": "age", "value": 0, "reason": "must be positive"}
    ]
}
```

The builders cover strings (`NotEmpty`, `Length`, `Email`, `Fullname`), numbers (`Positive`, `Negative`, `NonZero`, `InRange`) and slices, and compose with plain functions through `Func` for custom rules. `validation.Join` drops entries that passed, and returns `nil` when everything did.

Optional bodies and variants

`ctx.BindJSONOptional[T]()` behaves like `BindJSON` except that an empty request body is a success: it returns the zero value of `T` with `true` and skips validation. Use it when the body is a nice-to-have, such as an optional filter object.

Two more variants accept a body that may be either a single object or an array of objects: `ctx.BindJSONOneOrMany[T]()` and `ctx.BindJSONList[T]()` (which normalizes a single object into a one-element slice). Every element is validated.

Partial updates and the column list

A `PATCH` sends the body and the list of fields it means to change. The list travels in the `columns` query key, and reading it apart from the body invites the bug the whole family exists to prevent: a typo matches no column, the update changes nothing, and the client is told it succeeded. `ctx.BindUpdate[T]()` reads both and validates them against each other:

```go
func updateUser(ctx iris.Context) {
    id, ok := ctx.BindParam[int64]("id")
    if !ok {
        return
    }

    input, columns, ok := ctx.BindUpdate[UpdateUserInput]()
    if !ok {
        return // 400 already written: the body, or an unknown column.
    }

    updated, err := svc.UpdateUser(ctx, id, input, columns...)
    ctx.NoContent(updated, err) // 204, or 304 when nothing changed.
}
```

The body half is `BindJSON` exactly: the same decoder, the same `RequestValidator` hook, the same `Validate` call. The columns arrive either as a repeated key (`?columns=name&columns=email`) or as one comma-separated value (`?columns=name,email`); both spellings are accepted, values are trimmed, empties dropped and duplicates removed, keeping the client's order. No columns at all means a full update and yields a nil slice.

Every column is checked against the JSON member names of `T`, the `json` tag name or the Go field name where there is no tag, and an unknown one answers 400 naming the column and the names it could have been. The comparison is exact where the body decoder is case-insensitive, because a column name reaches an `UPDATE` statement: the endpoint must receive the spelling it declared rather than a normalized guess. The names that come back are still the client's, so a store that builds SQL from them maps them to its own columns and never interpolates them.

The Context Read Family

When the payload is not JSON, or when you want to handle the error yourself instead of letting the helpers respond, the context offers a `Read*` method per format. Each decodes into a pointer, using a struct tag that matches the source:

Method	Source	Struct tag
`ctx.ReadJSON(&v, opts...)`	JSON body	`json`
`ctx.ReadXML(&v)`	XML body	`xml`
`ctx.ReadYAML(&v)`	YAML body	`yaml`
`ctx.ReadMsgPack(&v)`	MessagePack body	—
`ctx.ReadProtobuf(msg)`	Protobuf body (`proto.Message`)	—
`ctx.ReadForm(&v)`	URL-encoded or multipart form	`form`
`ctx.ReadQuery(&v)`	URL query	`url` , else `form`
`ctx.ReadParams(&v)`	path parameters	`param`
`ctx.ReadHeader(&v)`	request headers	`header`
`ctx.ReadURL(&v)`	path parameters + query	`param` , `url`
`ctx.ReadBody(&v)`	dispatched by content type	depends

In the form, query, parameter and header readers a `types.SimpleDate` field is read from `YYYY-MM-DD` and a `time.Time` field from RFC 3339.

XML and YAML are, like JSON, human-readable text formats with different syntax. MessagePack and Protobuf are binary encodings: they trade readability for smaller, faster payloads and are common in service-to-service traffic rather than browser-facing APIs.

`ctx.ReadJSON` accepts optional `iris.JSONReader` options:

```go
var data map[string]any
err := ctx.ReadJSON(&data, iris.JSONReader{
    DisallowUnknownFields: true, // reject keys with no struct field.
})
```

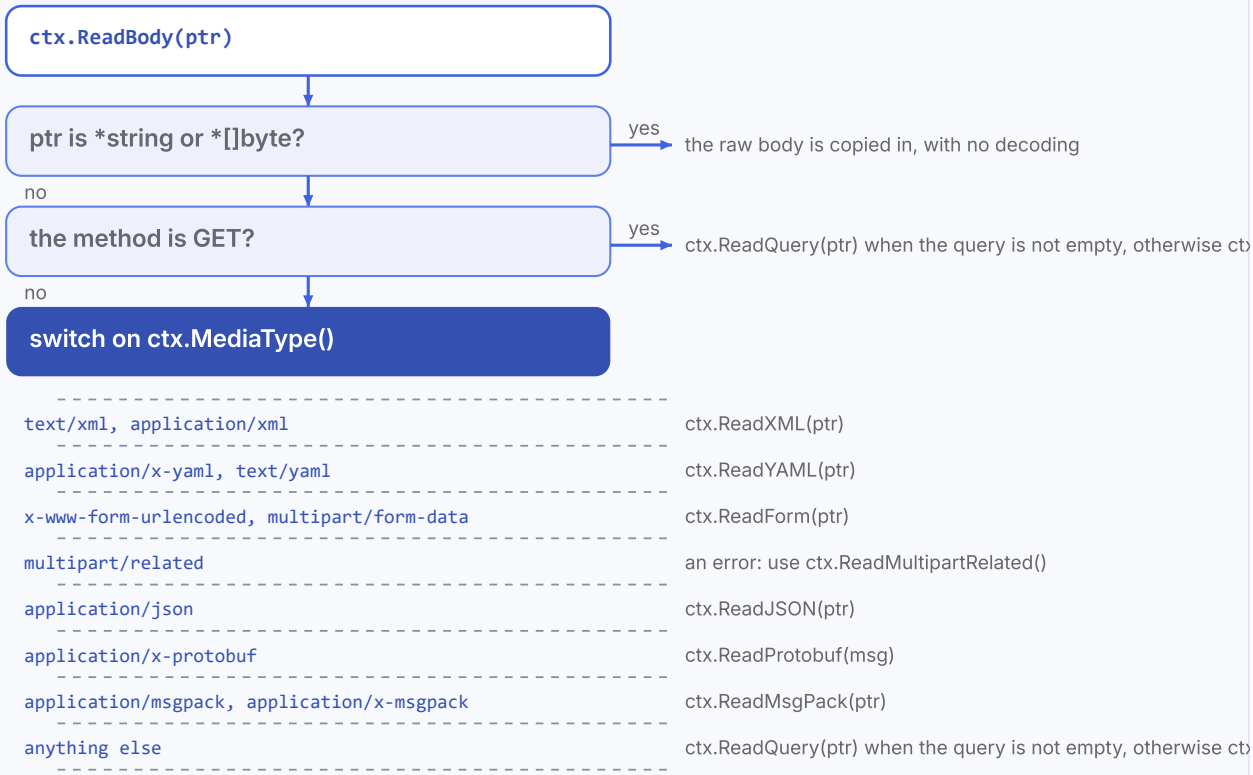
Decoding runs on the standard library's `encoding/json/v2` (Go 1.27). Two of its stricter rules apply to every reader: a body with duplicate object keys or invalid UTF-8 is rejected (the bind helpers render that as a 400). One rule is deliberately relaxed: member names still match struct fields case-insensitively, as deployed clients expect from the old `encoding/json` ; set `JSONReader.CaseSensitive` to restore v2's strict matching for a call. The `UnmarshalOptions` field takes raw `encoding/json/v2` options, appended after the defaults so yours win: `jsontext.AllowDuplicateNames(true)` , for example, accepts duplicate members again.

For large arrays, `ctx.ReadJSONStream` decodes one object at a time instead of buffering the whole body:

```go
var users []User
err := ctx.ReadJSONStream(func(decode iris.DecodeFunc) error {
    var u User
    if err := decode(&u); err != nil {
        return err
    }
    users = append(users, u)
    return nil
})
```

ReadBody: content-type dispatch

`ctx.ReadBody(&v)` picks the decoder for you. For `GET` requests it reads from the URL query (falling back to the form). For other methods it switches on the request's media type: XML, YAML, url-encoded or multipart forms, JSON, Protobuf and MessagePack are dispatched to the matching `Read*` method; an unrecognized type falls back to the query (when one exists) and finally to JSON.



Empty input is not an error by default: ReadForm, ReadQuery and ReadBody leave the struct at its zero value. Set ContextOptions.FireEmptyFormError to get iris.ErrEmptyForm instead. ReadProtobuf returns ErrContentNotSupported when the target is not a proto.Message.

ctx.ReadBody short-circuits for raw string and byte targets, reads the query or the form for a GET, and otherwise switches on the media type, falling back to the query and then to JSON.

```go
var raw []byte
if err := ctx.ReadBody(&raw); err != nil { // raw body, no decoding.
    errors.InvalidArgument.Err(ctx, err)
    return
}
```

The lowest level is `ctx.Body()`, which returns the body as `[]byte`. Like any HTTP body, it can be consumed once: a later `ReadJSON` on the same request finds it empty.

Empty-input behavior: `ReadForm`, `ReadQuery` and `ReadBody` return a `nil` error on empty input by default, leaving the struct at its zero value. Set `ContextOptions.FireEmptyFormError` to make them return `iris.ErrEmptyForm` instead. `ReadForm` errors on unknown client fields; ignore that class with `err != nil && !iris.IsErrPath(err)`.

Pagination Input

Returning an entire table from one endpoint does not scale: the response grows without bound and most clients display a screenful at a time. Pagination splits the list into fixed-size pages the client requests one by one; in database terms, the page number and size translate into how many rows the query fetches and how many it skips first.

List endpoints usually accept `?page=&size=&details=` plus an optional JSON filter body. `ctx.BindPagination` reads both at once:

```go
type BookFilter struct {
    Author string `json:"author"`
}

func listBooks(ctx iris.Context) {
    opts, filter, ok := ctx.BindPagination[BookFilter]()
    if !ok {
        return // error response already sent.
    }

    // opts.GetLimit(), opts.GetOffset() feed the database query.
    books, total, err := svc.Books(ctx, opts, filter)
    ctx.OK(iris.NewPage(opts, &filter, books, total), err)
}
```

`iris.PageOptions` carries `Page`, `Size` and `Details`; `GetLimit` returns `Size` bounded by `pagination.MaxPageSize` (falling back to `pagination.DefaultPageSize`, 100) and `GetOffset` derives the offset from the page number. Set the two bounds once at startup, either directly or through

`iris.Patches().Context().SetMaxPageSize(n)` and `SetDefaultPageSize(n)`. An empty body yields the zero filter. The filter is optional by design.

A filter body on a read raises the question of which method carries it. GET has no defined body semantics, and POST says the request changes something, which it does not. HTTP QUERY (RFC 10008) is the method made for this: safe and idempotent like GET, with a body like POST, and its response is cacheable. Iris registers it with `app.Query`, so one path can answer both shapes:

```go
app.Get("/books", listBooks)   // ?page=&size= only.
app.Query("/books", listBooks) // ?page=&size= plus a filter body.
```

Send the QUERY filter as JSON. On Go 1.27, net/http parses a url-encoded body only for POST, PUT and PATCH, so a form-encoded QUERY body never reaches `ctx.Form*`.

The page types themselves (`PageOptions`, `Page[T]`, `NewPage`, `ListFilter`, `CountResponse[N]`) live in `core/pkg/pagination`, a package that imports only the standard library. `iris` and `context` alias them, so `iris.Page[T]` and `pagination.Page[T]` are the same type, and the `client` package decodes a list into that type too.

`ctx.Paginated` collapses the whole flow (read options and filter, run a collector, respond with a `iris.Page[T]`) into one call:

```go
import (
    "context"

    "github.com/kataras/iris/v14"
)

func listBooks(ctx iris.Context) {
    ctx.Paginated(func(reqCtx context.Context,
        opts iris.PageOptions, filter BookFilter) ([]Book, int64, error) {
        return svc.Books(reqCtx, opts, filter)
    })
}
```

The response shape (`current_page`, `page_size`, `total_pages`, `total_items`, `has_next_page`, `filter`, `items`) is covered in Chapter 6.

The shared filter, and who chooses the sort

Most list filters carry the same four or five members before they carry anything of their own. `iris.ListFilter` is that half, ready to embed:

```go
type BookFilter struct {
    iris.ListFilter
```

```
    Author string `json:"author,omitzero" url:"author"`
}
```

It fields `Term`, `Names`, `Substring`, `Sort` and `SortDescending`, and carries both tag sets on purpose: the `json` names for a `QUERY` or `POST` list body and the `url` names for a `GET` list route read through `ReadQuery`. `f.SortColumn(allowed...)` returns the requested column or an error wrapping `iris.ErrSortNotAllowed`, and `f.SortOrder()` returns `"ASC"` or `"DESC"`.

A sort column reaches an `ORDER BY` clause, so which columns a list offers belongs to the endpoint and never to the request. Two options say so declaratively, and both answer 400 before the collector runs:

```go
func listBooks(ctx iris.Context) {
    ctx.Paginated(func(reqCtx context.Context,
        opts iris.PageOptions, filter BookFilter) ([]Book, int64, error) {
        return svc.Books(reqCtx, opts, filter)
    }, iris.AllowSort("title", "published_at"))
}
```

`iris.AllowSort(columns...)` limits the column a client may name, and the rejection names both the column and the allowed ones. `iris.DenySort(message)` refuses any sort at all, for a list whose order the query itself fixes. Both work on `ctx.BindPagination` too, and both read the decoded filter through its `SortRequest` method (the `pagination.SortRequester` interface), which `ListFilter` provides, so a filter without one is simply unaffected.

Sorting is often a permission rather than a shape. A list a client may read but not order should answer 403, not 400, and any error code carries the rejection:

```go
list, filter, ok := ctx.BindPagination[BookFilter](
    errors.PermissionDenied.DenySort("sorting is not available on this plan"))
if !ok {
    return
}
```

`errors.ErrorCodeName.DenySort`, from `rest/errors`, is `iris.DenySort` answered with that code and is otherwise identical.

A collector may take its filter by pointer, which is the shape a repository method already has, so it can be passed as a bare method value:

```go
ctx.Paginated(svc.ListBooksPaginated) // (context.Context, iris.PageOptions, *BookFilter) ([]Book, int64,
error)
```

Such a collector never receives nil. A request with no body decodes to the filter's zero value, so the repository reads an empty filter and the sort policy still has a real `SortRequest` to read.

Forms and File Uploads

Form values

HTML form submissions (`application/x-www-form-urlencoded` or `multipart/form-data`) bind through the `form` tag:

```go
type ContactForm struct {
    Name    string `form:"name"`
    Email   string `form:"email"`
    Message string `form:"message"`
}

func contact(ctx iris.Context) {
    var form ContactForm
    if err := ctx.ReadForm(&form); err != nil && !iris.IsErrPath(err) {
        errors.InvalidArgument.Err(ctx, err)
        return
    }
    // ...
}
```

Individual values are available through `ctx.Form()` (query + body values) and `ctx.PostForm()` (body values only), which return the same read-only entries view as `ctx.Query()`: `GetString`, `GetInt`, `GetSlice` and friends all apply.

Multipart form parsing buffers up to `ContextOptions.PostMaxMemory` in memory (default 32 MB, spillover goes to temporary files). Raise or lower it through `iris.Options{Context: iris.ContextOptions{...}}`.

File uploads

An uploaded file's name is client input like any other. A malicious client can submit a name such as `../../etc/crontab`, hoping the server joins it to the upload directory and writes outside it, an attack known as path traversal. Sanitizing the name before it touches the filesystem is therefore not optional, and the helper below does it for you.

`ctx.UploadFormFiles` is the batteries-included path: it saves every uploaded file into a destination directory, sanitizes file names against an allowlist pattern and blocks path-traversal attempts (unsafe names are silently skipped). The optional callback filters or renames files before saving:

```go
import (
    "mime/multipart"

    "github.com/kataras/iris/v14"
    "github.com/kataras/iris/v14/rest/errors"
)

func upload(ctx iris.Context) {
    files, total, err := ctx.UploadFormFiles("./uploads",
        func(ctx iris.Context, file *multipart.FileHeader) bool {
            return file.Size <= 8*iris.MB // reject anything larger.
        })
    if err != nil {
        errors.InvalidArgument.Err(ctx, err)
        return
    }

    ctx.JSON(iris.Map{"uploaded": len(files), "bytes": total})
}
```

For manual control, `ctx.FormFile(key)` returns the first uploaded file for a form key and `ctx.FormFiles(key, ...)` all of them; pair either with `ctx.SaveFormFile(header, dest)` to write to disk. When building the destination path yourself, sanitize the client-supplied name first. `context.SafeFilename` implements the same checks `UploadFormFiles` uses: it strips every directory component, holds the result to an allowlist of characters, rejects the reserved Windows device names (`NUL`, `COM1`, `CON.txt` and the rest, which open a device rather than a file), and resolves the destination directory before checking that the final path is inside it. The upload endpoints are a natural place for the body limits described below.

Cookies

A cookie is a small named value the server asks the browser to store; the browser then attaches it to every later request to the same site. That echo is what gives stateless HTTP a memory: it is how a login survives from one request to the next.

Two readers cover the request side:

```go
// Just the value; "" when the cookie is missing.
session := ctx.CookieValue("session_id")

// The full *http.Cookie; http.ErrNoCookie when missing.
cookie, err := ctx.Cookie("session_id")
if err == nil {
    _ = cookie.Value
}
```

`CookieValue` URL-unescapes the value for you. Setting, upserting and removing cookies happens on the response and is covered in Chapter 6.

Body Size Limits

Unbounded request bodies are a denial-of-service vector: a client can stream gigabytes into `ReadJSON` or hide them behind a compressed body. Iris offers an application-wide cap and a per-route middleware.

Application-wide: `ContextOptions.BodyMaxMemory` limits the total bytes read by `Body`, `ReadJSON`, `ReadJSONStream` and `ReadMultipartRelated`. That cap therefore also covers everything built on them, including `ReadXML`, `ReadYAML`, `ReadBody` and `ctx.BindJSON`:

```go
app := iris.New(iris.Options{
    Context: iris.ContextOptions{
        BodyMaxMemory: 4 * iris.MB,
    },
})
```

The default `0` means no limit. Reading past the cap fails with `context.ErrRequestEntityTooLarge` (package `github.com/kataras/iris/v14/context`), which the `rest/errors` package renders as a 413 `TOO_LARGE` response.

Per route or group: the `bodylimit` middleware applies the same enforcement wherever it is registered, typically on upload routes:

```go
import "github.com/kataras/iris/v14/middleware/bodylimit"

media := app.Router("/media", bodylimit.New(20*iris.MB))
media.Post("/upload", upload)
```

It rejects requests up front when the declared `Content-Length` already exceeds the limit, and wraps the body reader to catch chunked or compressed bodies that only reveal their size while being read. Such a body is read up to the limit and not one byte past it, and the read then fails with `context.ErrRequestEntityTooLarge`, in `ReadJSON` and `ctx.BindJSON` as in `Body`. A chunked form is the exception: `ReadForm` reports no error for one past the limit and binds nothing of it.

`PostMaxMemory` is a different knob: it bounds how much of a multipart form is buffered in memory during parsing, not how large the body may be. Use both where appropriate.

The RequestValidator Hook

The per-type `Validate() error` method with `ctx.BindJSON` is the recommended validation path. For teams standardizing on a tag-based validation library instead, the context exposes one application-level hook: `ContextOptions.RequestValidator`, any value implementing

```go
type Validator interface {
    Struct(any) error
}
```

It defaults to `nil`, so `validate:"..."` struct tags do nothing on their own. When set, every decoding `Read*` method (`ReadJSON`, `ReadXML`, `ReadYAML`, `ReadMsgPack`, `ReadForm`, `ReadQuery`, `ReadParams`, `ReadURL`, `ReadHeader`, `ReadBody`) calls it after a successful decode and returns its error. The typed handlers of `controller/mvc/handler` (Chapter 24) call it too, for a request or param struct bound from a body, a form, the URL query, path parameters or headers. Wiring a third-party library is a three-line adapter:

```go
import (
    "github.com/go-playground/validator/v10" // third-party.

    "github.com/kataras/iris/v14"
)

type structValidator struct{ v *validator.Validate }

func (s structValidator) Struct(ptr any) error { return s.v.Struct(ptr) }

func main() {
    app := iris.New(iris.Options{
        Context: iris.ContextOptions{
            RequestValidator: structValidator{validator.New()},
        },
    })
    // ...
}
```

The two mechanisms compose: `ctx.BindJSON` surfaces `RequestValidator` failures through the same structured 400 responses it uses for a failed `Validate()` method, so handlers keep the single `if !ok { return }` guard either way.

Summary

- `ctx.Method()`, `ctx.Path()`, `ctx.Host()` and friends expose request metadata; `ctx.RemoteAddr()` trusts proxy headers only when `RemoteAddrHeaders` is configured.
- `ctx.Header(name)` reads a request header (repeated values joined by `", "`); `ctx.ContentType()` / `ctx.MediaType()` describe the request body's declared type.
- `ctx.Params()` and `ctx.Query()` share typed accessors with fixed failure defaults (`GetString` → `""`, `GetInt` → `-1`); `Lookup` and `Exists` distinguish missing from zero; `GetSlice` returns raw multi-values.
- `ctx.BindJSON[T]()` is the recommended body reader: it decodes, runs the type's `Validate()` `error` automatically, and writes structured 4xx responses. `ctx.BindJSONOptional[T]()` accepts an empty body.
- The `ctx.Read*` family covers JSON, XML, YAML, MessagePack, Protobuf, forms, query, params and headers; `ctx.ReadBody` dispatches by content type and fills `*string` / `*[]byte` with the raw body.
- `ctx.BindPagination` and `ctx.Paginated` turn `?page=&size=` plus an optional filter body into a complete list endpoint.
- `ctx.UploadFormFiles` saves multipart uploads with sanitized names; `PostMaxMemory` bounds form buffering, `BodyMaxMemory` and the `bodylimit` middleware cap body size (413 on overflow).
- `ContextOptions.RequestValidator` is the app-wide hook for tag-based validation libraries; without it, `validate` tags are inert.

Further Reading

- MDN: HTTP messages. The anatomy of requests and responses: start line, headers, body.
- MDN: MIME types: how media types like `application/json` and `multipart/form-data` work.
- RFC 9110: HTTP Semantics: the authoritative definition of methods, headers and message bodies.
- RFC 8259: JSON: the complete JSON grammar; shorter than most tutorials about it.
- OWASP: Input Validation Cheat Sheet: server-side validation strategy, allowlists and common pitfalls.
- OWASP: File Upload Cheat Sheet: the risks behind upload endpoints and the checks that mitigate them.

CHAPTER 6

Response Handling

Iris gives you two layers for writing responses. The response helpers on the context are the recommended surface for JSON APIs: each takes the value and the error your service returned and decides what to send, including the error response when something went wrong. Underneath sit the `Context` writers (JSON, XML, YAML, text, files, streams) for the cases where you need full control over the bytes on the wire. This chapter covers both layers, plus status codes, response headers, redirects, streaming, Server-Sent Events, content negotiation and the handler that runs when a response write itself fails.

Background

An HTTP response mirrors the request's shape. Its first line (the status line) carries a three-digit status code and a short reason phrase ("200 OK", "404 Not Found"). Headers follow, describing what the body is (`Content-Type`), how large it is (`Content-Length`) and how it may be cached (`Cache-Control`). The body (HTML, JSON, file bytes) comes last, and some responses legitimately have none.

Status codes are grouped by their first digit, and the class alone tells a client how to react:

- **2xx**: success. 200 carries a body, 201 announces a created resource, 204 succeeds with nothing to return.
- **3xx**: redirection. The resource is elsewhere, or (304) the copy the client already holds is still fresh.
- **4xx**: the client made a mistake. Malformed input (400), missing credentials (401), no permission (403), no such resource (404).
- **5xx**: the server failed. The request itself may have been perfectly valid.

Treat the code you send as an API contract, because software you do not control acts on it. Browsers follow 3xx redirects on their own. Proxies and CDNs cache responses based on the code and headers: a 200 may be stored and replayed, a 500 will not be. HTTP clients and load balancers retry on some codes and give up on others. Monitoring counts 5xx responses to decide whether to page someone. Send a 200 with an error message hidden in the body and every one of those layers draws the wrong conclusion.

Content negotiation is the name for "one URL, several formats". The client's `Accept` request header lists the representations it can handle (`application/json`, `text/xml`, ...) with optional preference weights; the server compares that list against what it can produce and picks the best match, or answers 406 Not Acceptable when there is none. The same mechanism extends to compression (`Accept-Encoding`) and language (`Accept-Language`).

What this chapter's helpers remove from your code is the ceremony around those rules. Picking the right status for success and failure, shaping a consistent error body, remembering that headers must precede the first body byte. The response helpers encode those decisions once, so a typical handler ends with a single line instead of a block of status bookkeeping.

The Response Helpers

In a typical handler the response is the last line: call the service, then hand the result and the error to one helper. Every helper follows the same contract:

```go
ok := ctx.Helper(value, err) // ok == false: an error response was written
```

If `err` is non-nil, the helper routes it through the central error mapping of `rest/errors` (see Chapter 9) and a structured error response is written. Your handler never branches on the error. Otherwise the helper renders the success response. The returned `bool` is `true` on success; you only check it when more work follows the write.

```go
package main

import (
    "github.com/kataras/iris/v14"
)

type ArticlesAPI struct {
    svc *ArticleService // your application's service layer.
}

func NewArticlesAPI(svc *ArticleService) *ArticlesAPI {
    return &ArticlesAPI{svc: svc}
}
```

```
func (api *ArticlesAPI) Routing(r iris.Router) {
    r.Get("/", api.list)
    r.Post("/", api.create)
    r.Get("/{id:int64}", api.get)
    r.Put("/{id:int64}", api.update)
    r.Delete("/{id:int64}", api.delete)
}

func (api *ArticlesAPI) get(ctx iris.Context) {
    id, ok := ctx.BindParam[int64]("id")
    if !ok {
        return // error response already written.
    }

    article, err := api.svc.ByID(ctx, id)
    ctx.OK(article, err)
}

func (api *ArticlesAPI) create(ctx iris.Context) {
    input, ok := ctx.BindJSON[ArticleInput]()
    if !ok {
        return // 400 already written: malformed JSON or failed Validate.
    }

    id, err := api.svc.Create(ctx, input)
    ctx.Created(id, err) // 201 with {"id": ...}.
}

func (api *ArticlesAPI) update(ctx iris.Context) {
    id, ok := ctx.BindParam[int64]("id")
    if !ok {
        return
    }

    input, ok := ctx.BindJSON[ArticleInput]()
    if !ok {
        return
    }

    updated, err := api.svc.Update(ctx, id, input)
    ctx.NoContent(updated, err) // 204, or 304 if nothing changed.
}

func (api *ArticlesAPI) delete(ctx iris.Context) {
    id, ok := ctx.BindParam[int64]("id")
    if !ok {
        return
    }

    deleted, err := api.svc.Delete(ctx, id)
    ctx.NoContent(deleted, err)
}
```

The full set of response helpers:

Helper	Success status	Renders
`ctx.OK(resp, err)`	200 (status untouched)	`resp` as JSON; a nil `resp` sends nothing

`ctx.Created(resp, err)`	201	id scalars wrapped as `{"id": ...}`, other values as-is
`ctx.NoContent(modified, err)`	204 or 304	empty body: 204 when `modified` is true, 304 otherwise
`ctx.Count(rowsAffected, err)`	200	`{"count": N}`; zero is a valid answer, a negative count renders as zero
`ctx.List(items, err, tolerate...)`	200	`items` as a JSON array; empty and nil both render `[]`
`ctx.JSONOrError(resp)`	current	`resp` as JSON, with encode failures handled for you
`ctx.Paginated(collect)`	200	the standard page envelope

Two of the family are generic: `ctx.Count` infers the count's numeric type from its argument, and `ctx.Paginated` infers filter, item and count types from the collector you pass. Both follow the same contract as the others: the error routes through the central mapping, and `false` means a response was written.

`ctx.JSONOrError` is the low-level member of the family: it has no error argument, it simply calls `ctx.JSON` and converts an encoding failure into a proper error response instead of a broken body. Reach for it when the status code or the value needs preparing first.

What Created renders

`ctx.Created` recognizes identifier values and wraps them in a JSON object, so a create handler can return the new id directly from the service:

```go
type IDType interface {
    string | int | int64 | uint64 | uuid.UUID
}

type IDPayload[T IDType] struct {
    ID T `json:"id"`
}
```

A `string`, `int`, `int64`, `uint64` or `uuid.UUID` result becomes `{"id": ...}` (a UUID is rendered as its string form). The `uuid.UUID` here is the standard library's `uuid` package type (Go 1.27), the same one route macros and the book's examples use. A `[]string` whose elements are UUIDs becomes a JSON array of `{"id": ...}` objects, useful for bulk inserts. Any other value, such as the full created entity, is sent as-is. A nil value sends the bare 201.

What List renders

A list endpoint that is not paginated has two shapes to get right, and hand-written versions get both wrong. A nil slice marshals as `null` rather than `[]`, which every client has to special-case; and a store that reports "not found" for an empty result set turns an ordinary empty list into a 404. `ctx.List` answers both:

```go
func (api *ArticlesAPI) list(ctx iris.Context) {
    articles, err := api.svc.All(ctx)
    ctx.List(articles, err, sql.ErrNoRows) // 200 [] when there is nothing.
}
```

A nil or empty slice renders `[]`, never `null`, so the client can always iterate the answer. An error matching one of the tolerated sentinels (`errors.Is`, so a wrapped one matches) renders `[]` as well: a repository's not-found for an empty set is not a failure of the list endpoint. Any other error goes through the central error mapping and `List` reports `false`, exactly like the rest of the family. Passing something that is neither a slice nor an array is a programmer error and panics naming the type, rather than rendering a JSON object where the client expects an array.

Reach for `ctx.Paginated` instead as soon as the list can grow: `List` sends the whole set.

Pagination in one call

`ctx.Paginated` runs the complete listing flow: it reads `?page=` and `?size=` from the query and an optional JSON filter object from the request body, calls your collector with the parsed options, and renders the page envelope.

```go
import (
    stdContext "context"

    "github.com/kataras/iris/v14"
)

func (api *ArticlesAPI) list(ctx iris.Context) {
    ctx.Paginated(func(reqCtx stdContext.Context,
        opts iris.PageOptions, filter ArticleFilter) ([]Article, int64, error) {
        return api.svc.List(reqCtx, opts, filter)
    })
}
```

The collector returns the requested window of items and the total row count; everything else (limit/offset math, the envelope, error handling) is done for you. The response shape:

```json
{
    "current_page": 1,
    "page_size": 20,
    "total_pages": 3,
    "total_items": 42,
```

```
    "has_next_page": true,
    "filter": null,
    "items": [{"id": 1, "title": "First"}]
}
```

Request reading (`ctx.BindJSON` , validation, `ctx.BindParam`) is covered in Chapter 5.

Context Writers

When a helper does not fit (a non-JSON format, a custom shape, a template), drop down to the `Context` writers. Each one sets the matching Content-Type and writes the body:

Method	Content-Type	Options value
`ctx.JSON(v, ...)`	application/json	`iris.JSON`
`ctx.JSONP(v, ...)`	text/javascript	`iris.JSONP`
`ctx.XML(v, ...)`	text/xml	`iris.XML`
`ctx.YAML(v)`	application/x-yaml	—
`ctx.TextYAML(v)`	text/yaml	—
`ctx.Markdown(b, ...)`	text/html	`iris.Markdown`
`ctx.Text(s)`	text/plain	—
`ctx.HTML(s)`	text/html	—
`ctx.Binary(b)`	application/octet-stream	—

`JSON` , `JSONP` , `XML` and `Markdown` accept an optional options value as their last argument; the rest take none. Every one of them returns a single `error` , nil when the body was written.

JSONP is a legacy format: the JSON is wrapped in a JavaScript function call so that browsers predating cross-site request rules could load it through a `<script>` tag. New APIs rarely need it.

JSON

GO

```go
func report(ctx iris.Context) {
    data := iris.Map{"name": "Jane", "score": 9001}

    ctx.JSON(data, iris.JSON{
        Indent:         "  ",  // pretty-print with two spaces.
        EscapeHTML:     false, // true escapes <, >, & inside strings.
        Prefix:         "",    // optional string written before the body.
        EscapeNonASCII: false, // escape non-ASCII runes when true.
        ArrayPrefix:    false, // prepend "while(1);" to top-level arrays.
```

```
    })
}
```

Without options, `ctx.JSON(v)` streams the value with the default encoder, which is the standard library's `encoding/json/v2` (Go 1.27). Two of its defaults are worth knowing. HTML characters inside strings are *not* escaped: responses are served as `application/json`, never inline HTML, so the escaping buys nothing and costs bytes; set `EscapeHTML: true` to restore the old `encoding/json` behavior (JSONP always escapes, because there the payload does land in a script body). And nil slices and maps render as `[]` and `{}` rather than `null`, so list endpoints never make clients branch on `null`. `time.Duration` still encodes as int64 nanoseconds. For anything else the struct's `MarshalOptions` field accepts raw `encoding/json/v2` options, appended after the framework defaults so yours win. `iris.Map` is a shorthand for `map[string]any`.

Those framework defaults live in one place, `core/pkg/jsonutil`, and every part of Iris that touches JSON composes its options from there: the context writers and readers, the `rest/errors` renderers, the SSE and AI encoders, the access log and the HTTP client. That is why a value encoded by the client and decoded by a handler cannot disagree about what the wire looks like.

XML

Structs marshal through `encoding/xml`. Maps need the `iris.XMLMap` wrapper because Go maps have no element name:

```go
ctx.XML(payload, iris.XML{Indent: "  "})
ctx.XML(iris.XMLMap("Root", iris.Map{"key": "value"}))
```

Markdown

`ctx.Markdown` parses the given markdown bytes to HTML and serves the result as text/html. Set `Sanitize` when the source is user-provided:

```go
ctx.Markdown([]byte("# Release Notes"), iris.Markdown{Sanitize: true})
```

Raw writes

For output that needs no content-type handling, write directly:

```go
ctx.WriteString("plain body")
ctx.Writef("hello, %s", name)
ctx.Write([]byte{0xCA, 0xFE})
```

HTML template rendering goes through the `middleware/view` package (`view.Handler` plus `view.Render`) and is covered in Chapter 12.

Status Codes

`ctx.StatusCode` sets the response status; `ctx.GetStatusCode` reads what has been set so far. The status must be set before the first body write; writing the body flushes the status line and headers to the wire:

```go
func createLegacy(ctx iris.Context) {
    ctx.StatusCode(iris.StatusCreated) // 201, before the body.
    ctx.JSON(iris.Map{"id": 42})
}
```

That handler is named for what it is. `ctx.Created(42, err)` from The Response Helpers above sets the same status, wraps the same `{"id": 42}` and routes the error, so a real create endpoint never sets a status by hand.

Iris exposes the full set of constants: `iris.StatusOK`, `iris.StatusCreated`, `iris.StatusNoContent`, `iris.StatusBadRequest`, `iris.StatusNotFound`, `iris.StatusInternalServerError` and the rest. `iris.StatusText(code)` returns the standard reason phrase.

The `StopWith` family sets a status, writes an optional body and stops the handler chain in one call (following `ctx.Next` calls are ignored):

Method	Body
`ctx.StopWithStatus(code)`	none
`ctx.StopWithText(code, text)`	the given text
`ctx.StopWithJSON(code, v)`	`v` as JSON
`ctx.StopWithError(code, err)`	`err.Error()` as text; none when `err` is private
`ctx.StopWithPlainError(code, err)`	none; the error is stored for error handlers

`StopWithError` and `StopWithPlainError` also record the error through `ctx.SetErr`, so middleware running after the handler can inspect it with `ctx.GetErr`, or with `ctx.ErrPublic`, which reports whether the stored error is safe to show the client and hands back the original either way. If the error is private, because `iris.PrivateError` or `errors.Private` marked it or it wraps such an error anywhere in its chain, `StopWithError` withholds the message, sends only the status and stores the error as private. The “none” in the table is what the method writes. On an application

built with `iris.NewBuilder()` the response does not stay empty: `errors.Handler` fills a failure status that nobody gave a body to with the JSON error envelope after the chain returns. For API error responses prefer the structured senders of `rest/errors` (Chapter 9); the `StopWith` methods are the low-level tools they are built on.

Response Headers

`ctx.SetHeader` writes a response header; an empty value deletes it. Request and response access are separate methods, a common source of confusion:

Read the request	Read the response
`ctx.Header("X-Key")`	`ctx.ResponseHeader("X-Key")`
`ctx.ContentType()`	`ctx.ResponseContentType()`
`ctx.MediaType()`	`ctx.ResponseMediaType()`

```go
func cached(ctx iris.Context) {
    ctx.SetHeader("Cache-Control", "public, max-age=3600")
    ctx.SetHeader("X-API-Version", "1")

    ctx.JSON(iris.Map{"ok": true})

    // Reads back what this handler set, not what the client sent:
    _ = ctx.ResponseHeader("Cache-Control") // "public, max-age=3600"
    _ = ctx.ResponseMediaType()             // "application/json"
}
```

`SetHeader` replaces any value already set for that name, so a middleware registered twice cannot emit the header twice. When a list is what you mean, `ctx.AddHeader` appends instead, and the underlying header map is always there for anything else:

```go
ctx.AddHeader("Vary", "Accept-Encoding")
ctx.AddHeader("Vary", "Accept-Language")

ctx.ResponseWriter().Header().Set("X-Custom", "only-value")
ctx.ResponseWriter().Header().Add("Link", "</style.css>; rel=preload")
```

`ctx.SetContentType` sets the Content-Type and returns an error for invalid values; the writers above call it for you, so you only need it with raw writes. Like the status code, headers must be in place before the first body byte; anything set afterwards is silently dropped by `net/http`.

Redirects

A redirect is a response with a 3xx status and a `Location` header naming another URL; a browser (or any client configured to follow redirects) reacts by requesting that URL instead. The code chooses the semantics: 301 and 308 mean “moved for good, update your links” and may be cached by browsers for a long time, while 302, 303 and 307 mean “go here this time”.

`ctx.Redirect` sends a redirect and stops the handler chain. The status code argument is optional and defaults to 302 (`iris.StatusFound`):

```go
func oldEndpoint(ctx iris.Context) {
    ctx.Redirect("/v2/endpoint", iris.StatusMovedPermanently) // 301
}

func afterLogin(ctx iris.Context) {
    ctx.Redirect("/dashboard") // 302 by default.
}
```

Use `iris.StatusTemporaryRedirect` (307) or `iris.StatusPermanentRedirect` (308) when the client must replay the request with the same method and body, and `iris.StatusSeeOther` (303) to turn a POST into a follow-up GET. A non-3xx status set earlier on the context is ignored: redirect responses are always sent with a redirect code.

File Responses

Three methods cover files, differing in where the content comes from and how the browser treats it:

```go
// Inline: render in the browser (assets, images, PDFs to display).
func manual(ctx iris.Context) {
    if err := ctx.ServeFile("./documents/manual.pdf"); err != nil {
        return // 404, 403 or 500 status already set.
    }
}

// Attachment: force a download dialog with the given file name.
func export(ctx iris.Context) {
    ctx.SendFile("./exports/report-2026.csv", "report.csv")
}

// From memory or any io.ReadSeeker, with a modification time.
func generated(ctx iris.Context) {
    content := bytes.NewReader(pdfBytes)
    ctx.ServeContent(content, "invoice.pdf", time.Now())
```

```
}
```

`SendFile` is `ServeFile` plus a `Content-Disposition: attachment` header; it derives the download name from the source path when the second argument is empty. `ServeContent` delegates to `http.ServeContent`, so Range requests, `If-Modified-Since` and ETag-based conditional requests work out of the box.

Every method has a rate-limited variant that throttles the transfer. `limit` is bytes per second, `burst` the bucket size in bytes:

```go
ctx.ServeFileWithRate("./files/video.mp4", 1024*1024, 64*1024) // ~1 MB/s
ctx.SendFileWithRate("./files/big.zip", "big.zip", 256*1024, 32*1024)
ctx.ServeContentWithRate(content, "data.bin", modTime, 512*1024, 32*1024)
```

Conditional requests are how HTTP avoids resending unchanged content: the client remembers a fingerprint of the copy it holds (its modification time, or an ETag, an opaque version tag chosen by the server) and sends it back on the next request, meaning “only send the body if this changed”. When nothing changed, the server answers 304 Not Modified with an empty body and the client keeps its copy. Range requests are the related trick for large files: the client asks for a byte slice instead of the whole file, which is what makes download resumption and video seeking work.

For conditional non-file responses, the same primitives `http.ServeContent` uses are available directly: `ctx.SetLastModified(t)` sets the header, `ctx.CheckIfModifiedSince(t)` compares against the request, `ctx.WriteNotModified()` sends a 304, and `ctx.WriteWithExpiration(body, t)` combines all three into one call.

Serving whole directories is the job of the `controller/fileserver` package, covered with its caching options in Chapter 14.

Streaming

Normally the server buffers a response and sends it when the handler returns. Flushing changes that: each chunk is pushed to the client the moment it is written, turning the connection into a live feed.

For a response produced over time (a progress feed, a large export), write in chunks and flush each one to the client:

```go
func progress(ctx iris.Context) {
    ctx.SetContentType("text/plain")
```

```
    for i := 1; i <= 5; i++ {
        ctx.Writef("step %d done\n", i)
        ctx.ResponseWriter().Flush() // push the chunk now.
        time.Sleep(time.Second)
    }
}
```

`ctx.StreamWriter` packages the same loop: it calls your function, flushes, and calls it again until you return a non-nil error or the client goes away. Return `nil` to continue with the next chunk; return an error (`io.EOF` is the conventional "nothing more to send") to end the stream:

GO

```
import (
    "fmt"
    "io"
    "time"

    "github.com/kataras/iris/v14"
)

func countdown(ctx iris.Context) {
    ctx.SetContentType("text/plain")

    ticker := time.NewTicker(time.Second)
    defer ticker.Stop()

    left := 10
    ctx.StreamWriter(func(w io.Writer) error {
        if left == 0 {
            return io.EOF // any non-nil error ends the stream.
        }

        <-ticker.C
        left--
        _, err := fmt.Fprintf(w, "%d seconds left\n", left)
        return err // nil: flush this chunk and call me again.
    })
}
```

When the client disconnects, `StreamWriter` returns the request context's cancellation error, so the goroutine does not leak. In hand-rolled loops, check `ctx.IsCanceled()` between chunks for the same guarantee.

`StreamWriter` is a poll loop: the framework asks for the next chunk when it is ready for one. That suits a writer which blocks until it has something to say (a ticker, a queue read, a database cursor) and fits badly when the events arrive from a producer goroutine on their own schedule. For Server-Sent Events, the next section is the better path in both shapes.

One trap applies to every stream, whichever way you write it. Two of the bundled middleware buffer the body: `middleware/compress` writes into a compressor, and `middleware/recorder` writes into a `bytes.Buffer`, which `accesslog` installs whenever it reads the response body, and which

`iris.NewBuilder().LogRequests(true)` therefore turns on. Neither forwards anything when you flush, so the handler streams into a buffer and the client sees one lump on return. `ctx.DisableResponseBuffering()` removes those wrappers for the current response, before the first write, and returns an error naming any writer that refused to step aside. The `sse` package calls it for you.

Server-Sent Events

Server-Sent Events (SSE) is a one-way stream of text frames sent over an ordinary HTTP response. A frame is a group of `field: value` lines ended by a blank line, and a line beginning with a colon is a comment:

```
event: token
data: Hello

: keep-alive
```

Four fields are defined. `event` names the event and, when absent, the client receives it as `message`; `data` carries the payload; `id` becomes the client's resume cursor; `retry` sets how long the client waits before reconnecting. Browsers consume the format natively through `EventSource`, which reconnects on its own and sends the last id it saw back in a `Last-Event-ID` header. Everything else is ordinary HTTP: status codes, authorization headers and proxies all behave as they do for any other response.

Writing those frames by hand is two lines of `fmt.Fprintf`, which is why so many services do, and why so many of them fail in the same four places. A buffering middleware holds the frames. The server's absolute write timeout ends the response mid-answer. A reverse proxy buffers it anyway. And a client that vanishes without closing its connection is never noticed, so the work behind the stream runs on for nobody. The `sse` package exists for those four, and the rest of this section is what it does about them.

Opening a stream

`sse.Open` prepares the response and writes nothing:

```go
import (
    "fmt"
    "time"

    "github.com/kataras/iris/v14"
    "github.com/kataras/iris/v14/rest/errors"
```

```
    "github.com/kataras/iris/v14/sse"
)

func events(ctx iris.Context) {
    stream, err := sse.Open(ctx)
    if errors.HandleError(ctx, err) {
        return // ErrNotStreamable renders as a 500; nothing has been written.
    }
    // Close writes the terminal frame, on success as well as on failure.
    defer stream.Close(nil)

    ticker := time.NewTicker(time.Second)
    defer ticker.Stop()

    for i := 1; i <= 10; i++ {
        select {
        case <-ctx.ShuttingDown():
            return // the server is stopping: end while we still can.
        case <-stream.Context().Done():
            return // the client is gone.
        case <-ticker.C:
        }

        if err := stream.Send(sse.Event{
            Name: "tick",
            Data: fmt.Appendf(nil, "tick %d", i),
        }); err != nil {
            return
        }
    }
}
```

That `Open` writes nothing is the property the rest of the design rests on. The protocol headers commit on the first `Send`, so a check that runs after opening can still refuse the request properly, with a status code and a body:

```
func guarded(ctx iris.Context) {
    stream, err := sse.Open(ctx)
    if errors.HandleError(ctx, err) {
        return
    }
    defer stream.Close(nil)

    if ctx.Query().Get("token") == "" {
        // A real status code and a real body: not one byte has gone out.
        errors.Unauthenticated.Message(ctx, "token required")
        return
    }

    stream.Send(sse.Event{Name: "welcome", Data: []byte("you are in")})
}
```

A handler that opens a stream and returns without sending anything leaves the response untouched as well, which is what lets an error handler further up the chain render it.

`Open` does check one thing up front: whether this response can flush at all. It cannot under `Timeout.HandlerTimeout`, which wraps the router in `http.TimeoutHandler`, whose response writer implements neither `http.Flusher` nor `http.Hijacker`. Every write buffers and the client receives a 503 page instead of a stream. Because the answer is otherwise only available at flush time, which is far too late, `Open` reports `sse.ErrNotStreamable` with zero bytes written and the handler can still answer honestly. Setting `HandlerTimeout` on an application that streams anything logs a warning at startup for the same reason.

On the first `Send`, and for that one response, five adjustments land that no middleware could guess were wanted:

- every buffering response-writer wrapper is removed, so compression and the recorder that `accesslog` installs stop holding the events;
- the server's absolute `WriteTimeout` is cleared. It is a total deadline rather than an idle one, so it would otherwise end a long answer mid-sentence;
- full duplex is enabled on HTTP/1, so a request body can still be read after the response has started, which is how a POST-driven stream is called;
- `Cache-Control: no-cache, no-transform` and `X-Accel-Buffering: no` are set and any `Content-Length` is deleted, which is what stops nginx and the CDNs in front of it from buffering the stream into one response;
- `Connection: keep-alive` is set on HTTP/1 only, where it is meaningful. It is a protocol error on HTTP/2.

Sending events

`Send` writes one frame and flushes it. `SendJSON(name, v)` marshals a value into the data field, and `Comment` writes a comment-only frame. All of them are safe to call from more than one goroutine: the heartbeat takes the same lock, so a keep-alive can never interleave with a half-written frame.

`sse.Event` is the frame itself, and a zero field is simply absent from the wire:

Field	Wire field	Notes
`ID`	`id`	Becomes the client's `Last-Event-ID`. Stamped for you while `AutoID` is on
`Name`	`event`	Empty means `message`, which is what an `EventSource` `onmessage` handler receives
`Data`	`data`	A multi-line value is split across several `data` fields and rejoined by the client with newlines
`Retry`	`retry`	The client's reconnection delay, written once with the first event

`Comment` `:` Written before the fields. A nil `Data` with a comment dispatches no event at all

`ID`, `Name` and `Comment` are rejected with `sse.ErrInvalidField` when they contain CR, LF or NUL. Those values are terminated by a newline on the wire, so without the check a caller-supplied id forges frames of its own.

Closing, and the frame that says so

`Close` writes a final `done` event and releases the stream. It is idempotent, so a deferred `Close(nil)` and an explicit `Close(err)` coexist, and a panicking handler still terminates the stream cleanly.

It writes that frame on success too, and that is the point. If the terminator is always present, its absence is how a client detects truncation. Nothing below it offers that signal: chunked framing catches a torn connection, but a proxy idle timeout closes the stream cleanly and the client cannot tell a finished answer from a severed one. A browser client closes on it explicitly, since `EventSource` would otherwise reconnect:

```JS
const es = new EventSource("/events");
es.addEventListener("tick", (e) => console.log(e.data));
es.addEventListener("done", () => es.close()); // the terminal frame.
```

When `Close` is given an error and nothing has been committed, it writes nothing and returns the error, so the caller renders an ordinary HTTP response. After the first `Send` it writes an error event and then the terminator, because the status code is already spent. `stream.Committed()` reports which side of that line you are on, and `stream.Err()` reports why a stream ended.

What goes into that error event is the `ErrorFrame` option, and its default is the reason a failed stream is not a special case. `sse.ReportErrorFrame` runs the error through `errors.Report`, the `rest/errors` pipeline with the rendering step removed: the route's error mappings apply, the logger and the collector fire, a correlation id is generated, and an unmapped or private error is redacted to a generic message before anything reaches the client. The payload is the same `rest/errors` envelope an ordinary failure would have produced, so one client parser handles a failure whether it arrived before the first byte or after it.

The redaction is the part worth noticing. Half of what a streaming handler can fail on is somebody else's text, and an upstream message naming a vendor, a model or an internal host has no business on the wire. `sse.RawErrorFrame` sends the error's own words instead, which is appropriate only when you know where those words came from. A frame is skipped entirely when there is nothing to report, which is the case for a client that already disconnected while `errors.SkipCanceled` is on: there is nobody left to tell.

The options, and where their defaults come from

`sse.Open(ctx, opts)` takes an `sse.Options`. Start from `sse.DefaultOptions()` and change what you need: a zero field means off rather than default, so switching the heartbeat or the terminator off stays expressible.

Option	Default	What it is for
`Heartbeat`	`15s`	The longest the stream may go without writing anything
`HeartbeatFrame`	`": ping\n\n"`	What the heartbeat writes. A comment, which conforming clients ignore
`Retry`	`3s`	Advertised to the client once, with the first event
`AutoID`	`true`	Stamps an increasing id on every event that has none, so a gap is detectable within one connection. Not resumption: the counter restarts at 1 on every stream
`Terminator`	`"done"`	The event name `Close` writes
`ErrorFrame`	`sse.ReportErrorFrame`	Renders the error event `Close` writes when a committed stream failed
`SendTimeout`	`30s`	A write deadline for one `Send`. Exceeding it fails the stream with `sse.ErrSlowClient`

Two of those defaults carry an argument worth reading. Fifteen seconds is not a round number: it is the largest heartbeat that survives every common intermediary at once, since nginx `proxy_read_timeout` and an AWS Application Load Balancer both idle out at 60 seconds, Cloudflare returns a 524 at 100, and Heroku's router applies a rolling 55. The heartbeat's larger job is liveness rather than keep-alive: only a real FIN or RST, or a failing write, cancels a request context, so a client that disappears without one (a closed lid, a dropped link, an evicted NAT entry) stays invisible until TCP gives up, which on Linux is roughly 924 seconds of leaked goroutine and leaked upstream work. Writing something is what lets the kernel fail and the handler find out. A Google Cloud load balancer is the exception no heartbeat rescues, because it applies its backend timeout to the whole request and response; raise `timeoutSec` there instead.

`SendTimeout` is the backpressure mechanism, and it is a deadline rather than a queue on purpose. `Send` is synchronous, so there is no buffer of the framework's to budget, and a client that will not drain shows up as a write that does not complete. Exceeding it fails the stream and never skips an event, because a dropped delta corrupts the message the client is assembling with no way for it to notice.

`HeartbeatFrame` is configurable rather than fixed because not every client conforms. `sashabaranov/go-openai`, widely used against OpenAI-compatible servers, counts comment frames toward its `emptyMessagesLimit` and fails the stream instead of ignoring them. Serving such a client, send a real event.

Batching deltas

A fast producer is a syscall amplifier. One frame per token means one write, one flush, one TCP segment and one client-side render per token, and the tokens arrive faster than any of those is worth doing. `Coalesce` batches the events you name and emits at most one frame per name per interval, concatenating their payloads:

```go
batch := stream.Coalesce(200*time.Millisecond, "word")
defer batch.Close()

for _, word := range strings.Fields("the quick brown fox") {
    ev := sse.Event{Name: "word", Data: []byte(word + " ")}
    if err := batch.Send(ev); err != nil {
        return
    }

    time.Sleep(80 * time.Millisecond)
}
```

A background ticker drains the buffers, which is what makes the interval a ceiling rather than a floor. Without one, a buffered delta is only emitted when the next one arrives, so text that already reached the server sits unsent through exactly the mid-answer pause a reader notices.

Because the payloads are concatenated, this is for text-like deltas (running prose, a reasoning trace) and not for events whose data is a self-contained JSON document. Anything not named passes straight through, after the pending buffers are flushed, so ordering holds. The coalescer owns the stream while it is open and `Send` on the stream itself flushes it first, so the interleaving hazard a hand-rolled batcher has to document cannot be expressed here.

Resuming after a reconnect

`stream.LastEventID()` reports the cursor the client sent, from the `Last-Event-ID` header or the `lastEventId` query parameter. Both are load-bearing: a browser `EventSource` sends the header and cannot be made to send anything else, while a client built on `fetch` usually has to put it in the URL. The value is sanitised, because it is echoed back on every frame.

Where the events between that cursor and now come from is the application's business. The package stores nothing, which is what keeps it small enough to be correct: a replay buffer belongs where the data already lives.

Reading somebody else's stream

`sse.Read` is the other half, for consuming an event stream from an upstream service. It is the framework's first `iter.Seq2` API, so it ranges:

```go
resp, err := httpClient.Do(req)
if err != nil {
    return err
}
defer resp.Body.Close()

for ev, err := range sse.Read(resp.Body, sse.ReadOptions{
    Terminators: []string{"[DONE]"},
}) {
    if err != nil {
        return err
    }

    fmt.Println(ev.Name, string(ev.Data))
}
```

It implements the parsing algorithm rather than matching a prefix, which matters more than it sounds: `data:` with and without a space are both legal, and a reader that matches `data:` sees an empty stream from a producer that omits it. Multi-line data, comments, unknown fields, a leading byte order mark and lines ended by CR, LF or CRLF are all handled. It never yields a partially framed event, because an event is dispatched by the blank line that ends it; a stream cut mid-event ends the iteration with `io.ErrUnexpectedEOF` rather than quietly, as if the answer were complete. A clean end of stream simply ends the iteration, and `io.EOF` is not reported as an error.

`ReadOptions.Terminators` ends the iteration on an in-band sentinel such as `[DONE]`, which the OpenAI dialect and everything shaped after it emit. It is a concession on the reading side and never something this package writes, for the reason its own terminator is an event name: a sentinel travels on the same channel as the payload, so a producer that legitimately emits the literal string would end the stream. `RequireTerminator` goes with it: when the upstream promised a sentinel and closed without one, the iteration ends with `sse.ErrNoTerminator`, because a stream that ends short was cut however clean the close looked to the transport. `MaxEventBytes` caps one event, not the stream, at 1 MB by default, and `KeepComments` yields an upstream's own heartbeats instead of skipping them.

Relaying a model's stream is what the `ai` package is for, and it is built on both halves of this one. See Chapter 21.

Content Negotiation

When one endpoint must serve multiple representations, declare what the server offers with `ctx.Negotiation()`, then let `ctx.Negotiate` pick the best match for the request's `Accept` header:

```go
type User struct {
    ID   int    `json:"id" xml:"id"`
    Name string `json:"name" xml:"name"`
}

func user(ctx iris.Context) {
    // Server-side offers, in priority order.
    ctx.Negotiation().JSON().XML().Text()

    u := User{ID: 42, Name: "Jane"}
    ctx.Negotiate(iris.N{
        JSON: u,
        XML:  u,
        Text: fmt.Sprintf("user %d: %s", u.ID, u.Name),
    })
}
```

`iris.N` holds one value per offered representation (`Text`, `HTML`, `Markdown`, `Binary`, `JSON`, `JSONP`, `XML`, `YAML`, `Protobuf`, `MsgPack`, plus `Other` for custom types); the matched field is rendered with the corresponding writer. If nothing matches, `Negotiate` sets 406 Not Acceptable and returns `context.ErrContentNotSupported`. The builder can also carry the content itself (`ctx.Negotiation().JSON(v).XML(v)` followed by `ctx.Negotiate(nil)`) and offers `Charset(...)` and `EncodingGzip()` to negotiate charset and compression alongside the media type.

The Response Error Handler

The writers report failures (a value JSON cannot encode, a broken connection) back to the caller as an `error`. In practice handlers do not check that return value; instead, every failed write also invokes the context error handler, configured at `Options.Context.ErrorHandler`. The default logs the error in debug mode and replies with a bare 500.

Replace it application-wide through the options (partial options merge with the defaults, see Chapter 11):

```
    "github.com/kataras/iris/v14/rest/errors"
)

func main() {
    app := iris.New(iris.Options{
        Context: iris.ContextOptions{
            ErrorHandler: func(ctx iris.Context, err error) {
                // Logs the write failure and answers a canonical 500; the
                // encoder's own message never reaches the client.
                errors.Internal.LogErr(ctx, err)
            },
        },
    })

    app.Get("/", func(ctx iris.Context) {
        ctx.JSON(iris.Map{"message": "hello"})
    })

    app.Listen(":8080")
}
```

For a single route or group, `iris.WithErrorHandler` builds a middleware that overrides the handler for the requests it runs on. It calls `ctx.Next()` itself, so register it like any other middleware and the chain carries on:

```
app.Use(iris.WithErrorHandler(func(ctx iris.Context, err error) {
    errors.Internal.ErrPrivate(ctx, err)
}))
```
GO

It works the same in front of a single route:

```
app.Get("/report", iris.WithErrorHandler(reportErrors), reportHandler)
```
GO

Applications built with `iris.NewBuilder` get `errors.DefaultContextErrorHandler` from `rest/errors` installed automatically, so write failures render as the same structured error JSON as every other error in the app. Setting `Options.Context.DisableErrorHandler: true` turns the mechanism off entirely, leaving the returned `error` as the only signal.

Summary

- The Context helpers are the recommended way to respond in JSON APIs: one contract, `ctx.Helper(value, err)`, with `ctx.OK`, `ctx.Created` (id values become `{"id": ...}`), `ctx.NoContent`, `ctx.JSONOrError`, `ctx.Count` and `ctx.Paginated`, covering the standard cases and routing every error through the central mapping.

- The `Context` writers handle the formats directly: `JSON`, `JSONP`, `XML` and `Markdown` accept an options value; `YAML`, `TextYAML`, `Text`, `HTML` and `Binary` take none.
- Set status codes with `StatusCode` before the body; the `StopWith` family combines status, body and stopping the chain.
- `SetHeader` replaces and `AddHeader` appends a response header, and `ResponseHeader` reads one back, mirroring `ctx.Header` for the request side; `ResponseContentType` and `ResponseMediaType` pair with the request's `ContentType` and `MediaType`.
- `Redirect` defaults to 302; `ServeFile`, `SendFile` and `ServeContent` cover inline files, downloads and in-memory content, each with a rate-limited variant.
- Stream with `Write` plus `ResponseWriter().Flush()`, or let `StreamWriter` run the flush loop; end the stream by returning an error. `ctx.DisableResponseBuffering()` removes the middleware wrappers that would otherwise hold the chunks.
- The `sse` package writes Server-Sent Events: `sse.Open` commits nothing until the first `Send`, so a check after it can still return a status code; that first `Send` unbuffers the response, clears the absolute `WriteTimeout` and sets the anti-buffering headers; `Close` writes the terminal frame on success too, so its absence means truncation. `Coalesce` batches deltas and `sse.Read` consumes an upstream stream.
- A failure after the first byte is reported in-band by the `ErrorFrame` option, whose default runs the error through the `rest/errors` pipeline, so it is mapped, logged, collected and redacted like any other failure.
- `Negotiation()` and `Negotiate(iris.N{...})` serve multiple representations from one endpoint.
- Failed writes trigger `Options.Context.ErrorHandler`; the `iris.WithErrorHandler` middleware overrides it for the routes it runs on, and builder apps default to the structured `rest/errors` handler.

Further Reading

- MDN: HTTP response status codes: every status code with its intended use.
- MDN: Content negotiation: how `Accept` headers drive representation selection.
- MDN: Redirections in HTTP: the 3xx family and when to use each code.
- MDN: HTTP conditional requests: ETag, Last-Modified and the 304 mechanism in detail.
- MDN: Using server-sent events: the SSE wire format and the browser-side `EventSource` API.
- WHATWG HTML: Server-sent events: the normative field list and the parsing algorithm `sse.Read` implements.

- RFC 9110: HTTP Semantics: the authoritative source for status codes, headers and negotiation.

CHAPTER 7

Middleware

Iris has exactly one handler type, `func(ctx iris.Context)`, and middleware is nothing more than a handler that decides whether the rest of the chain runs. That single decision (call `ctx.Next()` or stop) powers authentication guards, request logging, panic recovery and everything in between. This chapter explains the handler chain, the registration methods and their exact scopes, how to write your own middleware, and the ordering rules that determine what runs when. The catalog of ready-made middleware packages lives in Chapter 12; this chapter is about the machinery they all share.

Background

Every non-trivial web service repeats certain work on every request: write an access log line, check credentials, attach an ID for tracing, reject oversized bodies, recover from crashes. These are cross-cutting concerns: work that belongs to no single endpoint but applies across all of them. Copying that work into every handler multiplies it by the number of endpoints and guarantees the copies drift apart; the login check that was fixed in nine handlers and forgotten in the tenth is a classic security hole.

Middleware is the pipeline answer to that problem. Instead of each handler doing everything, the framework assembles a chain of small functions around every request; each one does its piece and passes control onward. Think of the layers of an onion: the request travels inward through logging, authentication and limits before reaching the handler at the core, and the response travels back out through the same layers in reverse, which is why a timing middleware can note the clock on the way in and compute the duration on the way out. You write each concern once, register it once, and every route beneath it is covered.

The chain has a second power: any layer can stop the request. An authentication guard that finds no valid token writes a 401 and simply does not pass control on: the handler never runs, and neither does any code that assumed a valid user. Ordering follows from the same logic: the

cheap protective layers (rate limits, body caps) belong outside the expensive ones.

One distinction in this chapter deserves a plain definition up front. Routing is the step where the framework matches a request's method and path against the registered routes to pick a handler; the component doing the matching is called the mux (short for multiplexer). "Before routing" middleware (`UseRouter`) runs before that lookup, so it sees every request, including those that match nothing and will end as a

1. "Per route" middleware (`Use` and friends) is baked into each matched route's chain, so it runs only when routing found a handler. Practically: access logs and request IDs belong before routing, because a request that matches nothing still deserves a log line; a guard that loads the current user belongs on the routes that need a user.

Handlers and the Chain

Every request runs through a chain of handlers assembled for its route. Each handler receives the same `iris.Context` and controls the chain with three moves:

```go
func example(ctx iris.Context) {
    // 1. Work before the rest of the chain.

    ctx.Next() // 2. Run the remaining handlers.

    // 3. Work after the chain below has completed
    //    (post-processing: timing, logging, metrics).
}
```

Not calling `ctx.Next()` ends the chain quietly: the following handlers simply never run. `ctx.StopExecution()` goes further: it marks the context as stopped, so any later `Next` call anywhere in the chain becomes a no-op. `ctx.IsStopped()` reports that state. The `StopWith*` shortcuts combine stopping with a response:

- `ctx.StopWithStatus(code)`: status code only.
- `ctx.StopWithText(code, text)`: plain text body.
- `ctx.StopWithJSON(code, value)`: JSON body.
- `ctx.StopWithError(code, err)`: records the error and writes its message, or only the status when the error is private.
- `ctx.StopWithPlainError(code, err)`: records the error without writing a body, leaving rendering to an error handler such as the `rest/errors` pipeline (Chapter 9).

A “middleware” is a handler registered before others: by position in the route line, or through one of the registration methods below. There is no separate middleware interface to implement.

The Registration Family

Where you register a handler determines which requests it sees. The methods, with their exact scopes as implemented in `core/router/router_builder.go`:

Call	Runs for
`app.UseRouter(h)`	Every request, before routing, including 404/405
`r.UseRouter(h)` (child)	Every request under the child’s path prefix or subdomain, before routing
`app.Use(h)` (root)	Every matched route, plus the 404 and 405 chains
`r.Use(h)` (child)	Matched routes of that child and its children only
`app.UseRelevant(h)`	Matched routes only, never 404/405
`app.UseOnce(h)`	Like root `Use`, but insert-or-replace
`app.UseRelevantOnce(h)`	Like `UseRelevant`, but insert-or-replace

One caveat on the 405 rows: the no-method chain only runs when you set `Router.FireMethodNotAllowed`, which defaults to false. Until you do, a wrong-method request goes to the 404 chain instead.

UseRouter: before routing

`UseRouter` handlers run before the mux looks at the path. They fire for every request (matched, not found, or method not allowed), which makes them the right place for access logging, request IDs, CORS, compression and rate limiting. Because they run first, they also hold veto power: a `UseRouter` handler that writes a response without calling `ctx.Next()` prevents routing from ever happening.

CORS (Cross-Origin Resource Sharing) shows why “every request” matters. It is the browser mechanism that decides whether a page on one site may call your API on another; before certain cross-site requests the browser sends an automatic `OPTIONS` “preflight” request asking for permission. A preflight often matches no registered route, so a CORS handler attached per route would never see it. Registered with `UseRouter`, it answers the preflight before routing can 404 it.

Called on a child router, `UseRouter` still registers at the top level but wraps the handler in a prefix (and subdomain) check, so it fires only for requests under that child's path, still before routing.

Use: root versus child

The root-versus-child difference in `Use` is the one that surprises people. On the root application, `Use` handlers join every route's chain *and* are prepended to the no-route (404) and no-method (405) chains, so they observe requests that match nothing at all. On a child router, `Use` delegates to `UseRelevant`: the handlers join only the routes registered under that child.

```go
package main

import (
    "github.com/kataras/iris/v14"
)

func main() {
    app := iris.New()

    app.UseRouter(func(ctx iris.Context) {
        // Before routing: fires for "/", "/admin/panel"
        // and any 404 or 405 request.
        ctx.Next()
    })

    app.Use(func(ctx iris.Context) {
        // Root Use: all matched routes, plus the 404/405 chains.
        ctx.Next()
    })

    admin := app.Router("/admin")
    admin.Use(func(ctx iris.Context) {
        // Child Use: only routes registered under /admin.
        ctx.Next()
    })
    admin.Get("/panel", func(ctx iris.Context) {
        ctx.Text("panel")
    })

    app.Get("/", func(ctx iris.Context) {
        ctx.Text("index")
    })

    app.Listen(":8080")
}
```

When you want root middleware that skips 404s, say so explicitly with `app.UseRelevant(h)`. Note that `UseRelevant` and `UseRelevantOnce` are methods of the application; the `iris.Router` interface that child groups and `Routing(r iris.Router)` controllers receive exposes `Use`, `UseOnce` and `UseRouter`. A child's `Use` is already matched-only.

UseOnce and UseRelevantOnce: upsert

`UseOnce` and `UseRelevantOnce` insert a handler or, if a handler with the same identity is already registered, replace it in place. Identity is the function's resolved name, so re-running setup code does not stack duplicates: the second registration replaces the first. The same rule means two closures produced by the same factory function share a name and therefore replace each other; register factory-produced middleware with plain `Use` when you need several instances.

WrapRouter: below everything

For integrations that need raw `net/http` semantics, `app.WrapRouter` wraps the entire router:

```go
app.WrapRouter(func(w http.ResponseWriter, r *http.Request,
    router http.HandlerFunc) {
    // Runs before the Iris context even exists.
    router(w, r) // call the wrapped router (or don't).
})
```

Wrappers registered with `WrapRouter` execute last-registered-first; `app.AddRouterWrapper` is the first-registered-first variant. Prefer `UseRouter` unless a third-party package hands you an `http.Handler` wrapper you cannot avoid.

Guards and Per-Route Middleware

Handlers listed before the final one in a route registration act as middleware for that route alone:

```go
app.Get("/reports", requireAuth, reportsHandler)
```

In controller-style APIs this becomes the guard idiom: guards are methods on the API struct, so they see the same injected dependencies as the handlers. A *group guard* registered with `r.Use` protects every route of the controller; a *positional guard* between path and handler protects individual routes. From the Bookshelf application of the framework's `TUTORIAL.md`:

```go
// BooksAPI serves /shelves/{shelf_id:uuid}/books.
func (api *BooksAPI) Routing(r iris.Router) {
    // Group guard: every book route needs an existing shelf.
    r.Use(api.requireShelf)

    r.Get("/", api.list)
    // Positional guard: writes additionally need an unlocked shelf.
    r.Post("/", api.requireUnlocked, api.create)
    r.Put("/{id:uuid}", api.requireUnlocked, api.update)
    r.Delete("/{id:uuid}", api.requireUnlocked, api.delete)
}

// requireUnlocked rejects writes against a locked shelf. The shelf was
// loaded and cached by requireShelf, the group guard above.
```

```
func (api *BooksAPI) requireUnlocked(ctx iris.Context) {
    shelf, _ := ctx.ValueAs[library.Shelf](shelfContextKey)
    if shelf.Locked {
        errors.PermissionDenied.Detail(ctx, "shelf is locked",
            "unlock the shelf before modifying its books")
        return // the errors senders stop the request.
    }
    ctx.Next()
}
```

The contract is plain: call `ctx.Next()` to continue, or write a response (the `rest/errors` senders stop the request for you) to end it. The full controller, including the `requireShelf` guard that loads and caches the shelf, is developed in Chapter 4.

Writing Custom Middleware

A timing middleware

Go closures make middleware configuration natural. A closure is a function value that keeps access to the variables of the scope where it was created, so a returned handler can still read the arguments of the function that built it, long after that function returned.

Configurable middleware is a factory function returning a handler; the closure captures the configuration:

GO

```
package main

import (
    "time"

    "github.com/kataras/iris/v14"
)

// SlowRequestLogger warns about requests slower than "threshold".
func SlowRequestLogger(threshold time.Duration) iris.Handler {
    return func(ctx iris.Context) {
        start := time.Now()
        ctx.Next()

        if d := time.Since(start); d > threshold {
            ctx.Application().Logger().Warnf(
                "slow request: %s %s took %s",
                ctx.Method(), ctx.Path(), d)
        }
    }
}

func main() {
    app := iris.New()
    app.UseRouter(SlowRequestLogger(500 * time.Millisecond))
```

```
    app.Get("/", func(ctx iris.Context) {
        ctx.Text("ok")
    })

    app.Listen(":8080")
}
```

One caveat for post-processing: by the time `ctx.Next()` returns, the response body has usually been written, and headers cannot change after the first write. Code after `Next` should log, record metrics or update state. Set response headers *before* calling `Next`.

Request IDs with middleware/requestid

Correlating a response with its server-side logs is the classic middleware job, and Iris ships it as `middleware/requestid`:

GO

```
package main

import (
    "github.com/kataras/iris/v14"
    "github.com/kataras/iris/v14/middleware/requestid"
)

func main() {
    app := iris.New()

    // Assign an ID to every request, even 404s.
    app.UseRouter(requestid.New())

    app.Get("/", func(ctx iris.Context) {
        ctx.Text("request id: " + requestid.Get(ctx))
    })

    app.Listen(":8080")
}
```

The default generator reuses a client-supplied `X-Request-Id` header or generates a UUID, sets the response header and stores the ID on the context (`ctx.ID()` returns it, `requestid.Get(ctx)` as a string).

Accepting the client's ID is convenient for tracing a call through several services, but it also means the value lands in your logs exactly as the client wrote it. Log injection is the abuse of that trust: input crafted with newlines or control characters to forge or corrupt log entries. The access log guards its own lines (its text formats turn a line break in a logged value into a space, and JSON escapes it), but the ID is still the client's choice everywhere else it is written. When clients must not be able to choose or forge IDs, pass the server-side generator:

GO

```
app.UseRouter(requestid.New(requestid.ServerOnlyGenerator))
```

Conditional Middleware

`iris.NewConditionalHandler` combines a filter with one or more handlers into a single registerable handler. The filter is an `iris.Filter`, `func(iris.Context) bool`. When it returns true, the given handlers are spliced into the current request's chain; when it returns false, the chain simply continues without them. The wrapped handlers must behave like middleware and call `ctx.Next()`.

```go
package main

import (
    "github.com/kataras/iris/v14"
)

func auditLog(ctx iris.Context) {
    ctx.Application().Logger().Infof("audit: %s %s",
        ctx.Method(), ctx.Path())
    ctx.Next()
}

func main() {
    app := iris.New()

    // Audit only mutating requests.
    app.Use(iris.NewConditionalHandler(func(ctx iris.Context) bool {
        return ctx.Method() != iris.MethodGet
    }, auditLog))

    app.Get("/items", func(ctx iris.Context) {
        ctx.Text("read")
    })
    app.Post("/items", func(ctx iris.Context) {
        ctx.Text("write")
    })

    app.Listen(":8080")
}
```

For coarser conditions you rarely need it: prefer attaching middleware to the child router or subdomain whose requests it concerns, and let the routing structure do the filtering.

Error-Returning Middleware

Middleware that performs I/O (token validation, database lookups) reads better when it can return an error instead of hand-writing a response. `errors.FuncMiddleware` from `rest/errors` converts a `func(ctx iris.Context) error` into a standard handler: a nil return calls `ctx.Next()`, a

non-nil return flows through the central error pipeline (mapping, logging, JSON rendering) and stops the request.

```go
package main

import (
    "fmt"

    "github.com/kataras/iris/v14"
    "github.com/kataras/iris/v14/rest/errors"
)

// Accounts is the store the guard consults; in production it is a
// database.
type Accounts struct{ active map[string]bool }

func (a *Accounts) Active(ctx iris.Context, id string) (bool, error) {
    return a.active[id], nil
}

var accounts = &Accounts{active: map[string]bool{"acme": true}}

// requireActiveSubscription returns an error instead of writing a
// response; FuncMiddleware below turns it into an ordinary handler.
func requireActiveSubscription(ctx iris.Context) error {
    account := ctx.Params().GetString("account")

    active, err := accounts.Active(ctx, account)
    if err != nil {
        return err // the store is down: the central map makes it a 500.
    }

    if !active {
        return errors.FailedPrecondition.Wrap(
            fmt.Errorf("account %q has no active subscription", account))
    }

    return nil // continue the chain.
}

func main() {
    app := iris.New()

    data := app.Router("/{account:string}",
        errors.FuncMiddleware(requireActiveSubscription))
    data.Get("/data", func(ctx iris.Context) {
        ctx.JSON(iris.Map{"data": "value"})
    })

    app.Listen(":8080")
}
```

The check is a business rule, which is what this shape is for: it does I/O, it can fail two ways, and only your code knows it. Authentication is not in that class. A shared secret presented by a machine caller is `apikey.New` from `middleware/apikey`, covered in Chapter 12, which reads the

header, compares in constant time, renders the 401 and names the caller for you. Reading `X-API-Key` yourself in a middleware like the one above is the version to stop writing.

`errors.FuncHandler` is the terminal-handler counterpart (no `Next` on success), and `errors.NewRouter` wraps a whole router so every route and middleware registration accepts error-returning functions; see Chapter 4 for the wrapper and Chapter 9 for the error map that gives returned errors their status codes.

What the Builder Already Registers

Applications assembled with `iris.NewBuilder` come with a middleware baseline, installed once when the chain reaches `RouterMiddlewares` / `Middlewares`. Do not register these again:

- The options handler for your `Errors(...)` options, via `UseRouter`, so every sender and `HandleError` call uses your map, logger and renderer.
- `errors.Handler`, via `UseRouter` after compression, and as the 404 and 405 handler when the application is built. A bare failure status, an unknown path and (with `FireMethodNotAllowed`) a wrong method render as consistent JSON. To answer 404 or 405 yourself, register `HandleNoRoute` / `HandleNoMethod` from a `Configurator` passed to `NewBuilder`; the builder fills the two slots only when they are empty.
- Panic recovery: `recovery.WithErrorHandler(errors.Internal.LogErr)` followed by `recovery.Handler`, skipped if a handler named `iris.recovery` or `iris.errors.recover` is already present.
- The path-parameter type error handler: a malformed `{id:uuid}` renders 400 `INVALID_ARGUMENT` instead of a bare 404.
- The context error handler for failed writes such as `ctx.JSON` errors.
- A `Server: Iris` response header, plus `X-Developer` when configured.
- `compress.Safe` when `Compression(true)` was selected, skipped if a handler named `iris.compression` exists.
- The rate-limit exceed handler, bound to the JSON error renderer.

Your own additions slot into the same two steps: `RouterMiddlewares(...)` registers through `UseRouter` and `Middlewares(...)` through `Use`, both accepting dependency-injected factories as well as plain handlers; details in Chapter 3.

A panic is Go's runtime failure mode: a nil pointer dereference, an out-of-range index or an explicit `panic()` call abruptly unwinds the goroutine (the lightweight thread) serving the request. Left unrecovered, the client gets a dropped connection with no response and the

failure surfaces only as a stack trace in the server log. Recovery middleware wraps the chain in Go's `recover()` mechanism, so one broken code path yields a logged error and a clean 500 instead.

With `iris.New()` none of this exists, and recovery is the one piece you should always add yourself:

```go
package main

import (
    "github.com/kataras/iris/v14"
    "github.com/kataras/iris/v14/middleware/recovery"
)

func main() {
    app := iris.New()
    app.UseRouter(recovery.Handler)

    app.Get("/", func(ctx iris.Context) {
        panic("boom") // recovered: logged, 500 returned.
    })

    app.Listen(":8080")
}
```

To customize what a recovered panic returns, register a per-request error handler before `recovery.Handler`:

```go
app.UseRouter(recovery.WithErrorHandler(
    func(ctx iris.Context, err error) {
        // Logs the panic and answers the canonical INTERNAL body; the
        // panic value itself never reaches the client.
        errors.Internal.LogErr(ctx, err)
    }), recovery.Handler)
```

Ordering Rules

Three rules govern execution order, all verified by the router's own test suite (`core/router/router_handlers_order_test.go`):

1. Layers run outermost-first. For a matched route:



Router wrappers run first, then the one global UseRouter chain, then the mux, which either branches to the 404 and 405 chains or enters the matched route's frozen chain; code written after ctx.Next() runs on the way back out, innermost first.

2. Within a layer, registration order wins. Two `Use` calls run in the order they were made; handlers listed in one call run left to right. Code placed *after* `ctx.Next()` runs in reverse order on the way out, like deferred calls: the first middleware's post-processing executes last, so a timing middleware registered first measures everything beneath it.

3. A route's chain is assembled at registration time. When `Get`, `Post` or `Handle` runs, the router snapshots the `Use` / `UseRelevant` handlers registered so far and bakes them into the route. Middleware registered afterwards does not reach earlier routes, so register middleware first, routes second. The builder's compiler-enforced step order (`Middlewares` before `API`) exists precisely to make this mistake impossible; with `iris.New()` the discipline is yours.

`UseRouter` and `WrapRouter` are the exception to rule 3: they belong to the router itself rather than to individual routes, and take effect for all requests once the application is built.

Summary

- A middleware is an ordinary `iris.Handler` that calls `ctx.Next()` to continue or stops the chain: quietly by returning, or firmly with `ctx.StopExecution()` and the `StopWith*` shortcuts.
- Scope is chosen by registration method: `UseRouter` runs before routing for every request; root `Use` covers matched routes plus the $^{404}/_{405}$ chains; child `Use` (and `UseRelevant`) covers matched routes only; `UseOnce` / `UseRelevantOnce` insert-or-replace by handler name.
- Positional handlers guard single routes; guard methods on API structs share the controller's injected dependencies.
- `iris.NewConditionalHandler` gates middleware on a per-request filter; `errors.FuncMiddleware` lets middleware return errors into the central error pipeline.
- Builder-assembled applications already include recovery, error rendering, default headers and optional compression. Register only what you add on top. With `iris.New()`, start with `app.UseRouter(recovery.Handler)`.
- Layers run outermost-first, registration order wins within a layer, and route chains are frozen at route registration: middleware first, routes second.

Further Reading

- pkg.go.dev: net/http Handler: the standard library interface `WrapRouter` exposes underneath Iris.
- Go blog: Defer, Panic, and Recover: the language mechanics behind recovery middleware.
- A Tour of Go: Function closures: the feature configurable middleware factories are built on.
- MDN: Cross-Origin Resource Sharing (CORS): why preflight requests exist and which headers answer them.

- OWASP: Logging Cheat Sheet: what to log, what never to log, and injection-safe log handling.

CHAPTER 8

Dependency Injection

Iris ships a small dependency injection container. You register values and constructor functions once, at startup; the container calls your constructors with matching arguments and hands the finished objects to your controllers, handlers and middleware. There is no code generation and no service locator scattered through request code: reflection runs once while the application wires itself, and everything that can go wrong goes wrong before the first request. This chapter explains how registration and resolution work, how controllers and handlers receive their dependencies, and how the same mechanism turns testing into a one-line swap.

Background

What a Dependency Is

Almost no piece of code works alone. A function that saves a user needs something that talks to storage; a handler that sends a welcome message needs a mail client; a service that checks permissions needs the permission rules. Each of those “somethings” is a dependency: a value your code needs to do its job but did not create itself.

The Hard-Wiring Problem

The tempting shortcut is to let each piece build or grab what it needs: a package-level `var db *sql.DB` that everything reads, or a constructor that opens its own database connection. This is hard-wiring, and it costs you twice. First, the code becomes untestable: you cannot exercise `UserService` logic without a real database on the network, because the service insists on connecting to one. Second, the code becomes rigid: swapping storage, pointing staging at different infrastructure, or running two instances with different configuration all mean editing the code instead of changing what you pass in. Global variables add a third cost: any function anywhere may read or mutate them, so a function’s signature no longer tells you what it actually depends on.

Inversion of Control, Without the Jargon

The fix is to turn the direction around: instead of code reaching out for what it needs, the needs are handed in from outside. That is all “inversion of control” means. Its most direct form is constructor injection: a plain function such as `NewUserService(repo UserRepository)` lists its needs as parameters and returns the assembled value. The caller decides what to pass. There are no annotations and no framework requirement, just function parameters doing what they always do.

What a container adds is bookkeeping. In a real application, building the object graph by hand means calling dozens of `New` functions in the right order and threading the results between them; every new parameter ripples through `main`. The Iris container does that threading for you: you list values and constructors once, and it matches each parameter to a previously registered result. The problem that disappears from your code is the hand-maintained wiring, not the constructors themselves.

Interfaces: the Swap Point

An interface in Go is a named list of methods. Any type that has those methods satisfies the interface automatically. There is no `implements` keyword and no registration step. When a constructor asks for an interface instead of a concrete type, it declares “I need something that can `List`” without caring what that something is. That is the swap point this chapter leans on: `main` passes the database-backed implementation, a test passes a ten-line in-memory fake, and the code in between cannot tell the difference.

One more term used below: reflection is Go’s ability to inspect types and call functions at run time, through the standard `reflect` package. The container uses it once, at startup, to read your constructors’ parameter lists and match them to registrations; no reflection runs while requests are being served.

Why Constructor Injection

A constructor that declares its dependencies as parameters is the seam of your application. Whoever calls the constructor decides what “storage” or “mailer” means in that context: `main` passes the production implementation, a test passes a fake, and the type in the middle cannot tell the difference.

```go
// Hidden dependency: the service builds its own storage.
// Tests cannot replace it and every caller pays for the real thing.
func NewUserService() *UserService {
```

```
    return &UserService{repo: connectToPostgres()}
}

// Constructor injection: the caller decides what storage means.
func NewUserService(repo UserRepository) *UserService {
    return &UserService{repo: repo}
}
```

Constructor injection works without any framework. What the Iris container adds is the plumbing: it calls your constructors in the right order with the right arguments, so `main` does not degenerate into a hand-written pyramid of `New` calls that changes every time a signature grows a parameter.

A Complete Wired Application

The recommended v14 stack is `iris.NewBuilder` with `Services` for the dependencies and `NewXxxAPI` constructors for the controllers. The following program is complete; the rest of the chapter takes it apart.

GO

```
package main

import (
    stdContext "context"
    "sync"

    "github.com/kataras/iris/v14"
    "github.com/kataras/iris/v14/rest/errors"
)

// User is the resource served by this chapter's examples.
type User struct {
    ID   string `json:"id"`
    Name string `json:"name"`
}

// UserRepository is the seam between logic and storage. Constructors
// depend on this interface; main and the tests decide what implements it.
type UserRepository interface {
    List(ctx stdContext.Context) ([]User, error)
}

// MemUsers is an in-memory UserRepository: real storage for a small
// tool, the base fake for tests.
type MemUsers struct {
    mu    sync.RWMutex
    users []User
}

var _ UserRepository = (*MemUsers)(nil)

func NewMemUsers() *MemUsers {
    return &MemUsers{
```

```
        users: []User{
            {ID: "1", Name: "Ada"},
            {ID: "2", Name: "Linus"},
        },
    }
}

func (m *MemUsers) List(_ stdContext.Context) ([]User, error) {
    m.mu.RLock()
    defer m.mu.RUnlock()

    list := make([]User, len(m.users))
    copy(list, m.users)
    return list, nil
}

// UserService holds the business rules. It receives its repository
// through the constructor; it never builds one.
type UserService struct {
    repo UserRepository
}

func NewUserService(repo UserRepository) *UserService {
    return &UserService{repo: repo}
}

func (s *UserService) List(ctx stdContext.Context) ([]User, error) {
    return s.repo.List(ctx)
}

// UsersAPI is the HTTP layer: a constructor that declares its
// dependencies and a Routing method that declares its routes.
type UsersAPI struct {
    svc *UserService
}

func NewUsersAPI(svc *UserService) *UsersAPI {
    return &UsersAPI{svc: svc}
}

func (api *UsersAPI) Routing(r iris.Router) {
    r.Get("/", api.list)
}

func (api *UsersAPI) list(ctx iris.Context) {
    users, err := api.svc.List(ctx)
    ctx.List(users, err)
}

// newApp assembles the application. The repository arrives as a
// parameter: main passes the real store, tests pass fakes.
func newApp(repo UserRepository) *iris.Application {
    return iris.NewBuilder().
        NoPrefix().
        AllowOrigin("none").
        Compression(false).
        LogRequests(false).
        NoHealth().
        Errors(errors.NewOptions()).
        Services(repo, NewUserService).
        RouterMiddlewares().
        Middlewares().
```

```
        API("/users", NewUsersAPI).
        Build()
}

func main() {
    app := newApp(NewMemUsers())
    app.Listen(":8080")
}
```

Two lines carry the injection. `Services(repo, NewUserService)` registers the repository value and then calls `NewUserService` with it, because the parameter type matches. `API("/users", NewUsersAPI)` asks the container to call `NewUsersAPI` the same way, with the `*UserService` created a moment earlier. Nothing in `UsersAPI` knows that `MemUsers` exists.

Container Mechanics

Every router owns a `*dep.Container`, reachable through `DI()`. `RegisterDependency` feeds it directly; the builder's `Services` step feeds the same container with extra conveniences (next section). The manual form works on any application, including plain `iris.New()`:

GO

```
app := iris.New()

if err := app.RegisterDependency(
    NewMemUsers,    // constructor: called now, output registered.
    NewUserService, // called with the *MemUsers created above.
); err != nil {
    log.Fatal(err)
}

app.Router("/users", NewUsersAPI)
```

The rules the container follows:

- A plain value (struct pointer, client, config) is registered under its concrete type and returned as-is to whoever asks for that type.
- A function is treated as a constructor and is called immediately, at registration time. Its parameters are resolved from the dependencies registered before it; its outputs become new dependencies. Singletons therefore exist before the first request and a broken graph surfaces at startup, not in production traffic.
- A constructor may include an `error` output. A non-nil error aborts the registration and is returned by `RegisterDependency` (the builder's `Services` panics with it). The error itself is never registered.

- Lookup walks the registrations in reverse order, newest first. When two dependencies can satisfy the same type, the one registered later wins for every consumer registered after it.
- Order matters because construction is eager: register a dependency before the constructors that consume it.

Those two directions are the whole mechanism. The list is built forwards and read backwards:



Dependencies are appended in the order you register them and their constructors run immediately, while resolution walks the same list backwards so the newest registration that satisfies a type wins.

A singleton is a value created once and then shared, as opposed to built fresh for each use. Shared is literal here: the Go HTTP server handles requests concurrently, each on its own goroutine (a lightweight thread managed by the Go runtime), so many requests may call into the same instance at the same moment. “Safe for concurrent use” means the value tolerates that: it is read-only after construction, or it guards its mutable state with a mutex the way `MemUsers` does.

Singletons are the default and only lifetime for ordinary registrations: one instance, shared by every consumer and every request, so anything you register should be safe for concurrent use. The exception is the request-scoped function dependency, covered in its own section.

You rarely touch the container at request time, but it is available for manual work:

```go
container := app.DI()

// Resolve by exact type: pass a value of the registered output type
// (*UserService here) or a reflect.Type.
if v, ok := container.Resolve(new(UserService)); ok {
    svc := v.Interface().(*UserService)
    _ = svc
}

// Invoke a function with injected arguments and take its first result.
result, err := container.InvokeReturningFirst(
    func(svc *UserService) int {
        users, _ := svc.List(stdContext.Background())
        return len(users)
    })
if err != nil {
    log.Fatal(err)
}
count := result.(int)
```

Interface Resolution

`NewUserService` asks for the `UserRepository` interface, but the container holds a `*MemUsers`. Resolution bridges the gap:

1. An exact type match wins immediately.
2. When the requested type is an interface, the container checks each dependency's outputs with `Implements`. Within a dependency the candidates are ordered by method-set size, largest first, so the most specific output is preferred.
3. Across dependencies the reverse-order rule still applies: the newest registration that satisfies the interface wins.

(A method set is simply the collection of methods a type provides. "Largest first" means that when one registration produces several outputs that all satisfy the interface, the output with the most methods is treated as the most specific and wins.)

This rewards the standard Go advice: constructors return concrete types, consumers accept interfaces. Swapping implementations is a registration change, not a refactor:

```go
app.RegisterDependency(NewMemUsers) // development default.
```

```
// Registered later, wins for every consumer registered after it.
// NewPgUsers stands for any alternative implementation.
app.RegisterDependency(NewPgUsers)
app.RegisterDependency(NewUserService) // receives the Postgres store.
```

Because injection happens at registration time, an override placed after the consumers has no effect on them. Put replacements before the constructors that should see them.

Services and the Builder

`Services(deps ...any)` is the builder's registration step. It forwards its arguments to the router's container, in order, left to right, after routing a few shapes specially:

- `func()` and `func() error` become shutdown closers, not dependencies.
- A value with a `Close()` or `Close() error` method is registered as a dependency and as a shutdown closer. When the application shuts down gracefully, its `Close` runs without a separate `Deferrables` entry.
- An `iris.Configurator` is applied to the application instead of being registered.
- `nil` entries are skipped.
- Everything else is registered exactly like `RegisterDependency` would.

GO

```
db, err := sql.Open("postgres", dsn) // *sql.DB has Close() error.
if err != nil {
    log.Fatal(err)
}

app := iris.NewBuilder().
    // ... prefix, CORS, compression, logging, health, errors ...
    Services(
        db,             // dependency, closed automatically on shutdown.
        NewUserRepo,    // constructor: receives the *sql.DB above.
        NewUserService, // constructor: receives the repository.
    ).
    RouterMiddlewares().
    Middlewares().
    API("/users", NewUsersAPI).
    Build()
```

The auto-closer applies to values you pass in, not to what constructors create internally. When a constructor opens a resource, construct it in `main` and pass the value, or register a cleanup through `Deferrables`.

Any registration error makes `Services` panic immediately. That is deliberate; see Fail Fast at Startup.

One more convenience: `iris.NewBuilder` registers the `*iris.Application` itself as a dependency, so a constructor that needs the application (for its logger, for example) can simply accept `app *iris.Application`.

Controllers and Dependencies

A controller is any type with a `Routing(r iris.Router)` method (Chapter 4). Dependencies reach it three ways.

Constructor Resolution

Pass the constructor itself to `API` or `Router` and the container calls it:

```go
// Builder style.
.API("/users", NewUsersAPI)

// Direct style, same resolution.
app.Router("/users", NewUsersAPI)

// Manual style: no container involved, you call the constructor.
app.Router("/users", NewUsersAPI(svc))
```

This is the recommended form. The constructor documents exactly what the controller needs, and an unsatisfiable parameter stops the program at startup with the type and the constructor's file and line in the panic message.

Field Injection

When you pass a pointer to a struct, the router fills its exported, zero-valued fields from the container:

```go
// ReportsAPI relies on field injection instead of a constructor.
type ReportsAPI struct {
    Service *UserService // exported and zero: injected by type.

    // dep:"-" opts a field out of injection; set it yourself.
    Title string `dep:"-"`
}

func (api *ReportsAPI) Routing(r iris.Router) {
    r.Get("/", api.overview)
}

func (api *ReportsAPI) overview(ctx iris.Context) {
    users, err := api.Service.List(ctx)
    ctx.OK(iris.Map{
        "title": api.Title,
        "count": len(users),
```

```
    }, err)
}

// Registration: fields are bound at this moment.
app.Router("/reports", &ReportsAPI{Title: "overview"})
```

Fields that already hold a non-zero value are left untouched, and a field that cannot be resolved panics at startup. Prefer constructors for anything non-trivial; field injection is best for small controllers where a constructor would be ceremony.

Controller Helpers

Two helpers compose with dependency injection inside a `Router` or `API` call:

```go
// Override the inherited dependencies for the controllers that
// follow in the same call (a per-group swap).
app.Router("/preview", iris.Dependencies(previewService), NewUsersAPI)

// Reuse a controller instance already registered as a dependency.
app.Router("/admin", iris.ResolveController[AdminAPI]())
```

`iris.Dependencies` clones the group's container, so the override never leaks to sibling groups. `iris.ResolveController[T]` panics when no `*T` dependency exists.

There is no conditional-registration helper. A feature flag is read where you build the router, and the branch is ordinary Go:

```go
if flags.BillingEnabled {
    app.Router("/billing", iris.ResolveController[BillingAPI]())
} else {
    app.Router("/billing", iris.Middleware(comingSoonHandler))
}
```

Dependency-Injected Handlers

Controllers are not the only consumers. `HandlerFunc` turns a function with dependency parameters into an ordinary `iris.Handler`: the outer function runs once at startup with injected arguments and returns the handler that serves requests.

```go
app.Get("/stats", app.HandlerFunc(func(svc *UserService) iris.Handler {
    return func(ctx iris.Context) {
        users, err := svc.List(ctx)
        ctx.OK(iris.Map{"count": len(users)}, err)
    }
}))
```

The outer function must take only registered dependency types, never the context, and must return an `iris.Handler` (standard `http.Handler`-style returns are converted when possible). The same shape works as middleware:

```go
app.Use(app.HandlerFunc(func(svc *UserService) iris.Handler {
    return func(ctx iris.Context) {
        // Inspect the request using svc, then continue the chain.
        ctx.Next()
    }
}))
```

`HandlerFuncs` maps a list of such functions at once, and the builder's `Middlewares` and `RouterMiddlewares` steps accept them directly, mixed freely with plain handlers. `HandlerFunc` panics when a parameter cannot be resolved, when the function returns nil, or when the result is not a handler.

Request-Scoped Dependencies

Everything so far was a singleton, created once at startup. Some values only exist during a request. The container handles them with a different contract: a registered function whose inputs include a request type is not called at registration. Instead, the function itself becomes the dependency, injectable by its exact function type, and the consumer calls it with the live context when a request is in flight.

The input types that mark a function as request-scoped in v14:

- `iris.Context`
- the standard library's `context.Context`
- `*http.Request`
- `time.Time`

(Router wiring types such as `iris.Router` and controller functions are also reserved internally.) Custom types can be added with `dep.RegisterRequestDependencyType` from `github.com/kataras/iris/v14/core/dep`, before any registration.

```go
// requestID is request-scoped: the container injects this function
// itself, not its result.
func requestID(ctx iris.Context) string {
    return ctx.Header("X-Request-ID")
}

// AuditAPI declares the function type as a dependency.
type AuditAPI struct {
```

```
    requestID func(iris.Context) string
    svc       *UserService
}

func NewAuditAPI(
    rid func(iris.Context) string,
    svc *UserService,
) *AuditAPI {
    return &AuditAPI{requestID: rid, svc: svc}
}

func (api *AuditAPI) Routing(r iris.Router) {
    r.Get("/", api.show)
}

func (api *AuditAPI) show(ctx iris.Context) {
    ctx.OK(iris.Map{"request_id": api.requestID(ctx)}, nil)
}
```

Wire it like any other pair: `Services(requestID)` followed by `API("/audit", NewAuditAPI)`.

Two rules keep this honest. The container never resolves the inputs of a request-scoped function; the caller supplies all of them at call time. And a struct dependency may not contain a request-scoped field: the registration fails with an error telling you to use a function instead. Keeping per-request state on the context itself (`ctx.Values()`, Chapter 5) remains the simpler tool when a middleware already computed the value.

Startup Hooks and Shutdown Closers

Some services need a step after wiring but before traffic: warm a cache, prime a connection pool, load a ruleset. (A connection pool is a set of network connections kept open and reused, so requests skip the cost of dialing the database every time; priming it means opening those connections up front.) Implement `iris.Initializer`:

GO

```
// Initializer is implemented by services registered through Services
// (or RegisterDependency) that need a startup step.
type Initializer interface {
    Init(ctx context.Context) error
}
```

`Build()` runs `Init` exactly once for every singleton that implements it, after dependency injection completes, in registration order. A returned error stops the application from building. Internally the builder walks the container with `DI().VisitSingletons`, which visits each resolved singleton value once (request-scoped functions have no value to visit and are skipped).

A cache that warms itself at startup:

```go
// UserCache serves reads from memory after a one-time warm-up.
type UserCache struct {
    repo UserRepository

    mu    sync.RWMutex
    users []User
}

func NewUserCache(repo UserRepository) *UserCache {
    return &UserCache{repo: repo}
}

// Init implements iris.Initializer. It runs inside Build(), once,
// after injection: the app never starts with a cold cache.
func (c *UserCache) Init(ctx stdContext.Context) error {
    users, err := c.repo.List(ctx)
    if err != nil {
        return err // aborts the build.
    }

    c.mu.Lock()
    c.users = users
    c.mu.Unlock()
    return nil
}

func (c *UserCache) All() []User {
    c.mu.RLock()
    defer c.mu.RUnlock()
    return c.users
}
```

Register it with the rest: `Services(repo, NewUserService, NewUserCache)`. No extra call is needed; implementing the interface is the registration.

`Deferrables` is the mirror image at shutdown. It accepts `func()`, `func() error` and values with a `Close` method, and runs them when the server stops gracefully, together with the closers that `Services` collected from `Close()`-bearing values:

```go
.Deferrables(func() { log.Println("users: goodbye") }).
```

Startup order is registration order; use it when one service must warm up before another. The pair gives every service a full lifecycle: construct (constructor), start (`Init`), serve, stop (`Close` or a deferrable).

Fail Fast at Startup

The dependency system panics instead of limping:

- `Services` panics when a registration fails: a constructor parameter that nothing provides, or a constructor that returned an error.
- `Router` and `API` panic when a controller constructor cannot be satisfied or a field cannot be bound. The message names the missing type (and, for constructors, the function's file and line).
- `HandlerFunc` panics on an unresolvable parameter, a nil result or a non-handler result.
- `Build` panics when an `Init` returns an error.

This is a feature. All of these run during startup, so a broken graph kills the process before it binds a port: the failure is caught by your first local run, your tests or your deploy's health check, never by a user hitting a nil service at 3 a.m. When you want to handle a registration failure yourself, use `app.RegisterDependency`, which returns the error instead of panicking.

Testing with Fakes

A fake is a hand-written stand-in that satisfies the same interface as the real implementation: a slice instead of a database, a struct that always fails instead of a flaky network. Unlike mocks from a mocking library, fakes carry no expectation scripting. They are ordinary Go code, which keeps tests readable and refactor-proof.

The payoff of constructor injection is that `newApp` from this chapter's program accepts the `UserRepository` interface. Tests build the same application as production, minus one argument:

```go
package main

import (
    stdContext "context"
    "fmt"
    "testing"

    "github.com/kataras/iris/v14/httptest"
)

func TestListUsers(t *testing.T) {
    app := newApp(NewMemUsers())
    e := httptest.New(t, app)

    e.GET("/users").Expect().
        Status(httptest.StatusOK).
        JSON().Array().Length().IsEqual(2)
}

// failingUsers embeds the interface and overrides one method;
// everything else is inherited from the wrapped repository.
type failingUsers struct {
    UserRepository
```

```
}

func (failingUsers) List(stdContext.Context) ([]User, error) {
    return nil, fmt.Errorf("pg: connection refused on 10.0.0.3:5432")
}

func TestListUsersFailure(t *testing.T) {
    app := newApp(failingUsers{NewMemUsers()})
    e := httptest.New(t, app)

    e.GET("/users").Expect().
        Status(httptest.StatusInternalServerError)
}
```

`failingUsers` is the whole mocking framework: interface embedding makes a one-method fake five lines, with no library and no expectations DSL. For a fake that only one route group should see, `iris.Dependencies` scopes the override without a second `newApp` variant. The full testing story, including `httptest` options and response assertions, is Chapter 16; the tutorial's Bookshelf project applies this exact `failingRepo` pattern end to end.

Summary

- Constructor injection is the design; the container is the plumbing. Constructors declare dependencies, `Services` (or `RegisterDependency`) provides them, `API` resolves controllers through their `NewXxxAPI` constructors.
- Registration is eager: constructors run at registration time, in order, so list dependencies before their consumers. Lookup is newest first, which makes overriding a registration-order concern.
- Interface parameters are matched by method set, so constructors return concrete types while consumers accept interfaces.
- `Services` routes shapes: closers are collected from `func()`s and `Close()`-bearing values, `Configurator`s are applied, the rest become dependencies. `HandlerFunc` brings injection to plain handlers and middleware.
- Request-scoped dependencies are functions taking `iris.Context`, `context.Context`, `*http.Request` or `time.Time`; the function itself is injected and the consumer calls it with the live context.
- `iris.Initializer.Init` runs once inside `Build()` for cache warm-up and priming; `Deferrables` and auto-registered closers run at shutdown.
- Unresolvable dependencies panic at startup by design: the process refuses to serve with a broken graph.
- Fakes are interface embeddings passed to your `newApp` seam; see Chapter 16.

Further Reading

- Effective Go — Interfaces: the language's own guide to interfaces and implicit satisfaction, the mechanism every swap in this chapter relies on.
- Go Wiki: Code Review Comments — Interfaces: the canonical statement of "accept interfaces, return concrete types" that the container's resolution rules reward.
- The Go Blog: The Laws of Reflection: how reflection works in Go; what the container is doing when it reads your constructors' parameter lists at startup.
- Go standard library: sync: the mutexes and other primitives that make shared singletons safe for concurrent use.

CHAPTER 9

Error Handling

Error handling is where Iris differs most visibly from a bare `net/http` application. Instead of scattering `ctx.StatusCode` and ad-hoc JSON bodies across handlers, the `rest/errors` package gives you canonical error codes, a single wire format, and one central place where Go errors are translated into HTTP responses. Your domain packages return plain sentinel errors; your handlers forward them; the framework classifies, logs, and renders them. This chapter covers the whole system: the wire format, the code catalog, the direct senders, the central error map, field validation, private errors, driver-specific resolvers, error-returning handlers, and the HTML and RFC 7807 renderers.

Background

Errors Are Values, Not Exceptions

Go has no exceptions and no try/catch. A function that can fail returns an `error` (an ordinary value) as its last result, and the caller checks it with `if err != nil`. Nothing unwinds the stack, and an ignored error is simply gone. Two standard-library idioms shape everything in this chapter. A sentinel error is a package-level value created once with `errors.New` and compared by identity: `store.ErrNotFound` is *the* not-found error of that package. Wrapping adds context while keeping the original recognizable: `fmt.Errorf("shelf 42: %w", err)` produces a new error that still contains the sentinel, and `errors.Is` walks the chain of wrapped errors to find it (`errors.As` does the same for typed errors). That is why the central map matches an error no matter how many layers of context were added on the way up.

Go reserves `panic` for a different category: programmer bugs and unrecoverable states, not expected failures. A well-behaved server recovers from panics so one broken request cannot crash the process. Iris does this for you and routes the panic through the same error pipeline as any other failure.

Why APIs Need One Error Format

The consumer of your API is a program, not a person: browser JavaScript, a mobile app, another service. When a request fails, that program has to decide what to do: show a message under the right form field, send the user to the login page, retry in a few seconds, or give up. It can only decide mechanically if every endpoint reports failure the same way: one JSON shape, with a machine-readable code, documented once. Without that contract every client accumulates per-endpoint special cases, and every error you add breaks someone. The wire format in this chapter is that contract, and the problem that disappears from your handlers is deciding, at each call site, what an error response should look like.

Client Mistakes (4xx) versus Server Failures (5xx)

HTTP divides failure status codes into two families, and the split carries real meaning (RFC 9110). A 4xx status says the *request* is wrong: the input failed validation (400), the caller is not signed in (401) or not allowed (403), the resource does not exist (404). The client must change something; repeating the same request will fail the same way. A 5xx status says the *server* failed: a bug, a crashed dependency, an overloaded database. The client did nothing wrong and may retry later. Everything downstream keys off this split (monitoring should page a human for 5xx spikes but not for 404 noise, clients retry a 503 but never a 400), which is why the code catalog below forces every error into one family or the other.

Why Leaking Internals Is a Security Problem

Raw error text is written for the developer who owns the code, and it shows: SQL fragments, table and column names, file paths, internal hostnames, driver versions. Sent to a client, each of those is free reconnaissance for an attacker mapping your system, the class of flaw security guides call information disclosure. The safe pattern, applied throughout this chapter, is asymmetry: the client receives a stable, generic message plus a unique error `id`, while the full error text goes to the server-side logs where that `id` lets you find it. The `Private`, `ErrPrivate` and `LogErr` tools exist to make this asymmetry the path of least resistance.

The Philosophy: One Place for Errors

The design has three rules:

1. **The domain defines sentinel errors.** A service or repository returns `store.ErrNotFound`, `store.ErrShelfLocked`. These are plain `errors.New` values that know nothing about HTTP.

2. **The application maps them once.** A single `errors.NewOptions()` value, handed to the builder's `Errors` step, associates each sentinel with a canonical code. `errors.Is` does the matching, so wrapping with `fmt.Errorf("...: %w", err)` anywhere in the call chain never breaks it.
3. **Handlers just forward.** `ctx.OK`, `errors.HandleError` and friends send any non-nil error through the same pipeline. A handler never decides status codes for domain failures.

```go
// store/store.go
package store

import "errors"

var (
    ErrNotFound    = errors.New("store: not found")
    ErrDuplicate   = errors.New("store: already exists")
    ErrShelfLocked = errors.New("store: shelf locked")
)
```

```go
// main.go (excerpt)
app := iris.NewBuilder().
    Prefix("/api").
    AllowOrigin("none").
    Compression(true).
    LogRequests(true).
    Health(true, "production", "admin").
    Errors(errors.NewOptions().
        MapErrors(errors.NotFound, store.ErrNotFound).
        MapErrors(errors.AlreadyExists, store.ErrDuplicate).
        MapErrors(errors.PermissionDenied, store.ErrShelfLocked)).
    Services( /* repositories, services ... */ ).
    RouterMiddlewares().
    Middlewares().
    API("/books", NewBooksAPI).
    Build()
```

(The chain order is compiler-enforced and every step appears exactly once; `Errors` sits between `Health` and `Services`. See Chapter 3.)

```go
// api/books.go (excerpt)
func (api *BooksAPI) get(ctx iris.Context) {
    book, err := api.svc.Book(ctx, ctx.Params().GetString("id"))
    ctx.OK(book, err)
}
```

If the service returns `store.ErrNotFound` (even wrapped three levels deep), the client receives a `404 NOT_FOUND` wire error. Unmapped errors become a generic `500 INTERNAL`: the message sent to the client is a constant, the real error is logged server-side. Clients never see your SQL.

The Wire Format

Every error response shares one JSON structure, the “HTTP wire error” (`errors.Error`):

```json
{
    "id": "3e9c1b6e-8a3e-4a3c-9d7a-2f2c1a6b5e4d",
    "http_error_code": {
        "canonical_name": "NOT_FOUND",
        "status": 404
    },
    "message": "store: not found",
    "detail": "no book with id '42'",
    "validation": [
        {"field": "title", "value": "", "reason": "must not be empty"}
    ],
    "data": {"hint": "list books at /api/books"}
}
```

- `id`: unique identifier per occurrence, a UUID by default. Use it to correlate a client report with server logs.
- `http_error_code`: the canonical name and its HTTP status.
- `message`: short, human-readable summary.
- `detail` (optional): explanation specific to this occurrence.
- `validation` (optional): array of field-level failures.
- `data` (optional): structured extra data, any JSON value.

(A UUID, or universally unique identifier, is a 128-bit random value with a standard 36-character text form. Collisions are so improbable that one can be generated anywhere, with no coordination, which makes it a good per-occurrence tag.)

The struct also carries optional `type` and `instance` fields for RFC 7807 problem details, covered below.

Canonical Error Codes

An `errors.ErrorCodeName` couples a stable, machine-readable name with an HTTP status code. The most used ones:

Constant	Canonical name	Status

`errors.InvalidArgument`	`INVALID_ARGUMENT`	400
`errors.FailedPrecondition`	`FAILED_PRECONDITION`	400
`errors.OutOfRange`	`OUT_OF_RANGE`	400
`errors.Unauthenticated`	`UNAUTHENTICATED`	401
`errors.PaymentRequired`	`PAYMENT_REQUIRED`	402
`errors.PermissionDenied`	`PERMISSION_DENIED`	403
`errors.NotFound`	`NOT_FOUND`	404
`errors.MethodNotAllowed`	`METHOD_NOT_ALLOWED`	405
`errors.AlreadyExists`	`ALREADY_EXISTS`	409
`errors.Aborted`	`ABORTED`	409
`errors.TooLarge`	`TOO_LARGE`	413
`errors.UnsupportedMediaType`	`UNSUPPORTED_MEDIA_TYPE`	415
`errors.UnprocessableEntity`	`UNPROCESSABLE_ENTITY`	422
`errors.ResourceExhausted`	`RESOURCE_EXHAUSTED`	429
`errors.Internal`	`INTERNAL`	500
`errors.DataLoss`	`DATA_LOSS`	500
`errors.Unknown`	`UNKNOWN`	500
`errors.Unimplemented`	`UNIMPLEMENTED`	501
`errors.BadGateway`	`BAD_GATEWAY`	502
`errors.Unavailable`	`UNAVAILABLE`	503
`errors.DeadlineExceeded`	`DEADLINE_EXCEEDED`	504

The package registers a constant for effectively every HTTP failure status (406–511), plus a few Iris-specific ones such as `errors.InvalidToken` (498) and `errors.TokenRequired` (499). Register your own during initialization when the built-in vocabulary is not enough:

```go
import (
    "net/http"

    "github.com/kataras/iris/v14/rest/errors"
)

// UserBlocked renders as 403 with canonical name USER_BLOCKED.
var UserBlocked = errors.Register("USER_BLOCKED", http.StatusForbidden)
```

Registration is not thread-safe by design; do it in `init` or early in `main`, before the server starts.

Sending Errors Directly

Every code is also a sender. Each method below writes the wire error and stops the request, except `Wrap`, which only tags an error for later processing:

Method	Signature (after the code)	Purpose
`Message`	`(ctx, message string)`	Message only
`Messagef`	`(ctx, format string, args ...any)`	Formatted message
`Detail`	`(ctx, msg, detail string)`	Message + detail
`Detailf`	`(ctx, msg, detail string, args ...any)`	Formatted detail
`Data`	`(ctx, msg string, data any)`	Message + JSON data
`DataWithDetail`	`(ctx, msg, detail string, data any)`	All three
`Validation`	`(ctx, errs ...ValidationError)`	Field failures
`Err`	`(ctx, err error)`	`err.Error()` as the message
`ErrPrivate`	`(ctx, err error)`	Generic message, error kept server-side
`LogErr`	`(ctx, err error)`	Logs; `Internal` sends a generic message
`Log`	`(ctx, format string, args ...any)`	Like `LogErr` with formatting
`Problem`	`(ctx, title, detail, typeURI, instanceURI string, extra ...any)`	RFC 7807
`Wrap`	`(err error) error`	Tags an error, sends nothing

```go
import (
    "github.com/kataras/iris/v14"
    "github.com/kataras/iris/v14/rest/errors"
)

func handler(ctx iris.Context) {
    // Message and detail.
    errors.NotFound.Detailf(ctx, "book not found",
        "no book with id %q", ctx.Params().GetString("id"))

    // Structured data alongside the message.
    errors.FailedPrecondition.Data(ctx, "insufficient funds", iris.Map{
        "current_balance":  50.00,
```

```
        "required_balance": 100.00,
    })
}
```

One sentence gets written often enough to have its own function. `errors.NotFoundID(ctx, resource, id)` renders `NOT_FOUND` with the detail `"<resource> <id> not found"`, which is the `Detailf` above with its format string spelled out for the hundredth time:

```go
func getPlan(ctx iris.Context) {
    plan, err := svc.GetPlan(ctx, id)
    if errors.HandleError(ctx, err) {
        return
    }

    if plan == nil {
        errors.NotFoundID(ctx, "plan", id)
        return
    }

    ctx.OK(plan, nil)
}
```

The id is formatted with `%v`, so a string, an integer and a `uuid.UUID` all read the way they print. Its `bool` result follows the `Context.OK` convention, `true` meaning the handler succeeded, so it is always `false` here; note that `HandleError` in the same package reports the opposite, `true` when it wrote the response, which is why the example tests one and ignores the other.

Err, LogErr and ErrPrivate

The three error-value senders differ in what the client sees and what is logged:

```go
// Client sees err.Error() as the message under the given code.
// Validation errors are auto-routed to Validation().
errors.InvalidArgument.Err(ctx, err)

// Logs the error. For errors.Internal the client gets a generic
// message; for any other code the error text is sent.
errors.Internal.LogErr(ctx, err)

// The client always gets the generic message
// ("server error" / "an unexpected error occurred while
// processing the request"); the real error stays server-side.
errors.Internal.ErrPrivate(ctx, err)
```

Use `Err` for errors that are safe to show, `LogErr` as the default for server faults, and `ErrPrivate` when the error text may leak schema, paths or credentials. `Log` and `LogErr` also skip canceled requests: when the client has disconnected, nothing is logged or sent (package-level setting `errors.SkipCanceled`, enabled by default).

`Err` has one more convenience: if the error value is a struct, its exported fields are marshaled into the response's `data` member automatically, so a typed domain error carries its context to the client without extra code.

Wrap: tag now, send later

`Wrap` joins the code with an error so a lower layer can decide the status without touching the context:

```go
func (s *Service) Publish(ctx context.Context, id string) error {
    n, err := s.repo.Publish(ctx, id)
    if err != nil {
        return err
    }
    if n == 0 {
        return errors.NotFound.Wrap(fmt.Errorf("book %q", id))
    }
    return nil
}
```

When this error later reaches `errors.HandleError`, the tag is detected and a `404 NOT_FOUND` with message `book "42"` is sent. Prefer the central map for recurring cases; reach for `Wrap` when the classification depends on local information the map cannot see.

The Central Error Map

`errors.NewOptions()` returns an `*errors.Options` whose error-code map is **not** empty. The defaults (built in `rest/errors/options.go`) map:

- `database/sql.ErrNoRows` → `NotFound`
- the `github.com/kataras/jwt` token errors (`jwt.ErrExpired`, `jwt.ErrMissing`, `jwt.ErrTokenSignature`, `jwt.ErrBlocked` and the rest of the family) → `Unauthenticated`
- `view.ErrNoEngine` (from `middleware/view`) → `Internal`
- hex-decode failures (`encoding/hex: invalid byte`) → `InvalidArgument`, re-worded as "invalid secret format"

PostgreSQL mappings are deliberately not included; they are opt-in through the `pgerrors` subpackage.

Every `Map...` call extends the map; nothing is replaced:

```go
import (
    "fmt"
```

```
    "github.com/kataras/iris/v14"
    "github.com/kataras/iris/v14/rest/errors"

    "myapp/store"
)

opts := errors.NewOptions().
    // Sentinel values, matched with errors.Is (wrapping-safe).
    MapErrors(errors.NotFound, store.ErrNotFound).
    MapErrors(errors.AlreadyExists, store.ErrDuplicate).

    // Whole types, matched with errors.As semantics.
    MapErrorTypes(errors.InvalidArgument, store.QuotaError{}).

    // A function: return nil to pass, or the error to send.
    MapErrorFunc(errors.Internal, func(err error) error {
        if store.IsRetryable(err) {
            return errors.Private(err) // hide the cause from the client
        }
        return nil
    }).

    // Context-aware variant: enrich with request detail.
    MapErrorFuncCtx(errors.NotFound,
        func(ctx iris.Context, err error) error {
            if errors.Is(err, store.ErrNotFound) {
                return fmt.Errorf("%s: %w", ctx.Path(), err)
            }
            return nil
        })
```

- `MapErrors(code, targets ...error)`: exact sentinels via `errors.Is`.
- `MapErrorTypes(code, targets ...any)`: concrete or interface types; pass a typed nil pointer (`(*MyError)(nil)`) for interfaces.
- `MapErrorFunc(code, func(error) error)`: transform or claim errors; functions for the same code run in registration order until one returns non-nil. The two closure bodies people write most often, `errors.As` on a concrete type and a substring test on the message, have their own calls; see the next section before reaching for this one.
- `MapErrorFuncCtx(code, func(iris.Context, error) error)`: the same, with access to the request context. The context is nil when the map's `Filter(ctx, err)` method is called directly outside a request, so tolerate nil if you call it yourself.

Candidates are evaluated in ascending status-code order, so more specific client-error classifications win over catch-all 500 mappings.

Mapping a typed error, and mapping a text

Two shapes of `MapErrorFunc` closure are common enough to have their own call. The first finds a concrete error type and builds the response from its fields; without a helper it is a closure declaring a variable, calling `errors.As`, and returning `nil` on the way out.

```go
opts.MapErrorAs(errors.ResourceExhausted,
    func(e *store.QuotaError) *errors.Error {
        return &errors.Error{
            Message: "quota exceeded",
            Detail:  fmt.Sprintf("the limit is %d", e.Limit),
        }
    })
```

`errors.As` looks for the type anywhere in the error tree, so a wrapped one matches, and the `render` function receives the value it found. A nil result means "not mine" and the next mapping gets a turn, exactly like `MapErrorFunc` returning nil. Passing a nil `render` classifies without transforming, which needs the type spelled out because there is nothing left to infer it from: `opts.MapErrorAs[*store.QuotaError](code, nil)`.

`MapErrorAs` is a generic method on both `*errors.Options` and a bare `errors.ErrorCodeMap`, and it returns the receiver, so it chains after `MapErrors` like every other mapping call. The type parameter is inferred from the `render` function. When you need the same lookup inside your own code, `errors.AsType[*store.QuotaError](err)` is the standard library's `errors.AsType`, forwarded by `rest/errors` next to `Is` and `As`, so a file that imports the package as `errors` needs no second import.

The second shape sniffs a substring, which is all a driver leaves you when its error arrives as text:

```go
opts.MapErrorContains(errors.InvalidArgument,
    "violates check constraint", "invalid value")
```

The match is case-sensitive and the message replaces the original text, so the driver's own words never reach the client. Two caveats come with it. Prefer a typed mapping wherever one exists, because a driver is free to reword its text in the next release and a substring mapping fails silently when it does. And because the replacement discards the original, log before you map when those words are the only diagnostic you have, the way `pgerrors` does with `errors.LogError`. An empty `substr` registers nothing at all rather than a mapping that matches every error: a missing constant is a mistake, and the safe reading of it is "map nothing".

Wiring the options

With the builder, pass the options to the `Errors` step (its position in the chain is compiler-enforced):

```go
iris.NewBuilder().
    // ... Prefix, AllowOrigin, Compression, LogRequests, Health ...
    Errors(opts).
```

```
    // ... Services, RouterMiddlewares, Middlewares, API, Build ...
```

With a manual `iris.New()` application, register the options' middleware yourself, before the routes it should cover:

```go
app := iris.New()
app.UseRouter(opts.Handler())
```

`opts.Handler()` stores the options on each request's context; every sender and `HandleError` call then finds the map, logger, collector and renderer you configured. `UseRouter` makes it cover $^{404}/_{405}$ paths too; plain `app.Use(opts.Handler())` covers matched routes only (see Chapter 4 for the middleware families).

Forwarding Errors from Handlers

`errors.HandleError(ctx, err) bool`, from `rest/errors`, is the front door of the pipeline. It returns false for a nil error, otherwise it classifies the error, writes the response and returns true:

```go
import (
    "github.com/kataras/iris/v14"
    "github.com/kataras/iris/v14/rest/errors"
)

func (api *BooksAPI) create(ctx iris.Context) {
    input, ok := ctx.BindJSON[CreateBookInput]()
    if !ok {
        return // error response already written
    }

    book, err := api.svc.Create(ctx, input)
    if errors.HandleError(ctx, err) {
        return
    }

    ctx.Created(book, nil)
}
```

`ctx.Created(book, err)` would fold those three middle lines into itself, which is why handlers built on the response helpers rarely name `HandleError`. Call it directly for an error that arrives outside a helper's reach: from a guard, from a middleware, from a stream already in flight.

`ctx.HandleError(err)` is the same thing as a context method, for a step that has an error but nothing to render on success:

```go
func (api *PlansAPI) deactivate(ctx iris.Context) {
```

```
    if ctx.HandleError(api.svc.DeactivatePlans(ctx, id)) {
        return
    }

    ctx.NoContent(true, nil)
}
```

It is the error half that `OK`, `Created`, `NoContent`, `List`, `Count` and `Paginated` each run before rendering their success body, on its own. A handler whose only use of `rest/errors` was that one call no longer imports the package.

When the failure of one step means something specific at that call site and nothing in general, name the code instead of mapping it. `errors.<Code>.Handle(ctx, err)` renders the error as that code and reports whether it did:

```go
func (api *AuthAPI) changeEmail(ctx iris.Context) {
    input, ok := ctx.BindJSON[ChangeEmailInput]()
    if !ok {
        return
    }

    if ctx.HandleError(api.identity.ChangeEmail(ctx, input)) {
        return
    }

    // A third party being down is Unavailable here, whatever the
    // central map would have made of it.
    if errors.Unavailable.Handle(ctx, api.users.UpdateEmail(ctx, input.Old, input.New)) {
        return
    }

    ctx.NoContent(true, nil)
}
```

`Handle` is `Err` plus the answer to “did it fire”, and a nil error writes nothing and reports false. `Err` keeps the signature it has, because `func(*iris.Context, error)` is what `LogErrorFunc` and `cors.HandleErrorFunc` accept and `cors.New().HandleErrorFunc(errors.FailedPrecondition.Err)` passes it straight to them.

The classification order inside `HandleError` is a waterfall: the first test that recognizes the error writes the response, so a specific rule always beats a general one.



errors.HandleError walks ten tests in order and the first match writes the response, so a more specific rule always beats a more general one and an unrecognised error ends at a logged 500.

The family has three siblings:

- `errors.HandleRequestError(ctx, err)`: identical, except that it treats a failure to decode the request body as the client's fault. It differs at step 6 only: a JSON decoding error renders `400 INVALID_ARGUMENT` ("unable to parse request body") rather than a 500. An error that no step recognizes still ends at `500 INTERNAL` either way. Use it when processing user input, where a decode failure is almost certainly the client's.
- `errors.HandleErrorf(ctx, err, format, args...)`: wraps the error with call-site detail (using `%w`, so mappings still apply), then handles it:

```go
// Renders the mapped 404 with message "shelf 42: store: not found".
if err := api.svc.Rename(ctx, id, input.Title); err != nil {
    errors.HandleErrorf(ctx, err, "shelf %s", id)
    return
}
```

- `errors.HandleRequestErrorf`: the same enrichment with the 400 fallback.

The response helpers all call `HandleError` internally: `ctx.OK`, `ctx.Created`, `ctx.NoContent`, `ctx.Count` and `ctx.Paginated` (Chapter 6). That is why a typical handler body is one line: `ctx.OK(result, err)`.

Field Validation

`errors.ValidationError` is a small interface (`error` plus `GetReason() string`), and `errors.ValidationErrors` is a slice of them. The `rest/errors/validation` subpackage ships fluent, generic builders:

Constructor	Chainable rules
`validation.String(name, value)`	`NotEmpty()`, `Length(min, max)`, `Email()`, `Fullname()`, `Password(level)`, `Phone(kind)`
`validation.Number(name, value)`	`Positive()`, `Negative()`, `Zero()`, `NonZero()`, `InRange(min, max)`
`validation.Slice(name, value)`	`NotEmpty()`, `Length(min, max)`
`validation.Field(name, value)`	`Func(fns ...func(T) string)`, custom rules

Rules accumulate reasons; `validation.Join` filters out the fields that passed and returns nil when everything is valid. The idiomatic v14 place for this is a `Validate` method on the request type; `ctx.BindJSON[T]()` calls it automatically (Chapter 5):

```go
import (
    "github.com/kataras/iris/v14/rest/errors/validation"
)

type CreateBookInput struct {
    Title string `json:"title"`
    Pages int    `json:"pages"`
    Email string `json:"email"`
}

func (in *CreateBookInput) Validate() error {
    return validation.Join(
        validation.String("title", in.Title).NotEmpty().Length(1, 200),
        validation.Number("pages", in.Pages).Positive(),
```

```
        validation.String("email", in.Email).Email(),
    )
}
```

A failed validation renders with a fixed message and detail and one entry per failing field. This is the real wire format:

```JSON
{
    "http_error_code": {
        "canonical_name": "INVALID_ARGUMENT",
        "status": 400
    },
    "message": "validation failure",
    "detail": "fields were invalid",
    "validation": [
        {"field": "title", "value": "",
         "reason": "must not be empty, must be between 1 and 200 characters"},
        {"field": "pages", "value": 0, "reason": "must be positive"},
        {"field": "email", "value": "bad",
         "reason": "must be a valid email address"}
    ]
}
```

Outside `Validate` methods, send field errors explicitly, either through `Err` (validation errors are auto-routed) or `Validation` directly:

```GO
name := validation.String("name", input.Name).NotEmpty().Length(3, 50)
email := validation.String("email", input.Email).Email()

if err := validation.Join(name, email); err != nil {
    errors.InvalidArgument.Err(ctx, err)
    return
}
```

Custom rules are plain functions from the value to a reason string (empty means valid):

```GO
username := validation.Field("username", input.Username).Func(
    validation.NotEmpty,
    func(value string) string {
        if strings.Contains(value, " ") {
            return "must not contain spaces"
        }
        return ""
    },
)

if err := username.Get(); err != nil { // Get() returns nil when valid
    errors.InvalidArgument.Err(ctx, err)
    return
}
```

Any type of your own that implements `Error() string` and `GetReason() string` participates in the same machinery, useful when adapting a third-party validator's output.

Private Errors

Two complementary tools keep sensitive error text away from clients:

- `errors.Private(err)` marks an error at its *creation site*. Wherever it later surfaces in the pipeline, the client receives the generic message while the original text is preserved for server-side logging. Use it in service layers that cannot know how the error will be rendered, or inside `MapErrorFunc` transformations.
- `code.ErrPrivate(ctx, err)` decides at the *response site*: generic message now, real error kept server-side. That holds for every error type: a `*errors.Error` or a validation error gets the generic message here too.

```go
// Service layer: mark once, forget about it.
return errors.Private(
    fmt.Errorf("query failed on host %s: %w", host, err))

// Handler: forwards as usual; the marker is detected automatically.
errors.HandleError(ctx, err)
```

The scrubbed response is always the same shape:

```json
{
    "http_error_code": {"canonical_name": "INTERNAL", "status": 500},
    "message": "server error",
    "detail": "an unexpected error occurred while processing the request"
}
```

That is the shape `HandleError` and the senders produce. `errors.Handler` answers a private stored error by the status it carries, and it does so before any mapping is asked. Below 500 the client reads the status text and no detail: a private error stored with `ctx.StopWithError(401, iris.PrivateError(err))` reads `"message":"Unauthorized"`, not a server fault. At 500 and above it reads the generic "server error". The status the handler set stays, even when your map would have turned the error into a 404. The `jwt` middleware stops every bad token this way, so its 401 reads `Unauthorized` with or without your error options, and a client cannot learn from the body why its token was refused. The private text still reaches the collector.

Note the asymmetry worth memorizing: `errors.Internal.LogErr(ctx, err)` is generic to the client, but `errors.Internal.Err(ctx, err)` sends the error text. When in doubt on a 500 path, use `LogErr`.

Database Errors and Resolvers

`sql.ErrNoRows` is mapped out of the box. PostgreSQL mappings for the `github.com/kataras/pg` driver are opt-in, so the driver stays out of your dependency graph unless you use it:

```go
import _ "github.com/kataras/iris/v14/rest/errors/pgerrors"
```

(A blank import, the `_` alias, compiles a package purely for its side effects: its `init` functions run, registering the resolvers below, and no identifier enters your namespace.)

The blank import restores four behaviors inside `HandleError`:

- `pg.ErrNoRows` → `404 NOT_FOUND` ("resource not found")
- duplicate-key violations → `409 ALREADY_EXISTS` with a generic client detail; the raw driver error (which leaks table, column and constraint names) is logged server-side only
- foreign-key violations (SQLSTATE `23503`) → `400 INVALID_ARGUMENT` ("invalid reference"), naming the constraint when the driver reports one so the client can tell which reference was wrong. A row that points at something which is not there is bad input, not a server fault, and without this branch it fell through to a 500. The driver's own message and detail, which name the table, the column and the offending value, are logged and never forwarded
- SQLSTATE-tagged input errors ("invalid input syntax", "operator does not exist") → `400 INVALID_ARGUMENT` with the SQLSTATE noise trimmed

(SQLSTATE is the SQL standard's five-character error code, attached to every PostgreSQL error: `23505` is a unique-key violation, `22P02` an invalid text representation. The resolver reads it to tell client-caused input errors apart from genuine server faults.)

The subpackage is built on `errors.RegisterErrorResolver`, a public extension point you can use for any driver or SDK. Resolvers run inside `HandleError` after the per-route options map and before the built-in classification; the first one that reports true stops the chain:

```go
import (
    "github.com/kataras/iris/v14"
    "github.com/kataras/iris/v14/rest/errors"
)

func init() {
```

```
    errors.RegisterErrorResolver(
        func(ctx iris.Context, err error) bool {
            var terr *mydriver.TimeoutError
            if !errors.As(err, &terr) {
                return false // not ours, keep going
            }

            errors.Unavailable.Message(ctx,
                "storage temporarily unavailable")
            return true
        })
}
```

Register resolvers at init time or before serving, like error codes.

Handlers That Return Errors

Writing `if err != nil { errors.HandleError(ctx, err); return }` in every handler gets repetitive. The `errors.Router` wrapper accepts handlers of type `errors.Func` (`func(ctx iris.Context) error`) and pipes any returned error through `HandleError` automatically. A controller opts in by taking `*errors.Router` in its `Routing` method; the framework's controller resolver recognizes it (plain routing functions `func(r *errors.Router)` work too):

GO

```
import (
    "github.com/kataras/iris/v14"
    "github.com/kataras/iris/v14/rest/errors"

    "myapp/reports"
)

type ReportsAPI struct {
    svc *reports.Service
}

func NewReportsAPI(svc *reports.Service) *ReportsAPI {
    return &ReportsAPI{svc: svc}
}

func (api *ReportsAPI) Routing(r *errors.Router) {
    r.Get("/{year:int}", api.byYear)
}

func (api *ReportsAPI) byYear(ctx iris.Context) error {
    report, err := api.svc.ByYear(ctx, ctx.Params().GetInt("year"))
    if err != nil {
        return err // classified by the central map
    }

    return ctx.JSON(report)
}
```

Register it like any controller: `API("/reports", NewReportsAPI)` in the builder, or `app.Router("/reports", NewReportsAPI(svc))` manually. The wrapper exposes the full routing surface (`Get`, `Post`, `Put`, `Delete`, `Patch`, `Any`, `Subdomain`, `Use`, `UseRouter`) with error-returning signatures. Middleware follows the same rule: a nil return continues the chain, a non-nil error is handled and stops it:

```go
r.Use(func(ctx iris.Context) error {
    if api.svc.Rebuilding(ctx) {
        return errors.Unavailable.Wrap(
            errors.New("reports are being rebuilt"))
    }
    return nil
})
```

Write these for the rules only your code knows. Authentication is not one of them: a machine caller presenting a shared secret is `apikey.New` from `middleware/apikey` (Chapter 12), which renders the 401 through this same pipeline.

For a single handler on a plain router, `errors.FuncHandler(fn)` converts an `errors.Func` to a standard handler, and `errors.FuncMiddleware(fn)` does the same with `ctx.Next()` semantics.

HTML and RFC 7807 Rendering

JSON is the default rendering, with two built-in alternatives.

HTML error pages

When the request or the already-set response `Content-Type` is `text/html`, wire errors render as HTML pages automatically. The `errors.HTML` renderer looks for a view template named after the status code (`errors/404.html`, `errors/500.html`) using the view engine registered through `middleware/view` (Chapter 6); if no engine or template exists, a minimal inline page with the status, canonical name and message is served. To force HTML for every error regardless of content type:

```go
opts := errors.NewOptions().SetRenderer(errors.HTML)
```

Inside the template, the wire error is the root value: `{{.Message}}`, `{{.Detail}}`, `{{.ErrorCode.Status}}`, `{{.ID}}`.

RFC 7807 problem details

RFC 7807, “Problem Details for HTTP APIs”, is the IETF’s standard answer to the format question this chapter opened with: a small, extensible JSON vocabulary (`type`, `title`, `status`, `detail`, `instance`) served under its own media type so generic clients and tooling can recognize an error response without knowing your API. Reach for it when you must interoperate with consumers that expect the standard shape.

Each code has a `Problem` sender that produces an RFC 7807 “Problem Details” response with `Content-Type: application/problem+json`:

```go
errors.InvalidArgument.Problem(ctx,
    "Validation Failed",                              // title
    "The email field contains an invalid address",    // detail
    "https://example.com/problems/validation-failed", // type URI
    "/users/create")                                  // instance URI
```

```json
{
    "id": "3e9c1b6e-8a3e-4a3c-9d7a-2f2c1a6b5e4d",
    "status": 400,
    "title": "Validation Failed",
    "type": "https://example.com/problems/validation-failed",
    "detail": "The email field contains an invalid address",
    "instance": "/users/create"
}
```

Trailing `extra ...any` values become extension members: a single object-shaped value is merged into the problem at the top level, anything else (or multiple values) renders under a `data` key, nils are filtered. The problem format is used whenever `type`, `instance` and a title are present and no custom renderer is registered; to render *every* error as problem+json, set the package’s renderer:

```go
opts := errors.NewOptions().SetRenderer(errors.Problem)
```

`SetRenderer` accepts any `func(ctx iris.Context, err *errors.Error)`, so XML, content negotiation or a bespoke envelope are equally possible.

Logging, Collection and Error IDs

The `Options` value carries three more hooks, all chainable:

```go
import (
    "log/slog"
    "uuid"
```

```
    "github.com/kataras/iris/v14"
    "github.com/kataras/iris/v14/rest/errors"
)

opts := errors.NewOptions().
    // Called for logged errors (LogErr, Log, private errors,
    // unmapped 500s). Default: the application logger.
    SetLogger(func(ctx iris.Context, err error) {
        slog.Error("http error",
            "error", err.Error(),
            "path", ctx.Path(),
            "method", ctx.Method(),
        )
    }).
    // Called with the complete wire error before it is sent
    // (and before private scrubbing), synchronously — offload
    // slow work to a goroutine.
    SetCollector(func(ctx iris.Context, err *errors.Error) {
        slog.LogAttrs(ctx, slog.LevelWarn,
            "wire error", err.Attrs()...)
    }).
    // Error ID generation. Default: a UUID per error.
    SetIDGenerator(func(ctx iris.Context) string {
        if id := ctx.Header("X-Request-Id"); id != "" {
            return id
        }
        return uuid.New().String()
    })
```

- `SetLogger(func(ctx iris.Context, err error))` overrides logging for routes covered by these options; the package-level `errors.LogError` variable is the global fallback.
- `SetCollector` feeds monitoring: the collector observes the full error (including details that private scrubbing removes from the client copy). `(*errors.Error).Attrs()` returns ready-made `slog` attributes if you log wire errors structurally.
- `SetIDGenerator` / `SetID` control the `id` field. Static IDs via `SetID` are for tests; production should keep unique IDs so a client screenshot can be matched to a log line.

Reporting without rendering

Every method so far ends in a render, which is fine while the response is still unwritten. A streamed response is not: after its first byte the status code is spent, and writing an error body into the middle of an event stream corrupts it. Skipping the error system instead is worse, because the logger and the collector are the only things that carry a failure anywhere, and a stream that fails silently is a stream nobody knows failed.

`errors.Report(ctx, err)` runs the whole pipeline and returns the `*Error` without writing it:

```
reported := errors.Report(ctx, err) // nil when there is nothing to report.
```

GO

It resolves the error through the route's error-code map, honours `SkipCanceled`, calls the logger with the real error, generates the id, redacts a private or internal error down to a generic message, and calls the collector. What comes back is the client-safe copy, and it is the caller's to put wherever the transport allows. `errors.ErrorCodeName.Report(ctx, err)` is the same thing under a code you already know, for a failure that has been classified elsewhere.

This is what the `sse` package writes its error frame with, so a streaming handler gets the same treatment as every other one without asking for it. Chapter 6 covers the frame, and Chapter 21 covers what a relayed upstream failure looks like once it has been through here.

Router-Level Defaults

Builder-built applications get the error system's defaults wired without any code:

- **Bare failure statuses**: a status of 400 or above that reaches the client with nothing written answers the JSON envelope, with the status text as the message. It does not matter whether the handler stopped the chain: `ctx.StopWithStatus(403)`, `ctx.NotFound()`, the 401 of the `jwt` and `basicauth` middleware and a file the file server does not have all answer it, under the canonical name and status text of their own status (`PERMISSION_DENIED` and "Forbidden" for a 403). The 413 of the `bodylimit` middleware is the one whose message is not the status text: `TOO_LARGE` with "request body too large".
- **Unknown paths and wrong methods**: an unknown path answers `404 NOT_FOUND`. A method the path does not serve answers `405 METHOD_NOT_ALLOWED`, with the `Allow` header intact, when `Options.Router.FireMethodNotAllowed` is on; it is off by default, and the answer is then the 404 one.
- **Typed path parameters**: a request that fails a macro type check (`/users/{id:uuid}` receiving `abc`) renders `400 INVALID_ARGUMENT` with message "invalid path parameter", the parse failure as detail and the offending parameter as data.
- **Panics**: the recovery middleware routes panics through `errors.Internal.LogErr`: a logged, generic 500 wire error.
- **Rate limiting**: when the `rate` middleware trips, the client gets `429 RESOURCE_EXHAUSTED` with "you may try again in 2s" as detail.
- **Response write failures**: context-level errors are forwarded to `HandleError`.

The first two come from `errors.Handler`, which the builder registers as the last of its built-in router middleware, after compression, and as the 404 and the 405 handler. Your own `app.HandleNoRoute` or `app.HandleNoMethod`, registered from a `Configurator` given to `NewBuilder`

or on the value `iris.Default()` returns, still wins: the builder fills those two slots only when they are empty at `Build`.

What the client reads for an error stored on the context follows the rules of the rest of this chapter, with one rule ahead of them: a private error is never rendered. Private means stored through `iris.PrivateError` or `ctx.SetErrPrivate`, or carrying the mark anywhere in its chain, which is what `errors.Private` builds and what a private error wrapped again with `%w` still is. Such an error is answered before any mapping is asked, with the status the handler set: below 500 the status text is the message, and at 500 and above the client reads the generic "server error" while the error goes to the logger. Any other stored error that your map or a resolver recognises renders through its mapping; a malformed JSON body stored next to a 400, for example, answers `400 INVALID_ARGUMENT` with "unable to parse request body". Otherwise, below 500 the error's text is the message. At 500 and above a stored `*errors.Error` is sent as it is, because that type exists to be sent to clients, and anything else goes to the logger while the client reads "server error". A database that is down never has its address read by a client because somebody forgot to mark the error. `ctx.StopWithPlainError` and `ctx.SetErr` store an error and write nothing, so the rules above decide the body. `ctx.StopWithError` with an error that is not private writes the error's text itself, so that response has a body and `errors.Handler` leaves it alone. With a private error, whether `iris.PrivateError(err)`, `errors.Private(err)` or a rewrap of either, it sets the status, writes nothing and stores the error as private, so the private rule answers it.

A response that already has a body is left alone, and so is one that declares a `Content-Length`, one whose status line is already on the wire, or one whose connection was hijacked. The declared length matters for standard handlers mounted with `iris.FromStd`: one that sets `Content-Length: 0` and calls `w.WriteHeader(401)`, which is what `httputil.ReverseProxy` relays for an upstream's empty 401, keeps its empty answer, while one that sets a failure status and neither writes, flushes nor declares a length gets the envelope.

Two failures are answered ahead of `errors.Handler` and keep their bare status: a rejection from a handler that a `Configurator` registered with `UseRouter`, which runs before the builder's own middleware, and the 415 the compression middleware sends for a request body that is not what its `Content-Encoding` says.

Manual applications can adopt the same defaults. Where you register `errors.Handler` decides what it covers, and `UseRouter` alone is not enough for 404 and 405: it runs before the router, and by the time the chain comes back the router's default handler has already written `Not Found`. Register it as the no-route and no-method handler as well:

```go
func init() { errors.ApplyPatches() }
```

```
app := iris.New()
app.UseRouter(opts.Handler())      // your central map
app.UseRouter(errors.Handler)      // bare failure statuses
app.HandleNoRoute(errors.Handler)  // 404 as a wire error
app.HandleNoMethod(errors.Handler) // 405 (FireMethodNotAllowed)
```

`app.Use(errors.Handler)` on the root application covers the routes and the router's 404 and 405 in one line, but then a `HandleNoRoute` or `HandleNoMethod` registered after it never runs, because in those two chains `errors.Handler` answers and does not call the next handler. For a custom non-JSON 404 page on an `iris.New()` application, skip `errors.Handler` there and register your own handlers with `app.HandleNoRoute` / `app.HandleNoMethod` (Chapter 4). On a builder application, register them from a `Configurator` passed to `iris.NewBuilder`, as above.

Summary

- Domain packages return sentinel errors; one `errors.NewOptions()` map, wired through the builder's `Errors` step (or `app.UseRouter(opts.Handler())`), translates them; handlers forward with `ctx.OK` / `errors.HandleError`.
- Every error renders as one wire format: `id`, `http_error_code` (canonical name + status), `message`, optional `detail`, `validation` and `data`.
- Codes are senders: `Message`, `Detail`, `Data`, `Err`, `LogErr`, `ErrPrivate`, `Validation`, `Problem`, and `Wrap` for deferred handling. On 500 paths prefer `LogErr` or `ErrPrivate`; never leak internals.
- `MapErrors` matches sentinels, `MapErrorTypes` matches types, `MapErrorFunc` / `MapErrorFuncCtx` transform; the default map already covers `sql.ErrNoRows`, JWT errors and more. `pgerrors` adds PostgreSQL via a blank import; `RegisterErrorResolver` is the extension point.
- Validation builders (`validation.String/Number/Slice/Field` + `validation.Join`) produce the standard `validation failure` response; `ctx.BindJSON[T]()` runs your `Validate` method automatically.
- `errors.Router` lets handlers and middleware return errors instead of writing responses; `errors.HTML` and `errors.Problem` provide HTML pages and RFC 7807 output; `SetLogger`, `SetCollector` and `SetIDGenerator` hook logging, monitoring and correlation IDs.

Further Reading

- The Go Blog: Error handling and Go: the foundational piece on errors as values and the `error` interface.

- The Go Blog: Working with Errors in Go 1.13. Wrapping with `%w`, `errors.Is` and `errors.As`: the exact mechanics the central map relies on.
- RFC 9110 — HTTP Semantics: Status Codes: the authoritative definition of every 4xx and 5xx status.
- MDN: HTTP response status codes: the same catalog in reference-card form, with usage notes.
- RFC 7807 — Problem Details for HTTP APIs: the problem+json format behind the `Problem` sender (superseded by RFC 9457, which keeps the same wire format).
- OWASP Error Handling Cheat Sheet: why clients get generic messages and logs get the details.

CHAPTER 10

Authentication & Authorization

Iris ships several layers of authentication support, from a single-line HTTP Basic Auth middleware to a complete OAuth2 identity server. This chapter helps you pick the right layer, then walks through each one: `middleware/basicauth` for internal surfaces, the generic `auth` SDK for token-based user-facing APIs, `middleware/jwt` for raw token signing and verification, `middleware/sessions` for browser applications, and a tour of the `auth/identity` and `auth/oauth2` packages. Authentication answers *who* the caller is; authorization answers *what* the caller may do. The chapter closes the loop with plain guard handlers for roles and permissions.

Background

Two Questions, Two Answers

Authentication and authorization are separate problems that are easy to blur. Authentication establishes identity: is this request really from Maria? Authorization decides rights: Maria is real, but may she delete this record? HTTP encodes the difference in two status codes you will meet throughout the chapter: 401 means “I do not know who you are”, 403 means “I know who you are, and the answer is no”.

Credentials, Tokens, Sessions and Cookies

A credential is the proof of identity a caller presents: a username and password, an API key, a client certificate. The complication is that HTTP is stateless (the server forgets you the moment a response is sent), so *every* request must carry proof. Sending the password itself each time (which is exactly what Basic Authentication does) multiplies the chances of it leaking, so most systems exchange the password once for something weaker and shorter-lived.

That something takes two shapes. A token is a string the server issues after one successful credential check; the client attaches it to later requests instead of the password. Most tokens are *bearer* tokens: whoever holds one is treated as the user, no further questions, which is why

tokens expire quickly and why revocation matters. A session is the server-side variant: the server keeps the signed-in state in its own storage, and the client holds only a random session identifier that points at it. Tokens push state to the client; sessions keep it on the server.

A cookie is the browser-side transport for either. It is a small piece of data a server asks the browser to store, which the browser then sends back automatically with every request to that site. Three attributes recur in this chapter: `HttpOnly` (page scripts cannot read the cookie, which blunts token theft via injected JavaScript), `Secure` (sent over HTTPS only) and `SameSite` (limits sending on cross-site requests, the raw material of request forgery).

How a JWT Works

A JSON Web Token (JWT, RFC 7519) is a token with structure: three base64url-encoded parts (header, payload, signature) joined by dots. The payload carries *claims*: statements such as the user's ID, roles and an expiry time. The signature is computed over the other two parts with a key, so a verifier can prove the token was issued by the key holder and has not been altered, without a database lookup. That self-contained verification is the JWT's whole appeal for APIs. The catch to internalize: a JWT is signed, not encrypted. Anyone who obtains one can decode and read every claim, so the payload must never contain secrets, which is why this chapter insists the user type embedded in tokens stays small and clean. And because a stateless token cannot be recalled once issued, access tokens are kept short-lived and paired with refresh tokens or server-side revocation checks.

Password Hashing

Storing passwords as-is turns any database leak into a catastrophe: every account is compromised at once, and (since people reuse passwords) so are their accounts elsewhere. The remedy is hashing: a one-way function transforms the password into a fixed string that cannot be reversed; you store only the hash and re-run the function at sign-in to compare. General-purpose hashes like SHA-256 are too *fast*: attackers test billions of guesses per second offline. Password hashing therefore uses deliberately slow, salted algorithms such as bcrypt, the one used in this chapter's examples. The salt is a random per-user value mixed in so identical passwords produce different hashes and precomputed lookup tables are useless.

Choosing an Approach

You are building	Reach for
An internal or admin surface with a handful of operators	`middleware/basicauth`

Another service, a cron job or a webhook sender, with no user to sign in	`middleware/apikey`
A user-facing API with access and refresh tokens	the `auth` package (`auth.Auth[T]`)
A resource server that only verifies tokens issued elsewhere	`middleware/jwt`
A browser application with server-side state	`middleware/sessions`
A standalone identity provider for many applications	`auth/identity` + `auth/oauth2`

The layers compose. A typical production application protects its admin routes with basic auth, its public API with the auth SDK, and adds small guard handlers for role checks, all registered per router, so protection lives where the routes are mounted.

Basic Authentication

HTTP Basic Authentication sends a username and password with every request. It requires no token infrastructure, which makes it the right tool for internal dashboards and admin APIs served over HTTPS, and the wrong tool for anything user-facing.

(On the wire it is an `Authorization: Basic` header holding `username:password` base64-encoded. Base64 is an encoding, not encryption: anyone who can observe the traffic can read the credentials, which is why HTTPS is non-negotiable here.)

The middleware is a plain `iris.Handler`, so it applies wherever you mount it, a specific child router, not the whole application:

```go
package main

import (
    "github.com/kataras/iris/v14"
    "github.com/kataras/iris/v14/middleware/basicauth"
)

func main() {
    app := iris.New()

    admin := app.Router("/admin", basicauth.Default(map[string]string{
        "admin": "admin_password",
        "ops":   "ops_password",
    }))

    admin.Get("/stats", func(ctx iris.Context) {
        username, _ := ctx.User().GetUsername()
        ctx.Writef("hello, %s", username)
    })

    app.Listen(":8080")
}
```

On success the middleware stores the authenticated user on the request; `ctx.User()` retrieves it. Unauthenticated requests receive a 401 challenge automatically.

In builder-based applications the same handler slots into the `API` step, exactly as Chapter 3 shows:

```go
API("/admin", basicauth.Default(map[string]string{
    cfg.AdminUser: cfg.AdminPassword,
}), api.NewAdminAPI).
```

Hashed Passwords from a File

Never commit plain-text passwords. `basicauth.Load` reads a JSON or YAML user list and, with the `BCRYPT` option, compares the user input against bcrypt hashes:

```yaml
- username: admin
  password: "$2a$10$..." # bcrypt hash
  role: admin
- username: ops
  password: "$2a$10$..."
  role: operator
```

```go
auth := basicauth.Load("users.yml", basicauth.BCRYPT)
```

`Load` keeps each entry as a `basicauth.Map`, so the `role` above survives. To decode the same file into your own struct instead, pass `basicauth.AllowUsersFile[Operator]("users.yml", basicauth.BCRYPT)` as the `Allow` of `basicauth.New`, described next.

(The `$2a$10$` prefix of a bcrypt hash records the algorithm variant and its cost factor, here 2^10 internal rounds. Raising the cost makes every guess proportionally slower for an attacker, at the price of a slower comparison at sign-in.)

Custom Validation and Rate Limiting

`basicauth.New` accepts an `Options[U]` struct, typed by your own user, when you need database-backed users, credential expiration or a failure limit:

```go
import (
    "time"

    "github.com/kataras/iris/v14"
    "github.com/kataras/iris/v14/middleware/basicauth"

    "golang.org/x/crypto/bcrypt" // third-party
)
```

```
func newAdminAuth(repo *OperatorRepository) iris.Handler {
    return basicauth.New(basicauth.Options[*Operator]{
        Realm:    basicauth.DefaultRealm,
        MaxAge:   2 * time.Hour, // re-ask for credentials after 2h.
        MaxTries: 3,             // 403 after 3 failures.
        GC:       basicauth.GC{Every: 3 * time.Hour},
        Allow: func(ctx iris.Context, username,
            password string) (*Operator, bool) {
            op, err := repo.GetByUsername(ctx, username)
            if err != nil {
                return nil, false
            }

            err = bcrypt.CompareHashAndPassword(
                []byte(op.PasswordHash), []byte(password))
            if err != nil {
                return nil, false
            }

            return op, true
        },
    })
}
```

Whatever `Allow` returns is what the handlers get back, typed:

```
admin.Get("/me", func(ctx iris.Context) {
    op, _ := basicauth.GetUser[*Operator](ctx) // no type assertion
    since, _ := basicauth.AuthorizedAt(ctx)     // first login, needs MaxAge
    ctx.JSON(iris.Map{"username": op.Username, "since": since})
})
```

GO

`ctx.User()` still returns the same value as a `context.User`, and `ctx.Logout()` forgets the login. For a static list of your own type, `basicauth.AllowUsers(operators)` takes a `[]Operator` and reads the credentials from the `Username` and `Password` fields (json tags count), or from wherever a `basicauth.Credentials` option points.

API Keys for Machine Callers

Basic auth and the SDK both assume a person: a password to remember, a session to end, a token to refresh. A cron job, a webhook sender, a reporting service or a CLI tool has none of those. It has one long random string it was handed once and presents on every request, and what the server owes it is a constant-time comparison and a name in the log.

`middleware/apikey` is that. The key arrives on a header, by default `Authorization: Bearer <key>`, and the middleware matches it against the configured keys:

```go
package main

import (
    "os"

    "github.com/kataras/iris/v14"
    "github.com/kataras/iris/v14/middleware/apikey"
)

func main() {
    app := iris.New()

    guard := apikey.New(apikey.Options{
        Keys: map[string]string{
            "reporting": os.Getenv("REPORTING_API_KEY"),
            "billing":   os.Getenv("BILLING_API_KEY"),
        },
    })

    api := app.Router("/api", guard)
    api.Get("/orders", func(ctx iris.Context) {
        principal, _ := apikey.Get(ctx) // "reporting" or "billing".
        ctx.JSON(iris.Map{"called_by": principal})
    })

    app.Listen(":8080")
}
```

`Keys` maps the principal to its secret and not the other way round, so two callers cannot share an entry: one key per consumer is what makes a leak traceable and a revocation survivable.

The default is to deny. A middleware with no `Keys` and no `Lookup` rejects every request and logs a warning on the first one, rather than letting everyone through while the deployment is missing its configuration. `AllowAnonymous` opens that door for local development and is ignored the moment a key exists.

The scheme is matched case-insensitively, per RFC 7235. A caller that cannot set `Authorization` gets its own header instead, and then the value is bare:

```go
apikey.New(apikey.Options{
    Header: "X-API-Key",
    Scheme: apikey.NoScheme, // the "-" sentinel: accept a bare value.
    Keys:   keys,
})
```

`NoScheme` is a sentinel rather than the empty string on purpose: an unset `Scheme` field keeps the `Bearer` default, because a field nobody filled in must not turn a check off.

Two options keep the working credential out of the configuration. `Hash` is applied to the presented key before comparison, so `Keys` hold digests:

```go
apikey.New(apikey.Options{
    Hash: apikey.SHA256,
    Keys: map[string]string{
        "reporting": "9f86d081884c7d659a2feaa0c55ad015a3bf4f1b2b0b822cd15d6c15b0f00a08",
    },
})
```

A shared secret is a high-entropy random string and not a password, so a plain digest is the right hash here: there is nothing to brute-force. `Lookup` covers the keys that live in a table, and is consulted only after the static keys miss:

```go
apikey.New(apikey.Options{
    Lookup: func(ctx iris.Context, key string) (string, bool, error) {
        row, err := tokens.FindByHash(ctx, apikey.SHA256(key))
        if stderrors.Is(err, sql.ErrNoRows) {
            return "", false, nil // not a match: a 401.
        }
        if err != nil {
            return "", false, err // the store is down: a 500.
        }

        return row.Name, !row.Revoked, nil
    },
})
```

The distinction in that last branch matters: an error is a 500 and not a 401, because a database that is down must not read to the caller as an invalid key.

`Options.Query` reads the key from a query parameter when the header is absent, for a caller that cannot set headers at all. Turn it on only after reading what it costs: a key in a URL ends up in browser history, proxy logs and referrer headers, and, when `middleware/accesslog` is registered, in your own access log, which records every query argument. Its `Redact` hook covers request and response bodies only, so the parameter has to be dropped or masked in a custom formatter (`ac.SetFormatter`) before you enable it.

On success the principal is on the context. `apikey.Get(ctx)` returns it, and when nothing else has set a user the middleware also stores a `context.SimpleUser` whose `ID` and `Username` are that principal, so code written against `ctx.User()` sees the caller without knowing where the credential came from. An existing user is never replaced.

The rejection reasons are `apikey.ErrMissing` (no key, or a value the scheme does not fit) and `apikey.ErrInvalid` (a key that matched nothing), wrapped into the error an `ErrorHandler` receives, so the two can be logged differently. The default renders the `rest/errors` `Unauthenticated` envelope.

The Auth SDK

The `auth` package is the recommended way to secure a user-facing API. Its central type, `auth.Auth[T]`, signs and verifies JSON Web Tokens for your own user type `T`. It ships ready-made handlers for the whole token lifecycle (`SigninHandler`, `RefreshHandler`, `SignoutHandler`, `SignoutAllHandler`) plus a `VerifyHandler` middleware, EdDSA (Ed25519) signatures, encrypted refresh tokens with rotation, and optional secure cookie storage for browser clients.

You supply two things: a configuration (the keys) and a `Provider` (the user store).

A note on the key material before the YAML below: EdDSA is an asymmetric signature algorithm (Ed25519 key pairs). Asymmetric means the two keys split the roles (the private key signs tokens, the public key can only verify them), so other services can check your tokens without ever holding a key that could forge one. The `/.well-known/jwks.json` endpoint you will meet shortly publishes exactly that public key, in the standard JSON Web Key Set format. Symmetric algorithms such as HS256 use one shared secret for both jobs, which is simpler but means every verifier can also mint tokens.

Configuration and Keys

The configuration declares which headers carry the token, an optional cookie, and the signing keys. It usually loads from a YAML file:

```yaml
Headers:
  - "Authorization"
  - "X-Authorization"
Cookie: # optional; enables encrypted cookie storage for browsers.
  Name: "iris_auth_cookie"
  Secure: true
  Hash: "..."  # 64 characters, HMAC authentication.
  Block: "..." # 32 characters, AES-256 encryption.
Keys:
  - ID: IRIS_AUTH_ACCESS # required: short-lived access tokens.
    Alg: EdDSA
    MaxAge: 2h
    Private: |+
      -----BEGIN PRIVATE KEY-----
      ...
      -----END PRIVATE KEY-----
    Public: |+
      -----BEGIN PUBLIC KEY-----
      ...
      -----END PUBLIC KEY-----
  - ID: IRIS_AUTH_REFRESH # optional: enables the refresh flow.
    Alg: EdDSA
    MaxAge: 720h # 30 days; then the user must sign in again.
```

```
    EncryptionKey: "..." # refresh tokens travel encrypted.
    Private: |+ # own key pair, same PEM shape as above.
      ...
    Public: |+
      ...
```

The `IRIS_AUTH_ACCESS` key is mandatory; adding `IRIS_AUTH_REFRESH` enables refresh tokens. If the file does not exist, the load error prints a freshly generated example configuration you can start from. Loading options:

```go
// One call: bind the file and construct the instance (panics on error).
sso := auth.MustLoad[Customer]("auth.yml")

// Or bind manually, e.g. when auth config is part of your app config
// (see Chapter 11):
var cfg auth.Configuration
if err := cfg.BindFile("auth.yml"); err != nil {
    log.Fatal(err)
}
sso := auth.Must[Customer](cfg)

// Development and tests only: random, non-persistent keys.
var devCfg auth.Configuration
_ = devCfg.BindRandom()
```

The User Type and a Database-Backed Provider

`T` is any struct: it becomes the JWT payload, so keep it small and free of secrets. A `Provider[T]` connects the SDK to your user store:

```go
type Provider[T User] interface {
    Signin(ctx context.Context, username, password string) (T, error)
    Refresh(ctx context.Context, t T) (T, error)
    ValidateToken(ctx context.Context, claims StandardClaims, t T) error
    InvalidateToken(ctx context.Context, claims StandardClaims, t T) error
    InvalidateTokens(ctx context.Context, t T) error
}
```

`Signin` runs on sign-in, `Refresh` before a new token pair is issued, `ValidateToken` on every verified request, and the two `Invalidate` methods implement server-side sign-out. A complete provider against a SQL database, with a single `revocations` table for both “sign out this token” and “sign out everywhere”:

```go
package main

import (
    "context"
    "database/sql"
    "fmt"
    "time"
```

```
    "github.com/kataras/iris/v14/auth"

    "golang.org/x/crypto/bcrypt" // third-party
)

// Customer is embedded in the token payload: small, no secrets.
type Customer struct {
    ID     string   `json:"id"`
    Email  string   `json:"email"`
    Roles  []string `json:"roles"`
    Active bool     `json:"active"`
}

// CustomerProvider implements auth.Provider[Customer].
//
// Schema used below:
//
//    customers(id, email, active, password_hash)
//    revocations(token_id, customer_id, revoked_at)
type CustomerProvider struct {
    DB *sql.DB
}

func (p *CustomerProvider) Signin(ctx context.Context,
    username, password string) (Customer, error) {
    var (
        c    Customer
        hash string
    )

    err := p.DB.QueryRowContext(ctx,
        `SELECT id, email, active, password_hash
         FROM customers WHERE email = $1`, username).
        Scan(&c.ID, &c.Email, &c.Active, &hash)
    if err != nil {
        return Customer{}, fmt.Errorf("invalid credentials")
    }

    err = bcrypt.CompareHashAndPassword([]byte(hash), []byte(password))
    if err != nil {
        return Customer{}, fmt.Errorf("invalid credentials")
    }

    return c, nil
}

// Refresh reloads the customer before a new token pair is signed,
// so profile or role changes propagate on the next refresh.
func (p *CustomerProvider) Refresh(ctx context.Context,
    c Customer) (Customer, error) {
    err := p.DB.QueryRowContext(ctx,
        `SELECT id, email, active FROM customers WHERE id = $1`,
        c.ID).Scan(&c.ID, &c.Email, &c.Active)
    return c, err
}

// ValidateToken runs on every verified request. A token is rejected if
// its ID (or, for refresh tokens, the access-token ID it originates
// from) was revoked, or if all of the customer's tokens issued before a
// "sign out everywhere" event are revoked.
func (p *CustomerProvider) ValidateToken(ctx context.Context,
    claims auth.StandardClaims, c Customer) error {
```

```
    var revoked bool
    err := p.DB.QueryRowContext(ctx,
        `SELECT EXISTS (SELECT 1 FROM revocations
         WHERE token_id IN ($1, $2)
            OR (customer_id = $3 AND revoked_at >= $4))`,
        claims.ID, claims.OriginID, c.ID, claims.IssuedAt).
        Scan(&revoked)
    if err != nil {
        return err
    }
    if revoked {
        return fmt.Errorf("token revoked")
    }
    return nil
}

func (p *CustomerProvider) InvalidateToken(ctx context.Context,
    claims auth.StandardClaims, c Customer) error {
    _, err := p.DB.ExecContext(ctx,
        `INSERT INTO revocations (token_id, revoked_at)
         VALUES ($1, $2)`, claims.ID, time.Now().Unix())
    return err
}

func (p *CustomerProvider) InvalidateTokens(ctx context.Context,
    c Customer) error {
    _, err := p.DB.ExecContext(ctx,
        `INSERT INTO revocations (customer_id, revoked_at)
         VALUES ($1, $2)`, c.ID, time.Now().Unix())
    return err
}
```

For fully stateless deployments, `ValidateToken` and the `Invalidate` methods may simply return nil; tokens then remain valid until they expire. The context passed to every method is the request's `iris.Context` when invoked through the built-in handlers, so a provider can inspect the request if it needs to.

Registering as a Service

Construct the instance once, attach the provider, and register it as a service so any controller can receive it:

```
package main

import (
    "database/sql"
    "log"

    "github.com/kataras/iris/v14"
    "github.com/kataras/iris/v14/auth"
)

func main() {
    db, err := sql.Open("pgx", "postgres://localhost/shop")
    if err != nil {
        log.Fatal(err)
```

```
    }

    sso := auth.MustLoad[Customer]("auth.yml")
    sso.AddProvider(&CustomerProvider{DB: db})

    app := iris.NewBuilder(iris.Defaults()).
        Prefix("/api").
        AllowOrigin("https://shop.example.com").
        Compression(true).
        LogRequests(true).
        NoHealth().
        Deferrables(db).
        Services(db, sso).
        Middlewares().
        API("/account", NewAccountAPI).
        Build()

    // Optional, at the root: lets other services verify your tokens.
    app.Get("/.well-known/jwks.json", sso.JWKSHandler())

    app.Listen(":8080")
}
```

Verifying Requests per Router

The controller receives the `*auth.Auth[Customer]` through its constructor and applies `VerifyHandler` exactly where protection is needed: sign-in and refresh stay public, everything under `/me` requires a valid token:

GO

```
package main

import (
    "github.com/kataras/iris/v14"
    "github.com/kataras/iris/v14/auth"
)

type AccountAPI struct {
    sso *auth.Auth[Customer]
}

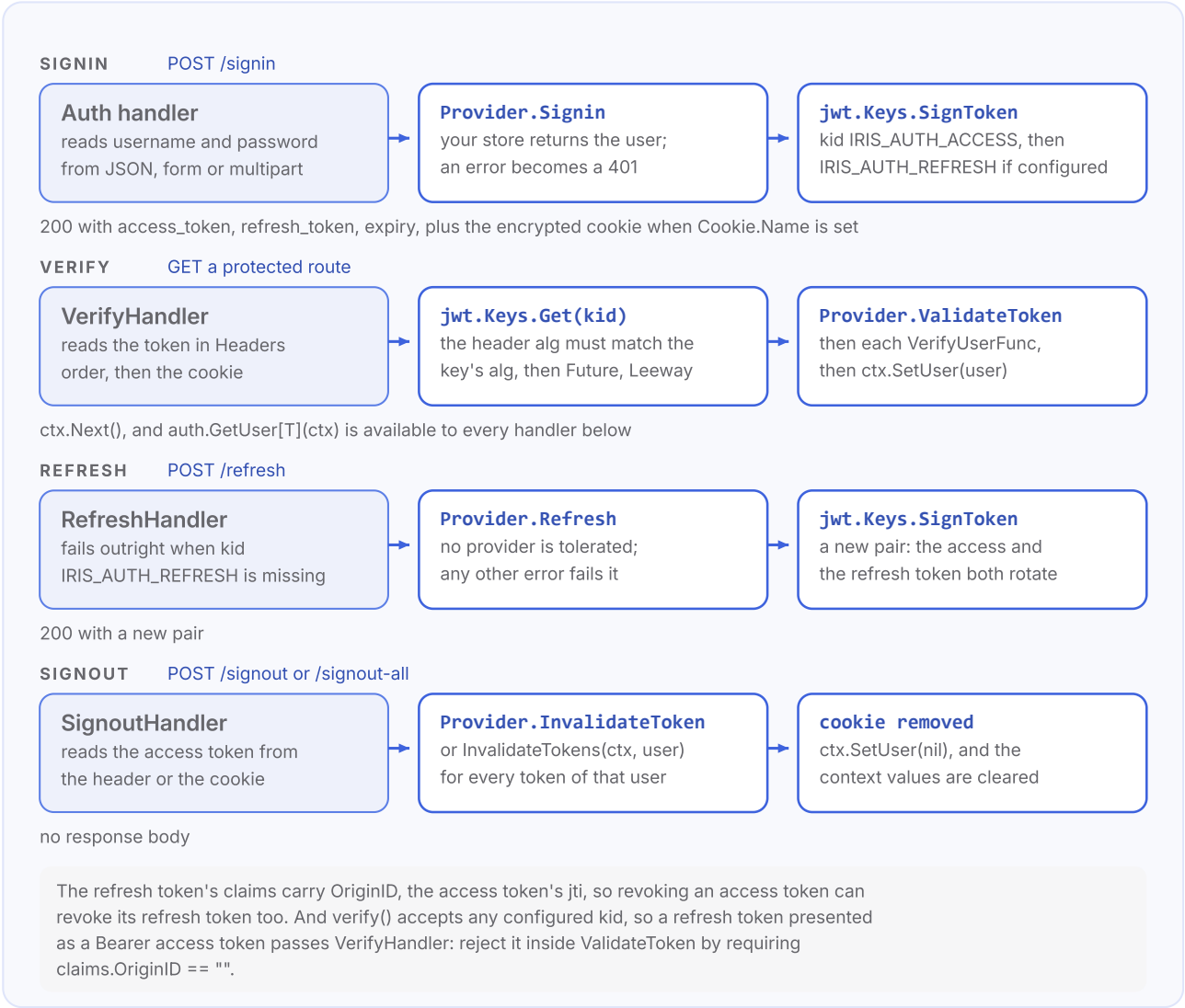
func NewAccountAPI(sso *auth.Auth[Customer]) *AccountAPI {
    return &AccountAPI{sso: sso}
}

func (api *AccountAPI) Routing(r iris.Router) {
    r.Post("/signin", api.sso.SigninHandler)
    r.Post("/refresh", api.sso.RefreshHandler)

    me := r.Router("/me", api.sso.VerifyHandler())
    me.Get("/", api.profile)
    me.Post("/signout", api.sso.SignoutHandler)
    me.Post("/signout-all", api.sso.SignoutAllHandler)
}

func (api *AccountAPI) profile(ctx iris.Context) {
    ctx.OK(auth.GetUser[Customer](ctx), nil)
}
```

The flow, end to end, is four acts. In each one the SDK's handler is the only thing your routes see; it drives your `Provider` and the key registry behind it:



Each Auth handler drives your Provider and the kid key registry: signin mints an access and refresh pair linked by OriginID, verify checks the access token and loads the user, refresh rotates both, and signout invalidates them.

One consequence of that design is worth guarding against. `verify` accepts any token signed by a *configured* key, and the refresh key is configured, so a client that presents its refresh token as `Authorization: Bearer <refresh_token>` passes `VerifyHandler`. The refresh token's claims carry `OriginID` (the access token's `jti`) and an access token's do not, so the check belongs in your `ValidateToken`: require `claims.OriginID == ""` and reject anything else.

Claims Checks with VerifyUserFunc

`VerifyHandler` accepts optional functions that further validate the already-verified user. Use them for cheap, declarative checks that apply to a whole router:

```go
import stdContext "context"

active := r.Router("/orders", api.sso.VerifyHandler(
    func(ctx stdContext.Context, c Customer) error {
        if !c.Active {
            return fmt.Errorf("account disabled")
        }
        return nil
    }))
```

A failing check produces the same 401 response as an invalid token, through the instance's `ErrorHandler` (replaceable via `SetErrorHandler`).

Reading the User in Handlers

After `VerifyHandler`, three package-level helpers are available:

```go
// The verified user value.
user := auth.GetUser[Customer](ctx)

// The verified token: raw bytes plus standard claims.
token, ok := auth.GetToken(ctx)

// True when a verified user is stored on this request.
if auth.UserAuthorized(ctx) { /* ... */ }
```

The Request Scope

In a multi-tenant API almost every query needs one value off the verified user: the tenant, the customer, the organisation the request belongs to. A handler that derives it again is a second chance to get it wrong, and a handler that forgets to filter by it is a data leak. `auth.Scope` derives it once, right after `VerifyHandler`, and every handler after that reads it back:

```go
app.Use(a.VerifyHandler(), auth.Scope(func(u Customer) string {
    return u.TenantID
}))

func listInvoices(ctx iris.Context) {
    tenantID, ok := auth.ScopeID[string](ctx)
    if !ok {
        return
    }

    invoices, err := svc.InvoicesOf(ctx, tenantID)
    ctx.List(invoices, err)
}
```

`ScopeID` is a function of the `auth` package rather than a context method, because the scope is stored by `auth`. It reports `false` when `Scope` did not run or when it stored an id of another type, so a zero id you must not trust cannot be mistaken for a real one.

Two rulings are worth knowing before you rely on it. A missing user is a 401 with `auth.ErrMissingUser`, the same failure `VerifyHandler` answers with, and so is a user of a *different* type: reading the user untyped and deriving a scope from a zero value would answer 200 with an empty scope, which is a cross-tenant read wearing the disguise of the next sentence. A derived *zero* id, on the other hand, is stored as it is, because a global administrator may legitimately have no scope; guard those requests by checking the id in the handler, not by expecting `Scope` to refuse them.

The cost is one `Values` read, the `derive` call and one `Values` set per request; `ScopeID` is one map lookup and one type assertion. No reflection is involved on either side. The stamping half, writing the scope onto every body a client sends, is `mvc.BeforeWrite` in Chapter 24.

Cookie Storage for Browser Clients

The cookie the SDK issues is both encrypted and authenticated: the `Block` key encrypts the content (AES), and the `Hash` key signs it (HMAC), so the browser stores the token but can neither read nor undetectably alter it.

When `Cookie.Name` is set (with `Hash` and `Block`, both required), `SigninHandler` also stores the access token in an encrypted, HttpOnly, SameSite=Lax cookie, and `VerifyHandler` falls back to that cookie when no `Authorization` header is present. The same `Auth[T]` instance then serves API clients and browser pages alike. A sign-in request may carry a `?redirect_to=/path` query; for safety, only same-site relative paths are followed; absolute or scheme-bearing targets are ignored and the token is returned as JSON instead.

The JWT Middleware

`middleware/jwt` is the thin layer beneath the auth SDK: a `Signer` and a `Verifier` without providers, refresh flows or user stores. Reach for it when you only need to mint or verify tokens: for example, a resource server that trusts tokens issued by another service.

```go
package main

import (
    "time"

    "github.com/kataras/iris/v14"
    "github.com/kataras/iris/v14/middleware/jwt"
    "github.com/kataras/iris/v14/rest/errors"
)

type userClaims struct {
    Username string `json:"username"`
}
```

```
func main() {
    secret := []byte("your-256-bit-secret-keep-it-safe")

    signer := jwt.NewSigner(jwt.HS256, secret, 15*time.Minute)
    verifier := jwt.NewVerifier(jwt.HS256, secret)

    app := iris.New()

    app.Post("/token", func(ctx iris.Context) {
        // Authenticate the caller first, then:
        token, err := signer.Sign(userClaims{Username: "makis"})
        if errors.HandleError(ctx, err) {
            return // logged, and a canonical INTERNAL body sent.
        }
        ctx.Write(token)
    })

    api := app.Router("/api",
        verifier.Verify(func() any { return new(userClaims) }))
    api.Get("/profile", func(ctx iris.Context) {
        claims, _ := jwt.Get[*userClaims](ctx)
        ctx.JSON(claims)
    })

    app.Listen(":8080")
}
```

`verifier.Verify` returns a middleware that extracts, verifies and decodes the token, then exposes the claims through `jwt.Get[T](ctx)` and the full token through `jwt.GetVerifiedToken(ctx)`. `T` is what the constructor passed to `Verify` returns, here `*userClaims`; `Get` reports false instead of panicking when the request carries no claims or claims of another type. Invalid or missing tokens receive a 401 via the verifier's replaceable `ErrorHandler`. The default one stores the reason as a private error, so under `errors.Handler` the body reads `"message":"Unauthorized"` whatever your error options map: a client cannot read from it why its token was refused, and the reason stays in your logs and collector.

Notes for v14:

- By default the verifier reads the `Authorization: Bearer <token>` header **only**. Tokens in the URL query string leak into access logs, proxy logs and browser history, so the `?token=` extractor is opt-in: `verifier.WithQueryToken()`.
- Server-side revocation is available through `verifier.WithDefaultBlocklist()` (in-memory); a blocklisted request can then be invalidated with `ctx.Logout()`.
- Both sides support AES-GCM payload encryption: `signer.WithEncryption(key, nil)` and `verifier.WithDecryption(key, nil)`.
- Production systems should prefer asymmetric algorithms (`jwt.EdDSA`, RSA/ECDSA variants) so verifying services never hold the signing key.

- The header check is exact: `alg`, `typ` and nothing else. A token from an identity provider such as Auth0 also names its key in a `kid` field, so it is refused with `jwt.ErrTokenAlg` until you set the verifier's optional `HeaderValidator` field. The `ValidateHeader` method of a `github.com/kataras/jwt` key set fits there. It loads the issuer's public keys from its JWKS URL and picks one by the `kid` of each token:

```go
import kjwt "github.com/kataras/jwt"

keySet, err := kjwt.NewRemoteKeySet(jwksURL)
if err != nil {
    panic(err)
}
defer keySet.Close()

verifier := jwt.NewVerifier(nil, nil)
verifier.HeaderValidator = keySet.ValidateHeader
```

Role and Permission Guards

Authorization in Iris does not require a framework feature: a guard is a plain handler that inspects the verified user and either calls `ctx.Next()` or writes an error and stops. Combined with the rest/errors codes, guards stay small and composable:

```go
package main

import (
    "slices"

    "github.com/kataras/iris/v14"
    "github.com/kataras/iris/v14/auth"
    "github.com/kataras/iris/v14/rest/errors"
)

// RequireRoles allows the request only when the verified customer
// holds every required role. Register it AFTER VerifyHandler.
func RequireRoles(required ...string) iris.Handler {
    return func(ctx iris.Context) {
        user := auth.GetUser[Customer](ctx)

        for _, role := range required {
            if !slices.Contains(user.Roles, role) {
                errors.PermissionDenied.Message(ctx,
                    "missing required role: "+role)
                return
            }
        }

        ctx.Next()
    }
}
```

Mount guards per router, in order: authentication first, then authorization:

```go
func (api *AccountAPI) Routing(r iris.Router) {
    // ...
    admin := r.Router("/admin",
        api.sso.VerifyHandler(), RequireRoles("admin"))
    admin.Get("/customers", api.listCustomers)
}
```

The status codes carry the semantics: a missing or invalid token is 401 (`errors.Unauthenticated`) because the caller is unknown; a known caller lacking rights is 403 (`errors.PermissionDenied`). Fine-grained, per-resource decisions ("is this user the owner of order 42?") belong in the handler or service layer, where the resource is already loaded. Compare against `auth.GetUser[Customer](ctx).ID` there instead of writing a guard.

The same pattern works for permission strings, plan tiers or any other claim your user type carries; because guards are ordinary handlers, they also slot directly into the builder's `API` step:

```go
API("/admin", sso.VerifyHandler(), RequireRoles("admin"),
    NewAdminAPI).
```

Session-Based Authentication

For browser applications that render pages and keep state on the server, `middleware/sessions` provides typed sessions with secure-by-default cookies. In v14, `sessions.NewOptions` returns options with `Secure: true`, `SameSite: Lax` and a 24-hour `MaxAge`; session cookies are always HttpOnly.

```go
package main

import (
    "time"

    "github.com/kataras/iris/v14"
    "github.com/kataras/iris/v14/middleware/sessions"
    "github.com/kataras/iris/v14/rest/errors"
)

// UserSession is the typed session payload.
type UserSession struct {
    UserID   string
    Username string
    Roles    []string
    SigninAt time.Time
}

func main() {
```

```
    // Persist these keys; regenerating them invalidates all sessions.
    hashKey := []byte("hash-key-32-characters-long-str!")  // 32 or 64
    blockKey := []byte("block-key-24-characters!") // 16, 24 or 32

    opts := sessions.NewOptions(hashKey, blockKey)
    opts.MaxAge = 8 * time.Hour
    // Secure=true and SameSite=Lax are the defaults. Loosen only for
    // local plain-HTTP development:
    // opts.Secure = false

    db := sessions.NewCookieDatabase[UserSession](opts)

    app := iris.New()

    site := app.Router("/", sessions.Handler(db, "app_session"))
    site.Post("/signin", signin)
    site.Post("/signout", signout)
    site.Get("/profile", requireUser, profile)

    app.Listen(":8080")
}
```

`sessions.Handler` loads (or creates) the session before your handlers run and commits it afterwards when the request changed it: its data, a flash message, its id or its `Options`. A new session counts as changed. It does not commit when a handler already did, or when the response is an error status. A cookie it cannot decode, because it was signed with other keys or is older than `MaxAge`, counts as no cookie: the request runs with a new session (`IsNew` is true) and the commit replaces the cookie. Only a failure of the store itself answers 500.

A request that only reads the session, such as `GET /profile` above, sends no cookie and writes nothing to the store. That matters when requests overlap: a read that is still running while another request signs the visitor out has nothing to store, so it cannot bring the session back. Two requests that both change the session at the same moment still lose one of the two writes.

That is the rule for the three databases of the package: the cookie one above, and the Redis and Postgres ones below. The session of a `Database` you write yourself is committed after every request, as before.

The same rule sets the lifetime. The eight hours of `MaxAge` count from the last change, not from the last request, so a visitor who signs in and then only reads is signed out eight hours after signing in. To keep the session alive on a route that only reads it, call `session.Commit(ctx)` in that handler: called by hand it saves the session and sends the cookie again, changed or not. A deployment that changes `Secure`, `Path` or `MaxAge` in the session options waits in the same way: it reaches a visitor at their next change of the session, not at their next request.

A read that sends no cookie has one consequence to check. `Set-Cookie` was what kept such a page out of shared caches, and now nothing does. A reverse proxy that skips responses with `Set-Cookie` can store one visitor’s page and serve it to the next. When such a proxy caches in

front of the application, send `Cache-Control: private` on every page that depends on the session: `ctx.SetHeader("Cache-Control", "private")` in the handler. The `httpcache` middleware is safe without the header as long as its handler is registered before `sessions.Handler`: it caches a request with cookies only when its key names every one of them (Chapter 14).

Sign-In with Fixation Defense

A session fixation attack plants a known session identifier in the victim's browser before sign-in, then reuses it once the session becomes privileged. The defense is to rotate the identifier at the moment the privilege level changes, which is exactly what `RegenerateID` does:

```
    ctx.OK(iris.Map{"message": "signed in"}, nil)
}
```

Set the authenticated data *before* calling `RegenerateID`: it persists the session itself, so no extra `Commit` is required. The Handler commits nothing more after it, so data set after the call is never stored on Redis or Postgres.

Guarding and Sign-Out

```go
func requireUser(ctx iris.Context) {
    session := sessions.Get[UserSession](ctx)
    if session == nil || session.Data.UserID == "" {
        errors.Unauthenticated.Message(ctx, "sign in required")
        return
    }
    ctx.Next()
}

func profile(ctx iris.Context) {
    session := sessions.Get[UserSession](ctx)
    ctx.JSON(session.Data)
}

func signout(ctx iris.Context) {
    session := sessions.Get[UserSession](ctx)
    if err := session.Delete(ctx); err != nil {
        errors.Internal.Err(ctx, err)
        return
    }
    ctx.OK(iris.Map{"message": "signed out"}, nil)
}
```

Server-Side Stores

The cookie database keeps everything in the (encrypted, authenticated) cookie itself: zero infrastructure, but sessions cannot be revoked server-side and the payload must stay small. For revocable sessions use a server-side store; only an opaque ID travels in the cookie:

```go
// Redis (third-party client: github.com/redis/go-redis/v9).
client := redis.NewClient(&redis.Options{Addr: "localhost:6379"})
db := sessions.NewRedisDatabase[UserSession](client, opts)

// PostgreSQL (third-party pool: github.com/jackc/pgx/v5/pgxpool).
db := sessions.NewPostgresDatabase[UserSession](
    sessions.DefaultTableName, conn, opts)
```

Both implement the same `sessions.Database[T]` interface, so the `sessions.Handler` registration and every handler above stay unchanged. With a server-side store, `RegenerateID` performs a real rotation.

Identity Server and OAuth2

Two heavier subpackages build on the auth SDK. Both are overviews here; each ships with its own documentation.

First, the terms. An identity provider is a service whose only job is knowing who users are, so many applications can share one sign-in. OAuth 2.0 is the standard protocol those applications use to obtain access on a user's behalf without ever seeing the user's password: the user authenticates at the identity provider, which hands the application a token with agreed-upon scope. Its different *flows* are choreographies for different client shapes: a web app that can redirect the browser (authorization code), a TV or CLI with no browser (device code), a trusted backend service (password grant, machine-to-machine).

auth/identity: a Complete Identity Server

`auth/identity` is a production identity server you mount into an Iris application. It is backed by PostgreSQL (via `github.com/kataras/pg`) and provides, out of the box:

- User management under `/u`: signup, signin, refresh, signout, password reset, soft delete and restore, bulk operations, paginated and filtered listing, and user groups.
- Declarative user schema validation configured in YAML (text, number, email, date, phone, bool, uuid, slice, map field types).
- OAuth2 client applications: user applications (authorization code and device code flows) and server applications (password grant, machine-to-machine) via `AddUserApplication`, which takes an `identity.UserApplicationRegistration` value, and `AddServerApplication`.
- Client management for an admin client: `AddAdminApplication` registers a server application that also holds the `clients:*` scopes, and with its token you create, list, count, read and delete clients, issue a client a new token and revoke its tokens under `/clients`. The SDK wraps these as `AdminCreateClient`, `AdminListClients`, `AdminCountClients`, `AdminGetClient`, `AdminDeleteClient`, `AdminIssueClientToken` and `AdminRevokeClientTokens`.
- The OAuth2 endpoints of the section below, plus `/.well-known/jwks.json`, optional Redis-backed response caching and usage metering.

```go
import "github.com/kataras/iris/v14/auth/identity"

identityServer := identity.Load("identity.yml")
app.Router("/auth", identityServer)
```

A typed Go client for the whole HTTP API lives in `auth/identity/sdk`: `sdk.New[Profile](opts)` gives admin operations (signup, list, partial updates), `UserSignin` via the OAuth2 password grant, and `sdk.NewAuth[Profile]`, a ready `*auth.Auth[Profile]` that verifies identity-server tokens in your own services. See the `auth` package README (`auth/README.md`) in the framework repository for details.

auth/oauth2: an OAuth2 Authorization Server

`auth/oauth2` implements the server side of OAuth 2.0, generic over your user data type. It supports the authorization code flow (with a built-in or custom consent page), the device code flow for browserless devices, the password grant for trusted server applications, and refresh token rotation. Mounting the server registers:

- `GET /authorize`, `GET|POST /consent`, `GET /consent/revoke`
- `POST /device/code`
- `POST /token`
- `POST /token/introspect`

Client and code stores have in-memory and PostgreSQL implementations. Authorization is expressed as resource-action policies (`oauth2.ResourcePolicy`) enforced by the `oauth2.AllowPolicies` middleware, with optional Open Policy Agent (Rego) integration under `auth/oauth2/opa`. The package README (`auth/oauth2/README.md`) walks through a complete server and all four client flows.

Summary

- Pick the lightest tool that fits: `basicauth` for internal surfaces, the `auth` SDK for user-facing token APIs, `middleware/jwt` for raw token verification, `middleware/sessions` for browser state, and `auth/identity` / `auth/oauth2` when you need a standalone identity provider.
- `auth.Auth[T]` covers the full token lifecycle with five handlers and one middleware; you contribute the YAML key configuration and a `Provider[T]` over your user store. Register the instance as a service and apply `VerifyHandler` per router, with optional `VerifyUserFunc` claims checks; read the user with `auth.GetUser[T]`.
- Authorization is plain handlers: guards that call `ctx.Next()` or respond with `errors.PermissionDenied` (403), distinct from `errors.Unauthenticated` (401).
- v14 defaults are secure: the JWT verifier reads the `Authorization` header only, and session cookies default to `Secure`, `SameSite=Lax` and HttpOnly. Call `RegenerateID` after sign-in to defeat session fixation.

- Testing authentication flows with `httptest` is covered in Chapter 16; broader hardening guidance lives in Chapter 18.

Further Reading

- MDN: HTTP authentication: the `Authorization` header, challenges and the Basic scheme.
- MDN: Using HTTP cookies: how cookies work, with `HttpOnly`, `Secure` and `SameSite` explained.
- RFC 7519 — JSON Web Token (JWT): the token structure and the registered claims (`exp`, `iss`, `sub`, `jti`) behind `StandardClaims`.
- OWASP Authentication Cheat Sheet: practical guidance on sign-in flows, credential handling and errors.
- OWASP Password Storage Cheat Sheet: current recommendations for bcrypt and friends, including cost factors.
- OWASP Session Management Cheat Sheet: session identifiers, expiration and the fixation attack that `RegenerateID` defeats.

CHAPTER 11

Configuration Management

Iris keeps its configuration in one plain struct: `iris.Options`. You build the value in code, load it from YAML, or read it from an environment variable, then hand it to `iris.New` or `iris.NewBuilder`. Every field is optional because the framework merges what you provide with its defaults. This chapter walks through the struct, the merge rules that make partial configuration safe, the loading helpers, the router and context option tours, and the server options passed to `app.Run`.

Background

Configuration is every value your program needs that is not part of its logic: which port to listen on, how verbose the logs are, which database to talk to, which password unlocks the admin account. The same compiled binary typically runs in several environments (your laptop, a staging server, the production cluster) and each needs different values. If those values live in the source code, every difference forces a rebuild, and every credential becomes visible to anyone who can read the code. The twelve-factor methodology, a widely cited set of principles for deployable applications, distills this into one rule: keep configuration in the environment, so the code stays identical across deployments and only the surrounding values change.

Two carriers for configuration appear throughout this chapter. A **YAML file** is a plain-text format for structured data: `key: value` pairs, nesting expressed by indentation, lists introduced with `-`. People can read and edit it easily, which makes it the usual choice for settings that live next to the code, such as development defaults. An **environment variable** is a named string the operating system hands a process when it starts. It is set outside the program (by the shell, a container runtime, a systemd unit or an orchestration tool) and read at run time with a call like `os.Getenv`. Because every deployment platform knows how to inject environment variables, they are the natural channel for anything that changes per deployment.

A **secret** is a configuration value that grants access: a password, an API key, a signing key, a database connection string. Secrets never belong in a repository, even a private one. Version control remembers everything, so a key committed once stays in the history long after the file is "fixed"; clones, forks, backups and CI machines all receive a copy; and revoking a leaked key is far more painful than never leaking it. The safe home for secrets is the environment, filled either by hand or by a secret manager (a vault service that stores secrets encrypted and injects them at deploy time).

Iris makes this discipline cheap to follow. One struct, `iris.Options`, holds all framework settings; it merges with sensible defaults, loads from YAML or a single environment variable, and embeds into your own config struct. What disappears from your code is the configuration plumbing itself: no flag-parsing forest, no per-environment build tags, no hand-rolled merge of defaults against overrides.

The Options Struct

```go
package main

import "github.com/kataras/iris/v14"

func main() {
    app := iris.New(iris.Options{
        Name:     "bookstore",
        LogLevel: "debug",
        Host:     "example.com",
    })

    app.Get("/", func(ctx iris.Context) {
        ctx.JSON(iris.Map{"status": "ok"})
    })

    app.Listen(":8080")
}
```

`iris.Options` has five fields, all optional:

Field	Type	Purpose
`Name`	`string`	Application name, used as the log prefix of the application's logger.
`LogLevel`	`string`	`"disable"`, `"fatal"`, `"error"`, `"warn"`, `"info"` (default) or `"debug"`.
`Host`	`string`	The trusted domain the server runs on, e.g. `"example.com"`. Also copied to `Router.Host`, which subdomain routing relies on.
`Router`	`iris.RouterOptions`	Routing behavior for all routes.

`Context`	`iris.ContextOptions`	Request and response handling for all routes.

`Options` implements the `iris.Configurator` interface, and `iris.New` accepts any number of configurators. For imperative setup steps you can also pass an `iris.ConfiguratorFunc(func(app *iris.Application) { ... })`, which runs against the application instance at construction time.

Merge Semantics

An `Options` literal passed to `iris.New` merges with `iris.DefaultOptions`: every field left at its zero value is filled from the defaults. A partial literal therefore configures only what it sets. The example above changes the name and the log level; the charset stays `"utf-8"`, the multipart form limit stays 32 MB, and so on.

Two refinements to that rule matter in practice.

A non-nil empty slice or map is an explicit "none". `nil` means "unset, give me the default", while an allocated empty value means "I want nothing here". The field where the distinction is visible out of the box is `RemoteAddrPrivateSubnets`, whose default is a list of common private IP ranges:

```go
import (
    "github.com/kataras/iris/v14"
    "github.com/kataras/iris/v14/core/pkg/netutil"
)

app := iris.New(iris.Options{
    Context: iris.ContextOptions{
        // nil would inherit the default private ranges;
        // an empty non-nil slice clears them.
        RemoteAddrPrivateSubnets: []netutil.IPRange{},
    },
})
```

A meaningful zero cannot be expressed at construction. For scalar fields the merge cannot tell "unset" apart from "deliberately zero". Setting `URLParamSeparator: ""` in the literal does nothing, because the empty string counts as unset and is replaced by the default `","`. The escape hatch is per-request mutation: `ctx.Options()` returns a request-scoped copy of the context options that a middleware may edit freely, and the change dies with the request:

```go
app.Use(func(ctx iris.Context) {
    // Treat "a,b" as one value instead of splitting on commas,
    // for this request only.
    ctx.Options().URLParamSeparator = ""
    ctx.Next()
})
```

The mechanism is not limited to zeros. Any handler can adjust the context options for the request it serves. A route group that accepts large uploads, for example, can raise `BodyMaxMemory` in its middleware while the rest of the application keeps the strict global cap.

Defaults and EditOptions

`iris.DefaultOptions` is the exported default configuration. When you want the defaults plus a few edits, `iris.Defaults` reads better than a literal because you modify the defaults in place:

```go
app := iris.New(iris.Defaults(func(opts *iris.Options) {
    opts.LogLevel = "debug"
    opts.Context.RemoteAddrHeaders = []string{"X-Forwarded-For"}
}))
```

After construction, `app.EditOptions` applies further changes:

```go
app := iris.New()

app.EditOptions(func(opts *iris.Options) {
    opts.LogLevel = "warn"
    opts.Context.BodyMaxMemory = 8 * iris.MB
})
```

`EditOptions` must run before the application is built, that is before `Build`, `Listen` or `Run`. Once built it only logs a warning and the change may not be applied, so never call it from inside a request handler. For per-request changes use `ctx.Options()` as shown above.

YAML Files and the SERVER_CONFIG Variable

Two helpers load `Options` from the outside world.

`iris.YAML` reads a YAML file, applies it over the defaults and panics when the file is missing or malformed. It fits programs where the file is required:

```go
app := iris.New(iris.YAML("config.yml"))
```

The YAML keys follow the struct's YAML tags, which are CamelCase and nest the router and context sections:

```yaml
Name: bookstore
LogLevel: info
Host: example.com
Router:
    ForceLowercaseRouting: true
    FireMethodNotAllowed: true
Context:
    BodyMaxMemory: 4194304 # 4 MB
    RemoteAddrHeaders:
        - X-Forwarded-For
```

`(*Options).Bind` is environment-first: it reads the `SERVER_CONFIG` environment variable as a full YAML document and only falls back to the given file when the variable is empty. It returns an error instead of panicking:

```go
package main

import (
    "log"

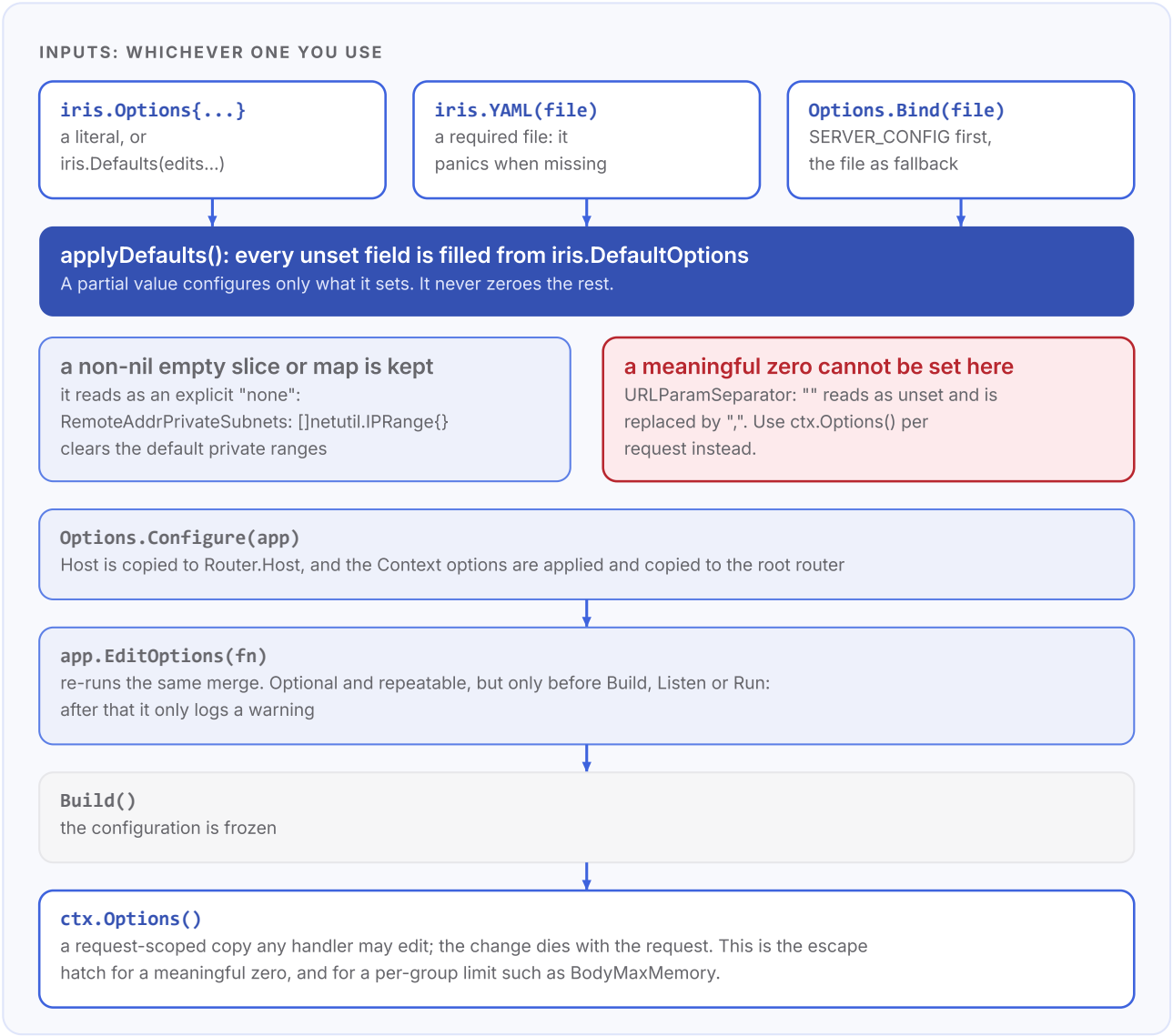
    "github.com/kataras/iris/v14"
)

func main() {
    var opts iris.Options
    if err := opts.Bind("config.yml"); err != nil {
        log.Fatal(err)
    }

    app := iris.New(opts)
    app.Listen(":8080")
}
```

One container variable can carry the entire configuration, so a deployment does not need to mount files. Fields the document leaves out still inherit the defaults when the value reaches `iris.New`.

All four inputs meet at the same merge, and the two rules that catch people out are consequences of it: an empty-but-non-nil collection survives, and a meaningful zero does not.



Whichever way you supply options, every field you left unset is filled from the defaults, Configure applies them, EditOptions may re-run the merge until Build freezes it, and ctx.Options() is the per-request escape hatch afterwards.

Router Options

`iris.RouterOptions` controls matching behavior for all routes. Every boolean defaults to false.

Field	Effect
`Host`	The server's host, including a non-standard port. Set automatically from `Options.Host`; subdomain routing depends on it.
`DisablePathCorrection`	By default a request to `/home/` is redirected (301) to a registered `/home`. Set to true to disable.

`DisablePathCorrectionRedirection`	With path correction on, execute the corrected route's handler directly instead of redirecting.
`EnablePathIntelligence`	Redirect not-found GET requests to the closest registered path, e.g. `/cont` to `/contact`.
`EnablePathEscape`	Keep encoded characters such as `%2F` inside path parameters.
`ForceLowercaseRouting`	Register and match all paths lowercased.
`AllowQuerySemicolons`	Convert unescaped `;` in the query to `&` (the pre-Go 1.17 behavior).
`EnableDynamicHandler`	Let routes change while the server runs: register or remove them, then call `app.BuildRouter()` again, from one goroutine at a time. The rebuild swaps in the new routes and the 404 and 405 handlers; router wrappers, `UseRouter` handlers and the router options, this one included, are fixed at the first build. Costs a read lock on every request while it is on. With it off, a second `BuildRouter` returns an error and a route registered after the build is not served.
`FireMethodNotAllowed`	Answer 405 instead of 404 when the path matches but the method does not. A builder application answers it with the `rest/errors` JSON envelope; for your own body, register `HandleNoMethod` from a `Configurator` passed to `NewBuilder`, because the builder fills that slot at `Build` (Chapter 3, Chapter 4).

```go
app := iris.New(iris.Options{
    Router: iris.RouterOptions{
        ForceLowercaseRouting: true,
        FireMethodNotAllowed:  true,
    },
})
```

Context Options

`iris.ContextOptions` configures request and response handling. The same struct is editable per request through `ctx.Options()`.

Request fields:

Field	Default	Effect
`URLParamSeparator`	`","`	Separator used to split multi-value entries in `ctx.Query()`, `ctx.Form()` and `ctx.PostForm()`.
`FireEmptyFormError`	`false`	Make `ReadForm`, `ReadQuery` and `ReadBody` return `iris.ErrEmptyForm` on empty form data instead of succeeding silently.

`TimeFormat`	`"Mon, 02 Jan 2006 15:04:05 GMT"`	Datetime parsing format, used for example by `ctx.CheckIfModifiedSince`.
`PostMaxMemory`	32 MB	Memory cap for multipart form parsing.
`BodyMaxMemory`	0 (no limit)	Byte cap for `Body`, `ReadJSON`, `ReadJSONStream`, `ReadMultipartRelated` and the readers built on them, including `ctx.BindJSON`. Exceeding it fails with the `context` package's `ErrRequestEntityTooLarge`, which `rest/errors` maps to 413.
`RemoteAddrHeaders`	none	Header names trusted to reveal the real client IP in `ctx.RemoteAddr()`, read in this order and each one from right to left (Chapter 5). Empty by default: no `X-` header is trusted unless you list it.
`RemoteAddrHeadersForce`	`false`	When the headers hold no public address, use the first address from the left of a header whose entries are all addresses, even though it falls inside a private subnet. A header with an entry that is not an address, such as `unknown`, gives nothing with the option on or off.
`RemoteAddrPrivateSubnets`	common private ranges	IP ranges treated as private when resolving the client IP.
`SSLProxyHeaders`	none	Header name/value pairs that mark a request as HTTPS for `ctx.IsSSL()`, e.g. `{"X-Forwarded-Proto": "https"}`.
`HostProxyHeaders`	none	Headers that may carry the proxied hostname for `ctx.Host()`.
`RequestValidator`	`nil`	Called after body decoding on `ReadJSON`, `ReadForm`, `ReadQuery` and friends.

Response fields:

Field	Default	Effect
`Charset`	`"utf-8"`	Character set appended to rendered Content-Type headers.
`ErrorHandler`	framework default	Handles errors produced while writing responses through `JSON`, `XML`, `Render` and the other writers.
`DisableErrorHandler`	`false`	Turn the write-error handler off entirely.

In production, requests often reach your server through a **reverse proxy**: an intermediary such as a load balancer, nginx or a CDN that accepts the client's connection and forwards it to your application. To your program every request then appears to come from the proxy's own IP, often

over plain HTTP, so proxies attach headers such as `X-Forwarded-For` (the original client IP) and `X-Forwarded-Proto` (the original scheme). These headers are ordinary text that any client could forge, which is why Iris trusts none of them until you explicitly list the ones your own proxy sets.

A typical proxy-aware production configuration:

```go
app := iris.New(iris.Options{
    Context: iris.ContextOptions{
        BodyMaxMemory:     8 * iris.MB,
        RemoteAddrHeaders: []string{"X-Forwarded-For", "X-Real-Ip"},
        SSLProxyHeaders: map[string]string{
            "X-Forwarded-Proto": "https",
        },
    },
})
```

`BodyMaxMemory` is the application-wide cap; for per-route limits use the `bodylimit` middleware instead (Chapter 12).

`RequestValidator` accepts any value with a `Struct(any) error` method. The third-party `github.com/go-playground/validator/v10` package satisfies it directly:

```go
import "github.com/go-playground/validator/v10"

app.EditOptions(func(opts *iris.Options) {
    opts.Context.RequestValidator = validator.New()
})
```

See Chapter 5 for the body readers themselves.

Embedding Options in Your Own Config

Real applications have settings of their own next to the framework's. Embedding `iris.Options` inline in the application's config struct makes one YAML document configure both, and the loader below is that shape written out in full, because it is worth seeing once. It is not what the tutorial or the rest of this book does any more: they embed `iris.BaseConfiguration`, which carries `Host`, `Port`, `AllowOrigin`, `EnableCompression`, `RequestLog` and the framework options under an `Iris` key, and they load it with the function the framework ships. The next section is the version to copy into a new service.

```go
package main

import (
    "fmt"
    "log"
```

```
    "os"

    "gopkg.in/yaml.v3"

    "github.com/kataras/iris/v14"
)

// Config holds the whole application configuration.
// AdminPassword is yaml:"-" on purpose: secrets come from the
// environment, never from a committed file.
type Config struct {
    iris.Options `yaml:",inline"`

    Addr          string `yaml:"Addr"`
    Env           string `yaml:"Env"`
    AdminUser     string `yaml:"AdminUser"`
    AdminPassword string `yaml:"-"`
}

// LoadConfig starts from working defaults, overlays YAML from the
// APP_CONFIG environment variable or the config.yml file (both
// optional), then reads the secrets from the environment.
func LoadConfig() (Config, error) {
    cfg := Config{
        Options:   iris.DefaultOptions,
        Addr:      ":8080",
        Env:       "development",
        AdminUser: "admin",
    }

    if raw := os.Getenv("APP_CONFIG"); raw != "" {
        if err := yaml.Unmarshal([]byte(raw), &cfg); err != nil {
            return cfg, fmt.Errorf("config from env: %w", err)
        }
    } else if data, err := os.ReadFile("config.yml"); err == nil {
        if err = yaml.Unmarshal(data, &cfg); err != nil {
            return cfg, fmt.Errorf("config.yml: %w", err)
        }
    }

    cfg.AdminPassword = os.Getenv("ADMIN_PASSWORD")
    if cfg.AdminPassword == "" {
        return cfg, fmt.Errorf("ADMIN_PASSWORD is required")
    }
    return cfg, nil
}

func main() {
    cfg, err := LoadConfig()
    if err != nil {
        log.Fatal(err)
    }

    app := iris.New(cfg.Options)
    // With the builder instead: iris.NewBuilder(cfg.Options)...
    // (see Chapter 3).

    app.Listen(cfg.Addr)
}
```

The committed `config.yml` contains settings only, no secrets:

```yaml
Name: bookstore
LogLevel: info
Addr: ":8080"
Env: development
AdminUser: admin
```

The `yaml:",inline"` tag makes the framework keys (`Name`, `LogLevel`, `Host`, `Router`, `Context`) appear at the top level of the same document as your own keys. The loading order mirrors `Options.Bind`: environment variable first, file as the development fallback. Never commit credentials, key material or connection strings, even in private repositories; the environment, or a secret manager feeding it, is their only home.

The loader the framework ships

That `LoadConfig` function is the same forty-five lines in every service that has one, so the framework has its own:

```go
type Configuration struct {
    iris.BaseConfiguration `yaml:",inline"`

    DatabaseURL string `yaml:"DatabaseURL"`
}

func main() {
    c, err := iris.LoadConfiguration[Configuration](
        "MY_SERVICE_CONFIG", "configuration.yml")
    if err != nil {
        log.Fatal(err)
    }

    app := iris.New(c.Iris)
    // ...routes...
    app.Listen(c.Addr())
}
```

`iris.LoadConfiguration[T]` reads a YAML document into `T` from the named environment variable when it is set, otherwise from the file. The variable may hold either the document *or* a path to it: a value naming an existing file is read as a file and anything else is decoded as YAML, so the same binary works whether the deployment mounts the configuration or passes it inline. It is not a generic `Options.Bind`; that one is hardcoded to `SERVER_CONFIG` and always reads its value as a document.

If `T` (or `*T`) has a `BindEnv()` method it runs *after* decoding, which is what makes the environment win over the document. Embedding `iris.BaseConfiguration` gets that for the fields every service shares:

Field	YAML key	Meaning
`Host`	`Host`	The interface to listen on; empty means every interface. Not `Iris.Host`, which is the trusted domain the router matches subdomains against
`Port`	`Port`	The TCP port; zero means 8080
`AllowOrigin`	`AllowOrigin`	The origin line, as `Builder.AllowOrigin` accepts it
`EnableCompression`	`EnableCompression`	As `Builder.Compression` accepts it
`RequestLog`	`RequestLog`	As `Builder.LogRequests` accepts it
`Iris`	`Iris`	The framework `Options`, as `iris.New` accepts them

Its `BindEnv` reads `HOST`, `PORT`, `HTTP_PLATFORM_PORT` and `ALLOW_ORIGIN`. `HTTP_PLATFORM_PORT` wins over `PORT`, because a hosting platform that picks the port passes it there while `PORT` may still carry the value baked into the image. A non-numeric port is ignored rather than turned into zero, so a typo cannot silently move the server to 8080. `Addr()` returns the `host:port` string for `app.Listen`, joined so that an IPv6 host is bracketed correctly.

There is deliberately no `app.Serve(host, port)`. `Listen` and `Run` stay the only two entry points, and `Addr()` is the part that was missing.

Multi-Environment Setups

Because the whole configuration travels as one YAML document, environments do not need file hierarchies or override matrices. Keep the development file in the repository, and let each deployment carry its own document in the environment variable:

```yaml
Name: bookstore
LogLevel: warn
Host: api.example.com
Addr: ":8080"
Env: production
Context:
    RemoteAddrHeaders:
        - X-Forwarded-For
    SSLProxyHeaders:
        X-Forwarded-Proto: https
```

```sh
APP_CONFIG="$(cat deploy/production.yml)" ./server
```

A Kubernetes ConfigMap, a systemd unit or a compose file sets the same variable the same way. Unset fields still merge with the defaults, so each document stays as small as the differences between environments.

Server Options

Serving is configured separately from the application, at the call to `app.Run`. `app.Listen(addr)` is the shorthand; the two are equivalent:

```go
app.Listen(":8080")

// is the same as:
app.Run(iris.ServerOptions{
    Addr:           ":8080",
    Network:        "tcp",
    KeepAlive:      3 * time.Minute,
    SocketSharding: true,
})
```

The fields of `iris.ServerOptions`:

Field	Purpose
`Addr`	Listen address in `"host:port"` form. Empty means `":http"`, or `":https"` when `TLSConfig` is set.
`Network`	`"tcp"` (default), `"tcp4"`, `"tcp6"`, `"unix"` or `"unixpacket"`.
`KeepAlive`	Keep-alive period for accepted connections; negative disables keep-alives.
`SocketSharding`	`SO_REUSEPORT` (`SO_REUSEADDR` on Windows) for multi-CPU scaling.
`Hosts`	Hostnames the TLS certificate is valid for; falls back to the application's `Host`, then to `Addr`.
`TLSConfig`	Custom `*tls.Config`.
`CertFile`, `KeyFile`	Paths to PEM-encoded certificate and key files.
`Letsencrypt`	Automatic HTTPS via Let's Encrypt.
`Ngrok`, `SSHTunnel`	Expose the local server through an ngrok or SSH reverse tunnel (Chapter 19). `Ngrok` needs the blank import `_ "github.com/kataras/iris/v14/core/server/ngrok"`; without it the server refuses to start.
`Timeout`	`*iris.TimeoutOptions`, see below.
`LogWriter`	Destination for errors surfaced by the underlying `http.Server`.

`DisableStartupLog`	Suppress the startup message.
`DisableInterruptHandler`	Skip the built-in interrupt handler that shuts the server down gracefully on CTRL+C.

TLS (Transport Layer Security) is the encryption layer that turns HTTP into HTTPS. A server proves its identity with a **certificate** (a file, signed by a trusted certificate authority, that binds a hostname to a public key) and keeps the matching private key to itself. Let's Encrypt is a certificate authority that issues certificates free of charge through an automated protocol; the `Letsencrypt` option drives that protocol for you, so the server obtains and renews its own certificates instead of you installing files. It proves ownership of the domain on the TLS listener itself (the tls-alpn-01 challenge), so Let's Encrypt has to reach this server on port 443 of each host; nothing is served on port 80 for it.

Use one TLS mechanism at a time: `TLSConfig`, the `CertFile` / `KeyFile` pair, `Letsencrypt`, `Ngrok` or `SSHTunnel`. The one sanctioned combination is `Letsencrypt` with `CertFile` / `KeyFile`, where the files act as a local fallback when the Let's Encrypt challenge fails.

Serving TLS from certificate files:

```go
app.Run(iris.ServerOptions{
    Addr:     ":443",
    CertFile: "server.crt",
    KeyFile:  "server.key",
})
```

Automatic HTTPS:

```go
app.Run(iris.ServerOptions{
    Addr:        ":443",
    Hosts:       []string{"api.example.com"},
    Letsencrypt: true,
})
```

Timeouts

A timeout is the maximum time you allow a network operation before giving up on it. Servers need timeouts because clients can be slow or hostile on purpose: a connection trickling one byte per second would otherwise hold memory and a file descriptor forever. Keep-alive, mentioned below, is the practice of reusing one TCP connection for several requests instead of opening a fresh connection each time; the idle timeout bounds how long such a reused connection may sit waiting for its next request.

`iris.TimeoutOptions` holds the standard `net/http` server timeouts plus an optional handler timeout:

Field	Effect
`ShutdownOnInterruptTimeout`	How long graceful shutdown may take on interrupt: the server's own drain, and the deadline of the context that `iris.RegisterOnInterrupt` functions receive. 15 seconds from `iris.Timeout()`, or when `Timeout` is nil. Zero or negative sets no limit, and zero is what a `&iris.TimeoutOptions{...}` literal carries.
`ServerTimeout`	One knob from which the read-header, write and idle timeouts are derived.
`ReadTimeout`	Maximum duration to read the whole request, body included.
`ReadHeaderTimeout`	Time allowed to read request headers; handlers keep control over body deadlines.
`WriteTimeout`	Maximum duration for writing the response.
`IdleTimeout`	Keep-alive wait between requests.
`HandlerTimeout`	Wraps the router in `http.TimeoutHandler` when greater than zero.
`HandlerTimeoutMessage`	HTML body sent when the handler timeout fires.

Start from the `iris.Timeout()` builder so the shutdown timeout and the default timeout message stay in place, then set what you need:

```go
package main

import (
    "time"

    "github.com/kataras/iris/v14"
)

func main() {
    app := iris.New()

    app.Get("/", func(ctx iris.Context) {
        ctx.JSON(iris.Map{"status": "ok"})
    })

    app.Run(iris.ServerOptions{
        Addr: ":8080",
        Timeout: iris.Timeout().
            WithServerTimeout(30 * time.Second).
            WithHandlerTimeout(15 * time.Second),
    })
}
```

Unless `DisableInterruptHandler` is set, the server catches the interrupt signal, stops accepting connections and waits up to `ShutdownOnInterruptTimeout` for in-flight requests to finish. `app.Run` returns nil when the shutdown is over, not when it begins: after that wait, and when the

shutdown is an `app.Shutdown`, after the closers it runs as well (Chapter 19). The same option is the deadline of the context a function registered with `iris.RegisterOnInterrupt` receives, so the shutdown that `iris.RegisterOnInterrupt(app.Shutdown)` starts is bounded by it too. A `Timeout` written as a `&iris.TimeoutOptions{...}` literal carries zero in that field unless you set it, and zero means no limit: the server then waits for the last request however long it takes. That is one more reason to start from `iris.Timeout()`.

Summary

- `iris.Options` (`Name`, `LogLevel`, `Host`, `Router`, `Context`) is the whole application configuration; every field is optional.
- Options merge with `iris.DefaultOptions`: partial literals are safe, a non-nil empty slice or map means "explicitly none", and meaningful zeros are set per request through `ctx.Options()` in a middleware.
- `iris.Defaults(edits...)` tweaks the defaults in place; `app.EditOptions` applies changes before the application is built, never from a handler.
- `iris.YAML("config.yml")` loads a required file; `(*Options).Bind` reads the `SERVER_CONFIG` environment variable first and falls back to a file.
- Embed `iris.Options` into your own config struct with `yaml:",inline"` for one document that configures both the framework and your application; keep secrets in the environment only.
- Serving is configured at `app.Run(iris.ServerOptions{...})`: address, TLS (files, custom config or Let's Encrypt), keep-alive, socket sharding and timeouts via `iris.Timeout()`.

Further Reading

- The Twelve-Factor App: Config: the canonical statement of why configuration belongs in the environment, separate from code.
- OWASP Secrets Management Cheat Sheet: practical guidance on storing, injecting and rotating secrets.
- YAML 1.2 specification: the definitive reference for the format read by `iris.YAML` and `Options.Bind`.
- yaml.v3 package documentation: the Go YAML library used in this chapter's embedding example.

- MDN: X-Forwarded-For: how proxy forwarding headers work and why they must not be trusted blindly.
- Let's Encrypt documentation: how automatic HTTPS certificate issuance and renewal work.

CHAPTER 12

Built-in Middleware

Iris ships thirty-one middleware packages under `github.com/kataras/iris/v14/middleware/`. Each one is a plain `iris.Handler` (or a small type that produces one), so they compose with your own handlers without adapters. This chapter is the tour: a catalog of everything that exists, what the `iris.NewBuilder` chain already installs for you, and a focused section with a working example for each middleware you are likely to reach for first. Topics that deserve a whole chapter get only a brief entry here with a pointer: authentication in Chapter 10, caching strategies in Chapter 14, i18n in Chapter 15 and observability in Chapter 17.

Background

Middleware, introduced in Chapter 2, is a handler that runs before or around your route handler. Every request flows through a chain: each middleware can inspect the request, attach data for later handlers, modify the response, pass control on with `ctx.Next()`, or stop the chain entirely. Cross-cutting concerns (the jobs every route needs but no route wants to implement) live naturally in this chain, written once and registered wherever they apply.

Each middleware family in this chapter exists because of a concrete problem:

- **CORS**: browsers deliberately block a page on one site from calling an API on another until the API states, through response headers, which foreign origins it accepts.
- **Rate limiting**: any public endpoint eventually gets hammered (by brute-force login scripts, scrapers or a buggy client stuck in a retry loop), so requests per client must be throttled.
- **Recovery**: one panicking handler would otherwise tear down its request with a blank response; crash containment turns the panic into a controlled, logged 500.
- **Compression and body limits**: text responses shrink several-fold when compressed, saving bandwidth on every reply. In the other direction, a client can send an arbitrarily large body and exhaust your memory unless a cap rejects it early.

- **Access logging**: when something goes wrong in production, the audit trail (who requested what, when, and how it ended) is the record you reconstruct events from.
- **Sessions and tokens**: HTTP is stateless, the server forgets each client between requests, so remembering "who is this" takes deliberate machinery.

"Batteries included" means these tools ship inside the framework rather than as third-party add-ons you assemble yourself. That saves more than typing: every package here is versioned with Iris, tested against the same context and router, and upgraded in lock-step, so the classic integration risks disappear: an adapter written for an older API, two libraries disagreeing about who writes headers first, a security default one package sets and another silently undoes. You can still plug in anything from the `net/http` ecosystem (see Adapting net/http Middleware); you just rarely need to.

The Middleware Catalog

Every package lives under `github.com/kataras/iris/v14/middleware/<name>`.

Middleware	Package	Purpose	Constructor form
Access log	`accesslog`	Structured request/response logging	`accesslog.File(path)` or `accesslog.New(w)`, then `.Handler`
API key	`apikey`	Shared-secret authentication for machine callers	`apikey.New(apikey.Options{Keys: ...})`
Basic auth	`basicauth`	HTTP Basic Authentication	`basicauth.Default(users)`, `basicauth.New(Options[U])`, `basicauth.Load(file)`
Body limit	`bodylimit`	Cap request body size (413 on excess)	`bodylimit.New(2 * iris.MB)`
Cache	`cache`	ETag, Cache-Control and friends	`cache.New(cache.Options{...})`
Compress	`compress`	Response compression negotiation	`compress.Handler`
CORS	`cors`	Cross-Origin Resource Sharing	`cors.New()....Handler()`
Counter	`counter`	Request counting with notifications	`counter.RequestCounter(counter.NewCounter[uint64]())`

Geolocation	`geolocation`	IP-based geolocation lookup	`geolocation.New(geolocation.Options{...})`
gRPC	`grpc`	Serve a gRPC server through Iris	`grpc.New(grpcServer)`
hCaptcha	`hcaptcha`	hCaptcha challenge verification	`hcaptcha.New(hcaptcha.Options{Secret: ...})`
HTTP cost	`httpcost`	Per-request cost (time, RAM, CPU)	`httpcost.New(httpcost.DefaultOptions)`
i18n	`i18n`	Translations and language detection	`i18n.NewHandler(locales, i18n.HandlerOptions{...})`
IP access	`ipaccess`	IP allowlist and blocklist	`ipaccess.New(allowed, blocked)`
JWT	`jwt`	Token signing and verification	`jwt.NewVerifier(alg, key).Verify(claims)`
Method override	`methodoverride`	PUT/DELETE from HTML forms	`methodoverride.New(methodoverride.Options{...})`
Mod revision	`modrevision`	Build and health info endpoint	`modrevision.New(modrevision.Options{...})`
Monitor	`monitor`	Process and OS statistics page	`monitor.New(monitor.Options{...})`
pprof	`pprof`	Go profiling endpoints	`app.Router("/debug/pprof", pprof.New())`
Rate	`rate`	Token-bucket rate limiting	`rate.Limit(limit, burst, rate.Options{...})`
reCAPTCHA	`recaptcha`	Google reCAPTCHA verification	`recaptcha.New(recaptcha.Options{Secret: ...})`
Recorder	`recorder`	Record request/response bodies	`recorder.Request` handler, `recorder.GetResponseRecorder(ctx)`
Recovery	`recovery`	Panic recovery	`recovery.Handler`
Referrer	`referrer`	Referer header parsing	`referrer.Handler`
Request ID	`requestid`	Unique per-request identifiers	`requestid.New(generator...)`
Rewrite	`rewrite`	Redirect and rewrite rules	`rewrite.New(rewrite.Options{...})`, `rewrite.Load(file)`
Server timing	`servertiming`	Server-Timing response header	`servertiming.Handler`, then `Start` / `End`

Sessions	`sessions`	Typed server-side sessions	`sessions.Handler(db, cookieName)`
Versioning	`versioning`	API versioning by header/query/subdomain	`versioning.New(versioning.Options{...})`
View	`view`	Template engine integration	`view.Handler(engine)`, then `view.Render`
WebSocket	`websocket`	WebSockets via neffos	`websocket.New(server, websocket.Options{})`

One convention runs through that last column. A package with settings exposes an `Options` struct whose zero value is valid, and a `New(opts Options) iris.Handler` that takes it; a package with nothing to configure exposes a plain `Handler` instead, so there is no empty constructor to call. Where construction can fail, `New` returns `(handler, error)` and a `Must` companion panics instead, for the common case of wiring at start-up. The handful of packages that produce something other than one handler, `accesslog` and `cors` among them, say so in the first paragraph of their package documentation.

That is why there are no functional options anywhere here: no `rate.PurgeEvery(...)`, no `methodoverride.HeadersOnly()`. Everything they used to set is a field on the package's `Options`.

Registration Levels

Where you register a middleware decides which requests it sees:

```go
app := iris.New()

// Root-level Use: every request, including 404 and 405 responses.
app.Use(handler)

// Child-level Use: only routes matched under this group.
api := app.Router("/api")
api.Use(handler)

// UseRouter: runs before routing itself — static files, rewrites,
// CORS preflights and everything else pass through it.
app.UseRouter(handler)
```

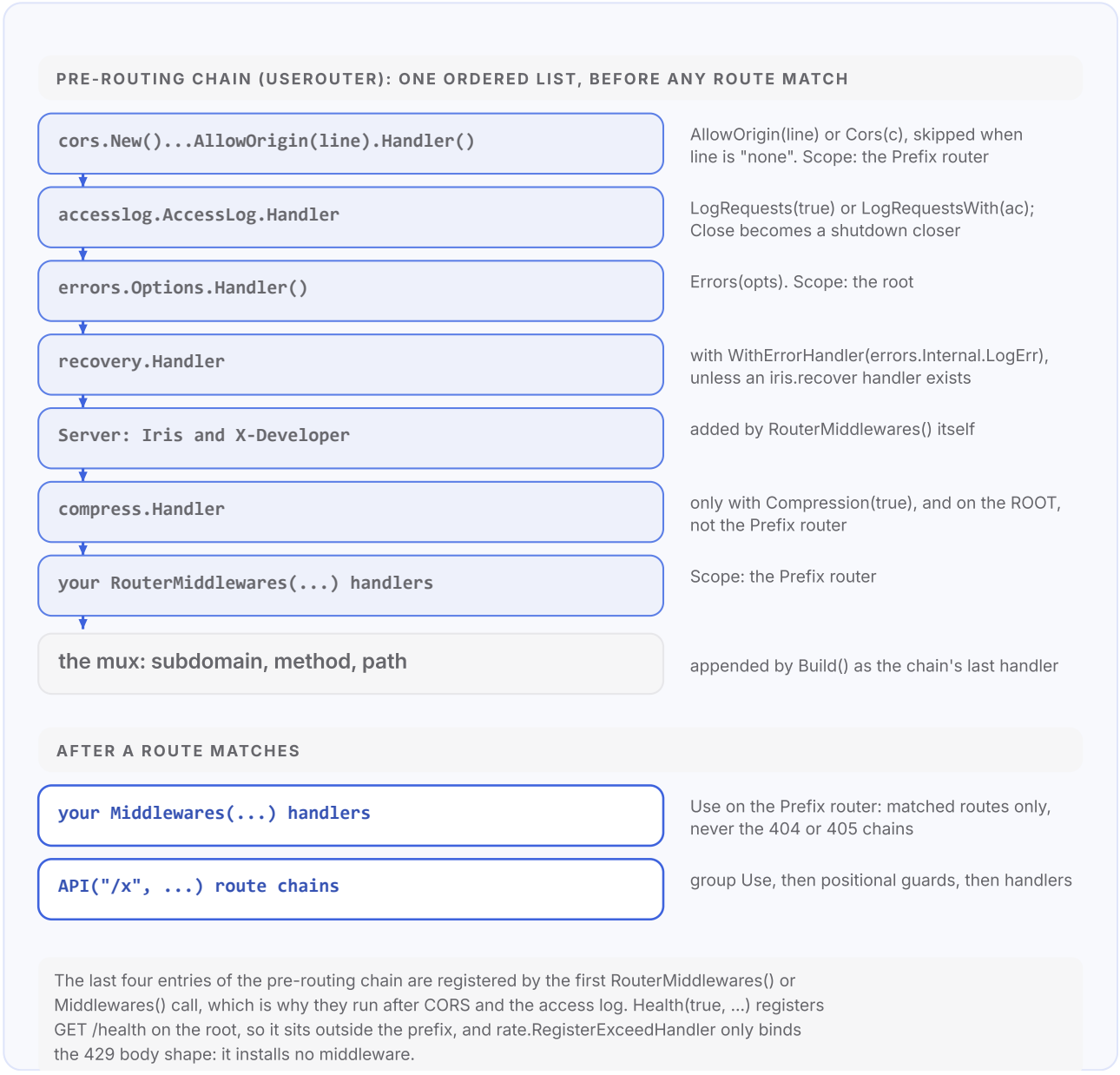
As a rule of thumb: infrastructure that must see every byte (CORS, compression, access logging, rewriting, versioning, recovery) belongs to `UseRouter`; request-scoped behavior (rate limits, body limits, sessions, authentication, cache headers) belongs to `Use` on the group that needs it.

What the Builder Installs for You

If you assemble your application with `iris.NewBuilder` (Chapter 3), several middleware from this chapter are registered by the chain itself:

- **Recovery** is installed unless one is already registered: `recovery.WithErrorHandler(errors.Internal.LogErr)` followed by `recovery.Handler`, so panics become logged 500 INTERNAL responses. The builder skips it when a handler named `iris.recovery` or `iris.errors.recover` is already on the router.
- **CORS**: the `AllowOrigin(originLine)` step registers `cors.New().HandleErrorFunc(errors.FailedPrecondition.Err). AllowOrigin(originLine).Handler()`. The `Cors(c)` step at the same position takes a CORS value you built yourself, for the settings the origin line cannot reach; it is used as given, so set `HandleErrorFunc` to keep the structured error body.
- **Compression**: `Compression(true)` only records the choice; the `compress.Handler` itself is registered later, by the first `RouterMiddlewares` or `Middlewares` call, and on the root rather than the prefix router.
- **Access logging**: `LogRequests(true)` registers a tuned `accesslog.AccessLog` (JSON lines to `access_log.json` plus stdout) and closes it on shutdown; `LogRequestsWith(ac)` registers your own instance instead.
- **Health endpoint**: `Health(true, env, developer)` registers `modrevision.New` at `GET /health` on the root, outside the prefix.
- **Rate exceed handler**: the builder binds the `rate` middleware's 429 response to the JSON error format of the `rest/errors` package.

Order matters here, and it is not the order of the list above. CORS and the access log are registered by their own builder steps, so they are already in the chain by the time the first `RouterMiddlewares` or `Middlewares` call installs the rest. That is why recovery does not wrap them, and why the access log sees the 500 that recovery produces:



The builder's middleware lands in one ordered pre-routing chain: CORS and the access log first because their steps register them directly, then the four that the first RouterMiddlewares call installs, and the mux last.

With the builder, you only reach into `middleware/` for what the chain does not cover: rate limits, sessions, JWT, body limits, cache headers and the rest of this chapter.

CORS

First, the mechanics. The browser's **same-origin policy** stops JavaScript served from `https://app.example.com` from reading responses of `https://api.example.com`, a different **origin** (scheme, host and port), unless the API opts in. CORS (Cross-Origin Resource Sharing) is that

opt-in: the server names acceptable origins in `Access-Control-*` response headers. For requests that could have side effects, the browser first sends a **preflight** (an automatic `OPTIONS` request asking "may I send a POST with these headers?") and proceeds only when the answer allows it. CORS protects browser users; it does not restrict non-browser clients such as curl, which ignore it entirely.

The `cors` middleware answers preflight requests and stamps the `Access-Control-*` response headers. It is configured through a fluent chain that ends in `.Handler()`, and it belongs to `UseRouter` so that preflights are answered even for unmatched paths.

```go
package main

import (
    "time"

    "github.com/kataras/iris/v14"
    "github.com/kataras/iris/v14/middleware/cors"
)

func main() {
    app := iris.New()

    crs := cors.New().
        // One comma-separated line; use AllowOrigins for a list.
        AllowOrigin("https://example.com,https://app.example.com").
        AllowHeaders("Content-Type", "Authorization").
        ExposeHeaders("X-Total-Count").
        MaxAge(12 * time.Hour).
        Handler()

    app.UseRouter(crs)

    app.Get("/api/data", func(ctx iris.Context) {
        ctx.JSON(iris.Map{"message": "cross-origin ready"})
    })

    app.Listen(":8080")
}
```

For dynamic origin decisions, pick one of three validators:

```go
// Match on the origin string alone.
cors.New().AllowOriginMatcherFunc(func(origin string) bool {
    return strings.HasSuffix(origin, ".example.com")
}).Handler()

// Context-aware: AllowOriginFunc(func(ctx iris.Context,
//     origin string) bool { ... })
// Regular expressions: AllowOriginRegex(`^https://.*\.example\.com$`)
```

Other chain methods: `AllowOrigins("https://a.com", "https://*.b.com")` for a list with wildcards, `DisallowCredentials()`, `ReferrerPolicy(cors.NoReferrerWhenDowngrade)`, `ExtractOriginFunc(cors.StrictOriginExtractor)`, `SkipLocalRequests(true)` and `HandleErrorFunc` to customize the rejection response. There is no `AllowMethods` method: a preflight's `Access-Control-Allow-Methods` always lists every standard method, QUERY included, and route registration controls which verbs actually exist.

Rate Limiting

Picture each client owning a bucket of tokens: every request spends one token, and the bucket refills at a steady rate up to a maximum, the burst. A client that stays under the refill rate never notices the limiter; one that surges drains its bucket and is rejected until tokens accumulate again. This "token bucket" scheme tolerates short, normal bursts of activity while capping the sustained rate.

The `rate` middleware implements a token-bucket limiter per client. `rate.Limit(limit, burst, rate.Options{})` takes events per second as a float; the `rate.Every` helper converts an interval, and the zero `rate.Options` keeps every default.

GO

```go
package main

import (
    "time"

    "github.com/kataras/iris/v14"
    "github.com/kataras/iris/v14/middleware/rate"
    "github.com/kataras/iris/v14/rest/errors"
)

func main() {
    app := iris.New()

    // 1 request per second, bursts of up to 5.
    app.Use(rate.Limit(1, 5, rate.Options{}))

    // A stricter group: 10 per minute, with periodic cleanup of
    // clients idle for more than 5 minutes and a custom response.
    strict := rate.Limit(rate.Every(time.Minute), 10, rate.Options{
        PurgeEvery:       5 * time.Second,
        PurgeMaxLifetime: 5 * time.Minute,
        ExceedHandler: func(ctx iris.Context) {
            ctx.SetHeader("Retry-After", "60")
            errors.ResourceExhausted.Message(ctx, "rate limit exceeded")
        },
    })

    auth := app.Router("/auth", strict)
    auth.Post("/login", func(ctx iris.Context) {
        ctx.JSON(iris.Map{"ok": true})
```

```
    })

    app.Listen(":8080")
}
```

By default clients are identified by remote IP and rejected with a plain

1. To key on something else (an API key, a user ID), call `rate.SetIdentifier` from a middleware registered before the limiter:

```go
app.Use(func(ctx iris.Context) {
    rate.SetIdentifier(ctx, ctx.Header("X-API-Key"))
    ctx.Next()
})
app.Use(rate.Limit(1, 5, rate.Options{}))
```

Inside handlers, `rate.Get(ctx)` returns the `*rate.Client` (its `ID`, optional `Data` filled by the `Options.ClientData` hook, and the underlying limiter) for emitting `X-RateLimit-*` headers. The limiter state is in-process memory; limits shared across replicas need a proxy or a store-backed limiter in front.

`rate.Limit` returns a handler and manages the limiter's lifetime for you. When you need the lifetime in your hands, for example to stop the purge goroutine of a limiter that lives shorter than the process, `rate.New(limit, burst, rate.Options{})` returns the `*rate.Limiter` itself: register its `Handler()` and call `Stop()` when the limiter is no longer needed.

Recovery

The `recovery` middleware turns panics into 500 responses instead of crashed connections. Register it with `UseRouter` so it wraps everything that follows.

```go
import "github.com/kataras/iris/v14/middleware/recovery"

app.UseRouter(recovery.Handler)

app.Get("/boom", func(ctx iris.Context) {
    panic("something went wrong")
})
```

To control the response, register `recovery.WithErrorHandler` before `recovery.Handler`:

```go
app.UseRouter(
    recovery.WithErrorHandler(func(ctx iris.Context, err error) {
        // Logs the panic dump, answers the canonical INTERNAL body.
        errors.Internal.LogErr(ctx, err)
```

```
    }),
    recovery.Handler,
)
```

The error passed to the handler is a `*context.ErrPanicRecovery` with `Cause`, `Stack`, `Callers`, `CurrentHandlerName`, `CurrentHandlerFileLine` and the `Request`. The `Request` dump has its credential-bearing headers replaced with `[redacted]`, the list being `recovery.RedactedRequestHeaders`, and `Error()` reports only the cause and the callers. Call `LogMessage()` when you want the dump as well, so that a handler formatting the error cannot put request headers in a response by accident. Builder-built applications already have recovery installed and wired to the central error map (`errors.Internal.LogErr`), so panics come back as JSON `INTERNAL` errors; see Chapter 9.

Compression

The `compress` middleware negotiates a content encoding from the client's `Accept-Encoding` header (gzip, deflate, brotli, snappy, s2 or zstd) and wraps the response writer. It is a single handler:

```go
import "github.com/kataras/iris/v14/middleware/compress"

app.UseRouter(compress.Handler)
```

That is the whole integration. Responses are compressed only when the client advertises support; already-encoded responses pass through. With the builder, `Compression(true)` registers exactly this handler. For manual control the package also exposes `compress.GetAcceptEncoding(r, offers...)` and writer helpers, but for nearly every application the one-liner is correct.

Access Logging

The `accesslog` middleware writes one record per request with latency, status, sizes and any custom fields. Create it once, register its `Handler` with `UseRouter`, and close it on shutdown.

```go
package main

import (
    "os"

    "github.com/kataras/iris/v14"
```

```
    "github.com/kataras/iris/v14/middleware/accesslog"
)

func main() {
    app := iris.New()

    ac := accesslog.File("access_log.json")
    defer ac.Close()

    ac.AddOutput(os.Stdout)
    ac.SetFormatter(&accesslog.JSON{Indent: "  ", HumanTime: true})
    ac.IP = true
    ac.RequestBody = true
    ac.ResponseBody = false

    ac.AddFields(func(ctx iris.Context, f *accesslog.Fields) {
        f.Set("request_id", ctx.ID())
    })

    app.UseRouter(ac.Handler)

    app.Get("/", func(ctx iris.Context) {
        ctx.JSON(iris.Map{"hello": "world"})
    })

    app.Listen(":8080")
}
```

`accesslog.New(w io.Writer)` writes to any writer, `accesslog.File(path)` opens a buffered file, and `accesslog.FileUnbuffered(path)` trades throughput for immediate writes. Formatters include `&accesslog.JSON{}`, `&accesslog.CSV{}` and `&accesslog.Template{Text: "..."}`. The last two, like the default text format, write one line per record: a line break in a value the client controls becomes a space (Chapter 17 has the details).

With the builder, `LogRequests(true)` installs a pre-tuned instance (JSON lines to `access_log.json` plus stdout, IP and body logging on) and registers its `Close` as a shutdown closer. The builder puts the access log outside compression, so that a panic is still logged, which means a compressed response reaches it as compressed bytes. It leaves the response body out of any line whose response carries a `Content-Encoding` other than `identity`, and does not call `Redact` for it; the status, the byte counts and every other field are logged as usual. A browser asks for compression on every request, so with `Compression(true)` expect response bodies in the log only for clients that did not. When the defaults do not fit, build your own and pass it to `LogRequestsWith`: same slot in the chain, your configuration:

```
ac := accesslog.File("access_log.json")
ac.SetFormatter(&accesslog.JSON{HumanTime: true})
ac.ResponseBody = false

app := iris.NewBuilder().
    Prefix("/api").
    AllowOrigin("*").
    Compression(true).
```

```
    LogRequestsWith(ac). // instead of LogRequests(true)
    Health(true, "dev", "you@example.com").
    // Errors, Services, middlewares, API — see Chapter 3.
    Build()
```

Body Limits

The `bodylimit` middleware rejects a request whose body exceeds a byte limit with a 413. A request that declares a larger `Content-Length` is refused before your handler runs. A body whose size shows only while it is read (a chunked one, or a compressed one behind the compress middleware) is stopped at the limit: the handler's read gets the bytes up to the limit and then `context.ErrRequestEntityTooLarge`, whether it reads with `ctx.Body`, `ctx.ReadJSON`, `ctx.BindJSON` or `ctx.FormFile`. Under `errors.Handler` both answer `413 TOO_LARGE` with the message "request body too large".

Forms are the exception. `ctx.ReadForm` and `ctx.Form()` report no error for a chunked form past the limit: nothing of it is bound and the handler goes on, so check for the fields you need. A chunked upload whose limit falls inside the header lines of a part answers 400 with the multipart parser's error, not 413.

GO

```
import "github.com/kataras/iris/v14/middleware/bodylimit"

// Global cap.
app.Use(bodylimit.New(2 * iris.MB))

// A larger cap only where uploads are expected.
uploads := app.Router("/uploads", bodylimit.New(100*iris.MB))
```

The `iris.KB` and `iris.MB` constants keep the sizes readable. This middleware complements (not replaces) the `ContextOptions.BodyMaxMemory` option, which caps what the body-reading helpers (`ctx.Body`, `ctx.BindJSON` and friends) will buffer; see Chapter 5.

Cache Headers

The `cache` middleware sets HTTP caching response headers (`ETag`, `Cache-Control`, `Last-Modified`, `Expires`, `Pragma`, `Vary`, `Proxy-Authenticate`) and answers conditional requests (`If-None-Match`, `If-Modified-Since`) with `304 Not Modified`. It does not store responses; it teaches clients and proxies to.

An **ETag** is a short fingerprint of a response (a hash, in essence) sent alongside the body. On the next visit the client asks "has it changed?" by echoing that fingerprint in `If-None-Match`; if the resource is unchanged the server answers `304 Not Modified`, a few bytes instead of the full body. `Cache-Control` works one step earlier: it tells the client how long it may reuse its stored copy without asking at all.

```go
package main

import (
    "time"

    "github.com/kataras/iris/v14"
    "github.com/kataras/iris/v14/middleware/cache"
)

func main() {
    app := iris.New()

    public := cache.New(cache.Options{
        ETag: true,
        CacheControl: cache.NewCacheControlBuilder().
            Public().
            MaxAge(time.Hour).      // browsers: 1 hour
            SMaxAge(2 * time.Hour). // shared caches: 2 hours
            MustRevalidate().
            Build(),
        LastModified: true,
        Expires:      1 * time.Hour,
        Vary:         cache.NewVaryBuilder().AcceptEncoding().Build(),
    })

    app.Get("/catalog", public, func(ctx iris.Context) {
        ctx.JSON(iris.Map{"items": []string{"a", "b", "c"}})
    })

    app.Listen(":8080")
}
```

`cache.DefaultOptions` is a sensible public-caching preset (`ETag`, one-hour `max-age`, two-hour `s-maxage`, 24-hour `Expires`, `Vary: Accept-Encoding`), usable as `cache.New(cache.DefaultOptions)`. The `CacheControlBuilder` also offers `Private()`, `NoCache()`, `NoStore()` and `ProxyRevalidate()`; the `VaryBuilder` offers `Accept()`, `AcceptLanguage()`, `UserAgent()` and `Add(...)`. Server-side response caching and store-backed strategies are the subject of Chapter 14.

Sessions

A **cookie** is a small piece of data the server asks the browser to store and send back with every later request, the standard way to recognize a returning visitor over stateless HTTP. A **session** builds on that: the cookie carries either the session data itself (signed and encrypted, as in the

cookie backend below) or just an ID pointing at data stored server-side. Signing makes tampering detectable; encryption keeps the contents unreadable to the client.

The `sessions` middleware provides typed, generic session data: you declare a struct, pick a backend, and get compile-time-safe reads and writes. The defaults are secure: cookies are signed and encrypted, `Secure` is on and `SameSite` is `Lax`.

GO

```go
package main

import (
    "os"
    "time"

    "github.com/kataras/iris/v14"
    "github.com/kataras/iris/v14/middleware/sessions"
)

type UserSession struct {
    ID       string
    Username string
}

func main() {
    app := iris.New()

    // 64-byte hash key, 32-byte block key: the payload is signed
    // and encrypted. Load them from configuration, never hardcode.
    opts := sessions.NewOptions(
        []byte(os.Getenv("SESSION_HASH_KEY")),
        []byte(os.Getenv("SESSION_BLOCK_KEY")),
    )
    opts.MaxAge = 1 * time.Hour // default is 24 hours
    // Defaults: Secure=true (HTTPS only), SameSite=Lax, Path="/".
    // For plain-HTTP local development only: opts.Secure = false

    db := sessions.NewCookieDatabase[UserSession](opts)

    // Loads the session per request and commits it at the end,
    // when the request changed it.
    app.Use(sessions.Handler(db, "my_session"))

    app.Post("/login", func(ctx iris.Context) {
        sess := sessions.Get[UserSession](ctx)
        sess.Set(UserSession{ID: "1", Username: "makis"})
        ctx.JSON(iris.Map{"ok": true})
    })

    app.Get("/me", func(ctx iris.Context) {
        sess := sessions.Get[UserSession](ctx)
        ctx.JSON(sess.Data)
    })

    app.Listen(":8080")
}
```

Useful session methods: `SetFlash` and the generic `GetFlash` for one-time messages; `notice, ok := sess.GetFlash[string]("notice")` returns the typed value and removes it, while a message of a different type stays in place with `(zero, false)`, so a mistyped read is never destructive (use `any` as the type argument for the old untyped behavior). `SetExtra` and `GetExtra[V]()` carry data beside the typed `Data` struct; the promoted `Extra` field remains readable directly where the raw `any` is wanted. Then `Delete(ctx)` to expire the session, `RegenerateID(ctx)` to rotate the server-side ID after sign-in, and `IsNew`. `sessions.ReadOnlyHandler` loads the session and never commits it, and `sessions.Single` binds a database to a fixed cookie name for use without the middleware.

`sessions.Handler` commits only a session the request changed. `/login` above answers with a `Set-Cookie`. `/me`, for a visitor who already has the session, sends none and writes nothing to the store. So a session ends `MaxAge` after its last change, not after its last request: here, one hour after the login. Call `sess.Commit(ctx)` in a handler that only reads when that visit should keep the session alive.

Two limits of that rule. It holds for the three databases of the package: the session of a custom `Database` is committed after every request, as before. And a response without `Set-Cookie` is one that a shared cache may store, so a page like `/me` sends `Cache-Control: private` when a cache sits in front of it (Chapter 14).

Beyond the cookie backend, `sessions.NewPostgresDatabase[T](table, conn, opts)` and `sessions.NewRedisDatabase[T](client, opts)` store the data server-side. How sessions combine with authentication flows is covered in Chapter 10.

JWT

A **JSON Web Token** is a self-contained credential: a small JSON payload (the **claims**: who the user is, when the token expires) signed by the server. The signature lets the server trust an incoming token without a database lookup, because any tampering breaks verification. The flip side is that a stolen token works until it expires, which is why lifetimes are kept short and a blocklist enables revocation.

The `jwt` middleware signs and verifies JSON Web Tokens. In v14 the verifier reads the token from the `Authorization: Bearer <token>` header only; query-string extraction is opt-in, because URLs leak into logs.

```go
package main

import (
    "os"
```

```
    "time"

    "github.com/kataras/iris/v14"
    "github.com/kataras/iris/v14/middleware/jwt"
    "github.com/kataras/iris/v14/rest/errors"
)

type UserClaims struct {
    jwt.Claims
    Username string `json:"username"`
}

func main() {
    app := iris.New()

    secret := []byte(os.Getenv("JWT_SECRET"))
    signer := jwt.NewSigner(jwt.HS256, secret, 15*time.Minute)
    verifier := jwt.NewVerifier(jwt.HS256, secret)
    verifier.WithDefaultBlocklist() // enables token revocation

    app.Post("/token", func(ctx iris.Context) {
        token, err := signer.Sign(&UserClaims{
            Claims:   jwt.Claims{Subject: "user-123"},
            Username: "makis",
        })
        if errors.HandleError(ctx, err) {
            return
        }
        ctx.OK(iris.Map{"token": string(token)}, nil)
    })

    protected := app.Router("/api",
        verifier.Verify(func() any { return new(UserClaims) }))

    protected.Get("/me", func(ctx iris.Context) {
        claims, _ := jwt.Get[*UserClaims](ctx)
        ctx.JSON(iris.Map{"username": claims.Username})
    })

    app.Listen(":8080")
}
```

`verifier.WithQueryToken()` additionally accepts `?token=...` for cases like browser file downloads that cannot set a header. Supported algorithms include `HS256`, `HS384`, `HS512`, `RS256` and `ES256`. A token whose header carries a `kid`, as the RS256 tokens of Auth0 and other identity providers do, is refused until the verifier's `HeaderValidator` field is set. Chapter 10 shows that setup, along with complete login flows, refresh tokens and the higher-level `auth` package.

Basic Authentication

HTTP Basic Authentication is the protocol's built-in login scheme: the client sends `username:password` with every request, base64-encoded but not encrypted, so it is only acceptable over HTTPS.

The `basicauth` middleware guards routes with HTTP Basic Authentication. Three constructors cover the usual cases:

```go
import "github.com/kataras/iris/v14/middleware/basicauth"

// Quick: a static user map.
guard := basicauth.Default(map[string]string{"admin": "secret"})

// Configured: realm, user source, re-ask interval. Options[U] names the
// user type the handlers read back with basicauth.GetUser[U](ctx).
guard = basicauth.New(basicauth.Options[basicauth.Map]{
    Realm:  "Admin Area",
    Allow:  basicauth.AllowUsersFile[basicauth.Map]("users.yml"),
    MaxAge: 5 * time.Minute,
})

// Shortcut: users straight from a JSON or YAML file.
guard = basicauth.Load("users.yml")

admin := app.Router("/admin", guard)
```

`Default` stores a `*iris.SimpleUser`, `Load` a `basicauth.Map`, and `basicauth.AllowUsers(list)` with a `[]YourUser` stores your own type; `ctx.User()` sees each of them as a `context.User`. Suitable for internal dashboards and tooling; for user-facing authentication prefer sessions or JWT.

API Keys

`basicauth` and `jwt` both assume a person. The `apikey` middleware covers the caller that is not one: another service, a cron job, a webhook sender or a CLI tool, presenting one long random string it was handed once.

```go
import "github.com/kataras/iris/v14/middleware/apikey"

guard := apikey.New(apikey.Options{
    Keys: map[string]string{
        "reporting": os.Getenv("REPORTING_API_KEY"),
    },
})

api := app.Router("/api", guard)
```

The key arrives as `Authorization: Bearer <key>` by default, with the scheme matched case-insensitively per RFC 7235; set `Scheme: apikey.NoScheme` for a bare value on a header such as `X-API-Key`. `Keys` maps the principal name to its secret, so one key per consumer makes a leak traceable, and `apikey.Get(ctx)` names the caller that matched. When nothing else has set a user, a `context.SimpleUser` carrying that principal is stored too, so `ctx.User()` works.

The default is to deny: no `Keys` and no `Lookup` rejects every request and logs a warning on the first one. `Hash: apikey.SHA256` lets `Keys` hold digests instead of working credentials, and `Lookup` resolves keys that live in a database, where an error is a 500 rather than a 401. `Options.Query` reads the key from a query parameter, which the access log then records with every other query argument, so mask it in a custom formatter first. Chapter 10 covers all of it.

Request IDs

The `requestid` middleware assigns each request a unique ID, exposes it through `requestid.Get(ctx)` and `ctx.ID()`, and reflects it in the `X-Request-Id` response header, the anchor for tracing a request across logs.

```go
import "github.com/kataras/iris/v14/middleware/requestid"

app.Use(requestid.New())

app.Get("/", func(ctx iris.Context) {
    ctx.JSON(iris.Map{"request_id": requestid.Get(ctx)})
})
```

The default generator honors a client-supplied `X-Request-Id` header and generates a UUIDv4 only when it is absent. That is convenient behind a trusted proxy that assigns IDs, but it lets arbitrary clients choose the value that lands in your logs. When IDs must be unforgeable, use the server-only generator:

```go
app.Use(requestid.New(requestid.ServerOnlyGenerator))
```

`requestid.HashGenerator(includeBody)` derives a fixed-length ID from the request hash instead (not unique across identical requests).

Internationalization

The `i18n` middleware loads translation files and detects the request language from a URL parameter, cookie, subdomain or the `Accept-Language` header.

```go
import "github.com/kataras/iris/v14/middleware/i18n"

locales := i18n.New()
if err := locales.Load("./locales/*/*", "en-US", "el-GR"); err != nil {
    panic(err)
}
locales.SetDefault("en-US")

app.Use(i18n.NewHandler(locales, i18n.HandlerOptions{
    URLParameter: "lang", // ?lang=el-GR
    Cookie:       "lang",
}))

app.Get("/", func(ctx iris.Context) {
    ctx.JSON(iris.Map{
        "message": i18n.GetMessage(ctx, "welcome", "Iris"),
        "locale":  i18n.GetLocale(ctx).Language(),
    })
})
```

`i18n.SetLanguage(ctx, code)` switches the language for the current visitor, and `i18n.NewRouter` serves path-prefixed locales (`/en-US/...`). Translation file formats, pluralization and template integration are covered in Chapter 15.

Views

The `view` middleware connects any template engine that implements a single-method interface:

```go
type Engine interface {
    ExecuteTemplate(w io.Writer, templateName string,
        layoutName string, data any) error
}
```

Register the engine once, render from handlers:

```go
import (
    "github.com/kataras/iris/v14/middleware/view"

    "github.com/kataras/blocks"
)

templates := blocks.New("./views")
if err := templates.Load(); err != nil {
    panic(err)
}

app.UseRouter(view.Handler(templates))
```

```
app.Get("/", func(ctx iris.Context) {
    view.Render(ctx, "index", "", iris.Map{"Title": "Home"})
})
```

Helpers on the package set per-request state: `view.Layout`, `view.Data` and `view.KeyValue`. `view.AddFunc` is the exception: it registers a template function on the engine, which every request shares, so register functions on the engine itself before `Load`. Response rendering in full (including content negotiation) is in Chapter 6.

WebSockets

A **WebSocket** is a persistent, two-way connection between browser and server that begins as an ordinary HTTP request and then upgrades. Unlike the request/response cycle, either side can push messages at any moment, which is what chat, live dashboards and multiplayer features need.

The `websocket` middleware integrates the neffos WebSocket library.

GO

```
import (
    "github.com/kataras/iris/v14/middleware/websocket"

    "github.com/kataras/neffos"
    "github.com/kataras/neffos/gorilla"
)

events := neffos.Namespaces{
    "default": neffos.Events{
        "chat": func(c *neffos.NSConn, msg neffos.Message) error {
            c.BroadcastOthers(msg)
            return nil
        },
    },
}

server := neffos.New(gorilla.DefaultUpgrader, events)
app.Any("/ws", websocket.New(server, websocket.Options{}))
```

`websocket.GetContext(conn)` recovers the originating `iris.Context` inside event callbacks, useful for reading authentication state set by earlier middleware. neffos itself (namespaces, rooms, client libraries) is a third-party library with its own documentation.

The optional neffos features work behind `websocket.New` as they do without Iris: `Conn.Close`, `Conn.Terminate(code, reason)` and `Server.Close` send their close frame, and a server built with `neffos.WithTimeout` gets its `PingInterval` heartbeat and its `MaxMessageSize` cap. Two costs

come with them: the cap counts the neffos envelope as well as the body, and an event callback that runs longer than `PingInterval` can have its connection closed by the heartbeat.

Monitoring and Profiling

Four packages cover observability basics; the full story (metrics, tracing, dashboards) is Chapter 17.

monitor renders live process and OS statistics (CPU, RAM, connections, load):

```go
import "github.com/kataras/iris/v14/middleware/monitor"

m := monitor.New(monitor.Options{ViewTitle: "My App"})
app.Get("/monitor", m.View)        // HTML dashboard
app.Get("/monitor/stats", m.Stats) // JSON
```

httpcost measures what each request costs in elapsed time, memory and CPU, using configurable per-unit prices:

```go
import "github.com/kataras/iris/v14/middleware/httpcost"

app.Use(httpcost.New(httpcost.DefaultOptions))

app.Get("/report", func(ctx iris.Context) {
    // In a later middleware or accesslog field:
    if metric, ok := httpcost.Get(ctx); ok {
        _ = metric // metric.Cost and the raw measurements
    }
    ctx.JSON(iris.Map{"ok": true})
})
```

pprof mounts the standard Go profiling endpoints as a routing controller:

```go
import "github.com/kataras/iris/v14/middleware/pprof"

app.Router("/debug/pprof", pprof.New())
// then: go tool pprof http://localhost:8080/debug/pprof/profile
```

Guard it (behind `ipaccess`, `basicauth` or a debug-only conditional) before it reaches production.

servertiming emits the `Server-Timing` response header so browser devtools show your server-side phases:

```go
import "github.com/kataras/iris/v14/middleware/servertiming"

app.Use(servertiming.Handler)
```

```
app.Get("/api/data", func(ctx iris.Context) {
    servertiming.Start(ctx, "db", "Database Query")
    // ... query ...
    servertiming.End(ctx, "db")
    ctx.JSON(iris.Map{"data": "result"})
})
```

Utility Middleware

Smaller tools, each a one- or two-line integration.

methodoverride lets HTML forms issue PUT and DELETE through POST, via headers (`X-HTTP-Method-Override` and friends) or a `_method` form field. The target may be any standard method, QUERY included:

```go
import "github.com/kataras/iris/v14/middleware/methodoverride"

app.UseRouter(methodoverride.New(methodoverride.Options{
    // Defaults: override POST only, standard headers, "_method" field.
    SaveOriginalMethodContextKey: "_originalMethod",
}))
// ctx.Method() then reflects the overridden verb.
```

rewrite applies redirect rules in `"CODE PATTERN TARGET"` form:

```go
import "github.com/kataras/iris/v14/middleware/rewrite"

engine, err := rewrite.New(rewrite.Options{
    RedirectMatch: []string{
        "301 /old-api/(.*) /api/v1/$1",
    },
    PrimarySubdomain: "www", // redirect root domain to www.
})
if err != nil {
    panic(err)
}
app.UseRouter(engine.Handler)
// Or from a YAML/JSON file: app.UseRouter(rewrite.Load("redirects.yml"))
```

ipaccess restricts by IP or CIDR, with optional exceptions:

```go
import "github.com/kataras/iris/v14/middleware/ipaccess"

guard, err := ipaccess.New(
    []string{"127.0.0.1", "10.0.0.0/8"}, // allowed
    []string{"10.0.5.9"},                // blocked exceptions
)
if err != nil {
    panic(err)
```

```
}
admin := app.Router("/admin", guard)
```

Both lists are matched against `ctx.RemoteAddr()`. The allowed patterns above are a loopback address and a private range, and those match the connection. With `RemoteAddrHeaders` set, such an address in a header is skipped as one of your own proxies, unless `RemoteAddrHeadersForce` is on or, for the private range, `RemoteAddrPrivateSubnets` is empty (Chapter 18).

versioning routes requests to versioned route trees based on a header, query parameter or subdomain:

```go
import "github.com/kataras/iris/v14/middleware/versioning"

app.UseRouter(versioning.New(versioning.Options{
    Default:    "1.0.0",
    Latest:     "2.0.0",
    Header:     "Accept-Version",
    QueryParam: "v",
    Versions: versioning.Versions{
        {Range: ">=1.0.0 <2.0.0", Path: "/v1", Deprecated: true},
        {Range: ">=2.0.0", Path: "/v2"},
    },
}))
```

Deprecated matches receive `X-Api-Warn` and deprecation headers.

modrevision serves build metadata (module version, revision, build time, environment); the builder's `Health` step uses it for `GET /health`:

```go
import "github.com/kataras/iris/v14/middleware/modrevision"

app.Get("/health", modrevision.New(modrevision.Options{
    ServerName:     "My API",
    Env:            "production",
    Developer:      "team@example.com",
    CloudDetection: true,
}))
```

counter counts requests with a generic, notification-capable counter:

```go
import "github.com/kataras/iris/v14/middleware/counter"

c := counter.NewCounter[uint64]()
app.Use(counter.RequestCounter(c))

app.Get("/stats", func(ctx iris.Context) {
    ctx.JSON(iris.Map{"total_requests": c.Get()})
})
```

referrer parses the `Referer` header into its parts, so a landing route can record where a visitor arrived from without a third-party script:

```go
import "github.com/kataras/iris/v14/middleware/referrer"

app.Use(referrer.Handler)

app.Get("/landing", func(ctx iris.Context) {
    r := referrer.Get(ctx)
    if r.Type == referrer.ReferrerSearch {
        // Label is the engine ("Google", "Bing"); Query is the search text
        // when the engine still sends it, which most no longer do.
        ctx.JSON(iris.Map{"engine": r.Label, "query": r.Query})
        return
    }

    ctx.JSON(iris.Map{"type": r.Type.String(), "domain": r.Domain})
})
```

`Type` is one of indirect, direct, email, search or social. The header is client-supplied and often absent, so treat it as a hint for analytics and never as proof of where a request came from.

Auditing Requests and Responses

`recorder` keeps a copy of the request body and of the response body, so a later handler can read either after the handler that produced it has finished. Register `recorder.Request`, `recorder.Response`, or both.

Ordinarily the request body is drained by the first reader, and the response goes straight to the socket. That is what makes an audit trail awkward to write by hand, and what this middleware solves:

```go
import "github.com/kataras/iris/v14/middleware/recorder"

// Runs after the handler: call Next first, then read what was recorded.
func auditTrail(ctx iris.Context) {
    ctx.Next()

    entry := iris.Map{
        "request_id": ctx.ID(),
        "path":       ctx.Path(),
        "request":    string(recorder.ReadRequestBody(ctx)),
    }

    if rec := recorder.GetResponseRecorder(ctx); rec != nil {
        entry["status"] = rec.Code
        entry["response"] = rec.Body.String()
    }

    auditLog.Write(entry)
```

```
}

orders := app.Router("/admin/orders",
    recorder.Request, recorder.Response, auditTrail)

orders.Post("/", createOrder)
```

`recorder.GetResponse(ctx)` returns the same thing as an `*http.Response` when that shape is more convenient. A handler that flushes while it is being recorded still sends its own status code and headers with that flush; the body is what stays recorded until the handler chain returns.

Two costs are worth stating plainly. Recording buffers the whole body in memory, so registering it globally turns every large upload and every file download into an allocation of the same size; put it on the routes that need the trail and nowhere else. And the recorded body is the real one, passwords and card numbers included, so scrub it before it reaches a log the way `accesslog`'s `Redact` does.

Geolocation

`geolocation` resolves the client IP to a country, region, city and timezone. The common production use is picking a sensible default before the visitor has chosen anything: a currency, a language, a nearest region.

```go
import (
    stderrors "errors" // "errors" here is the framework's rest/errors.

    "github.com/kataras/iris/v14/middleware/geolocation"
    "github.com/kataras/iris/v14/rest/errors"
)

app.Use(geolocation.New(geolocation.Options{
    APIKey:          os.Getenv("IP_API_KEY"),
    Cache:           geolocation.NewMemoryCache(10_000),
    DisableLanguage: true, // skip the second lookup when you do not need it.
}))

app.Get("/pricing", func(ctx iris.Context) {
    info, err := geolocation.Fetch(ctx)
    if err != nil {
        if stderrors.Is(err, geolocation.Err) {
            ctx.OK(iris.Map{"currency": "USD"}, nil) // fall back, do not fail.
            return
        }

        errors.HandleError(ctx, err)
        return
    }

    ctx.OK(iris.Map{"country": info.CountryCode, "city": info.City}, nil)
})
```

`geolocation.Get(ctx)` returns the already-resolved `*Info` without fetching, for handlers that run after the middleware.

The lookup is a network call to an external service, so two rules follow. Configure the cache, or a burst of traffic from one office becomes a burst of identical API calls. And write the fallback branch first, because the service will be slow or down at some point and a pricing page that fails closed on a geo lookup is worse than one that shows dollars.

Geolocation is a guess about a network address, not a fact about a person. It answers “which currency should this page default to”, never “is this request allowed”. A VPN defeats it in one click.

Captcha Verification

`hcaptcha` and `recaptcha` verify a challenge token with the provider before the handler runs. Register either as a route middleware on the endpoint that accepts the form:

```go
import "github.com/kataras/iris/v14/middleware/hcaptcha"

app.Post("/contact",
```

```
    hcaptcha.New(hcaptcha.Options{Secret: os.Getenv("HCAPTCHA_SECRET")}),
    func(ctx iris.Context) {
        // Only reached when the challenge passed.
        res, _ := hcaptcha.Get(ctx)
        ctx.Application().Logger().Infof("solved for %s", res.Hostname)

        submitTicket(ctx)
    })
```

`recaptcha` works the same way. Both take an `Options` value whose `FailureHandler` field replaces the default rejection response, a 429 with no body; `recaptcha.DefaultFailureHandler` and `hcaptcha.DefaultFailureHandler` are the ones it replaces:

```go
import (
    "github.com/kataras/iris/v14/middleware/recaptcha"
    "github.com/kataras/iris/v14/rest/errors"
)

app.Post("/signup", recaptcha.New(recaptcha.Options{
    Secret: os.Getenv("RECAPTCHA_SECRET"),
    FailureHandler: func(ctx iris.Context) {
        errors.PermissionDenied.Message(ctx, "captcha verification failed")
    },
}), signup)
```

Both packages can render the client-side form for you during development: `hcaptcha.RenderForm(ctx, siteKey, "/contact")` and `recaptcha.GetFormHTML(siteKey, "/signup")`. Real applications usually render their own markup and keep only the verification here.

The secret is a server-side credential and belongs in the environment, not in the source or in the template beside the site key. Verification adds a round trip to the provider on every submission, bounded by `hcaptcha.VerifyTimeout` (10 seconds by default), so a captcha in front of a hot path costs latency on every request that reaches it.

Serving gRPC on the Same Port

`grpc` lets one Iris application answer both gRPC and ordinary HTTP on a single port, which matters when you have exactly one TLS port to spend:

```go
import (
    grpcWrapper "github.com/kataras/iris/v14/middleware/grpc"
    "google.golang.org/grpc"
)

grpcServer := grpc.NewServer()
pb.RegisterGreeterServer(grpcServer, &greeterService{})

app := iris.New()
```

```
app.UseRouter(grpcWrapper.New(grpcServer))

// Ordinary routes are unaffected; gRPC traffic never reaches them.
app.Get("/health", healthHandler)

app.Run(iris.ServerOptions{
    Addr:     ":443",
    CertFile: "cert.pem",
    KeyFile:  "key.pem",
})
```

The wrapper inspects each request and forwards it to the gRPC server only when it is a gRPC call, which it decides from HTTP/2 plus the `application/grpc` content type. Everything else calls `ctx.Next()` and continues to the router.

That check is also the constraint: gRPC requires HTTP/2, and Go's HTTP/2 support is only enabled over TLS. Serve this over plaintext and the gRPC half silently never matches, because the requests arrive as HTTP/1.1. Use `Run` with a certificate, as above, rather than `Listen`.

Adapting net/http Middleware

Anything written for `net/http` converts with `iris.FromStd`:

GO

```
// Supported forms:
//   iris.FromStd(h http.Handler)
//   iris.FromStd(func(w http.ResponseWriter, r *http.Request))
//   iris.FromStd(func(w http.ResponseWriter, r *http.Request,
//       next http.HandlerFunc))
app.Use(iris.FromStd(stdMiddleware))
```

The converted handler sees the route the way a standard one expects. In all three forms `r.PathValue("id")` returns the route parameter, and `r.Pattern` holds the matched route's template as you registered it, for example `/users/{id:uuid}`, so standard middleware that labels metrics or traces by route works unchanged. Register that middleware with `Use`, not `UseRouter`: the pre-routing slot runs before a route is matched, so there `r.Pattern` is empty, as it is in the built-in 404 and 405 handlers.

And any handler chain can run conditionally with `iris.NewConditionalHandler`:

GO

```
app.Use(iris.NewConditionalHandler(func(ctx iris.Context) bool {
    return ctx.Header("X-Debug") == "1"
}, debugHeadersMiddleware))
```

Community Middleware

Thirteen more middleware live outside the framework, in iris-contrib/middleware. They cover the integrations that would drag a vendor SDK into the framework's dependency tree if they shipped here:

Middleware	What it adds
`casbin`	Authorization with ACL, RBAC and ABAC models
`cloudwatch`	AWS CloudWatch metrics
`csrf`	Cross-site request forgery tokens
`expmetric`	Request counters exposed through `expvar`
`newrelic`	The official New Relic Go agent
`pg`	PostgreSQL integration
`prometheus`	A Prometheus metrics endpoint
`secure`	A set of common security headers
`throttler`, `tollboothic`	Two other rate-limiting approaches
`zap`	Request logging through `go.uber.org/zap`

Each is its own Go module, so you install only the one you want and the rest of its dependency tree stays out of your build:

```sh
$ go get github.com/iris-contrib/middleware/casbin@master
```

Sentry maintains its own Iris handler in getsentry/sentry-go rather than here.

Two entries in that repository, `cors` and `jwt`, predate the framework's own. Use the built-in `middleware/cors` and `middleware/jwt` instead: they are maintained alongside the framework and the builder wires both for you.

The `main` branch of that repository targets v14. The v12 line is kept unchanged at its `v12` tag, so an application that has not migrated yet still has something to pin to.

These modules are maintained by their contributors rather than with the framework, so they move on their own schedule. Check when one was last touched before you build something load-bearing on it.

Summary

- Thirty-one middleware packages live under `github.com/kataras/iris/v14/middleware/`; each is a plain handler or a small constructor that returns one.
- Registration level matters: `UseRouter` for infrastructure that must see every request (CORS, compression, access logs, rewrites, recovery), `Use` for request-scoped behavior (rate and body limits, sessions, authentication, cache headers).
- `iris.NewBuilder` already installs recovery, and (through its chain steps) CORS, compression, access logging and the `/health` endpoint, all wired to the `rest/errors` JSON error format.
- The heavy hitters each take a handful of lines: `cors.New()... .Handler()`, `rate.Limit(limit, burst, rate.Options{})`, `compress.Handler`, `accesslog.File(path)`, `bodylimit.New(n)`, `cache.New(cache.Options{...})` with its `CacheControlBuilder`, typed `sessions` with secure cookie defaults, a `jwt` verifier that reads the `Authorization` header by default, and `apikey.New(apikey.Options{Keys: ...})` for the caller that is a program rather than a person.
- `requestid.ServerOnlyGenerator` prevents clients from forging the IDs that reach your logs; the default generator trusts an inbound `X-Request-Id`.
- The deeper treatments live in their own chapters: authentication in Chapter 10, caching in Chapter 14, i18n in Chapter 15, observability in Chapter 17.

Further Reading

- MDN: Cross-Origin Resource Sharing (CORS): the same-origin policy, preflights and every `Access-Control-*` header explained.
- MDN: HTTP caching: how `Cache-Control`, `ETag` and conditional requests interact.
- MDN: Using HTTP cookies: cookie attributes (`Secure`, `SameSite`, expiry) that session security relies on.
- RFC 7519: JSON Web Token. The JWT specification: claims, structure and validation rules.
- OWASP Session Management Cheat Sheet: session ID rotation, expiry and cookie hardening in practice.
- net/http/pprof package documentation: the Go profiling endpoints the `pprof` middleware mounts.

CHAPTER 13

Database Integration

Iris is database-agnostic by design: it ships no ORM, no drivers, and no storage adapters. What it does ship is everything a database-backed application needs around the driver: dependency injection to construct and share connections, lifecycle hooks for startup checks and graceful shutdown, request contexts that cancel queries when clients disconnect, and a central error map that turns driver errors into correct HTTP responses. This chapter teaches that pattern once, shows it end to end with PostgreSQL, and tells you how to apply it to any other store.

Background

A **database** is a program whose entire job is storing and retrieving data reliably. Relational databases (PostgreSQL, MySQL, SQLite) organize data into **tables**: grids where each **row** is one record (one product, one user) and each column one attribute. Document databases (MongoDB) store **documents** instead, JSON-like records that need not share a fixed shape. You could, in principle, persist data by writing files yourself, until two requests write at the same moment, the process crashes mid-write, or you need "all products cheaper than X" without reading every file. Databases exist because they solve exactly those problems: concurrent access, crash safety and fast lookup over large data sets.

Four terms carry this chapter:

- A **query** is a request you send to the database ("give me the product with this ID", "insert this row"), usually written in SQL, the standard language of relational databases.
- A **connection** is the channel a query travels over. Opening one is comparatively expensive, so applications keep a **connection pool**: a small set of open connections that requests borrow and return, instead of every request dialing the database from scratch.
- A **transaction** groups several statements into an all-or-nothing unit: either every step commits, or the database rolls them all back as if none had happened. Money leaving one

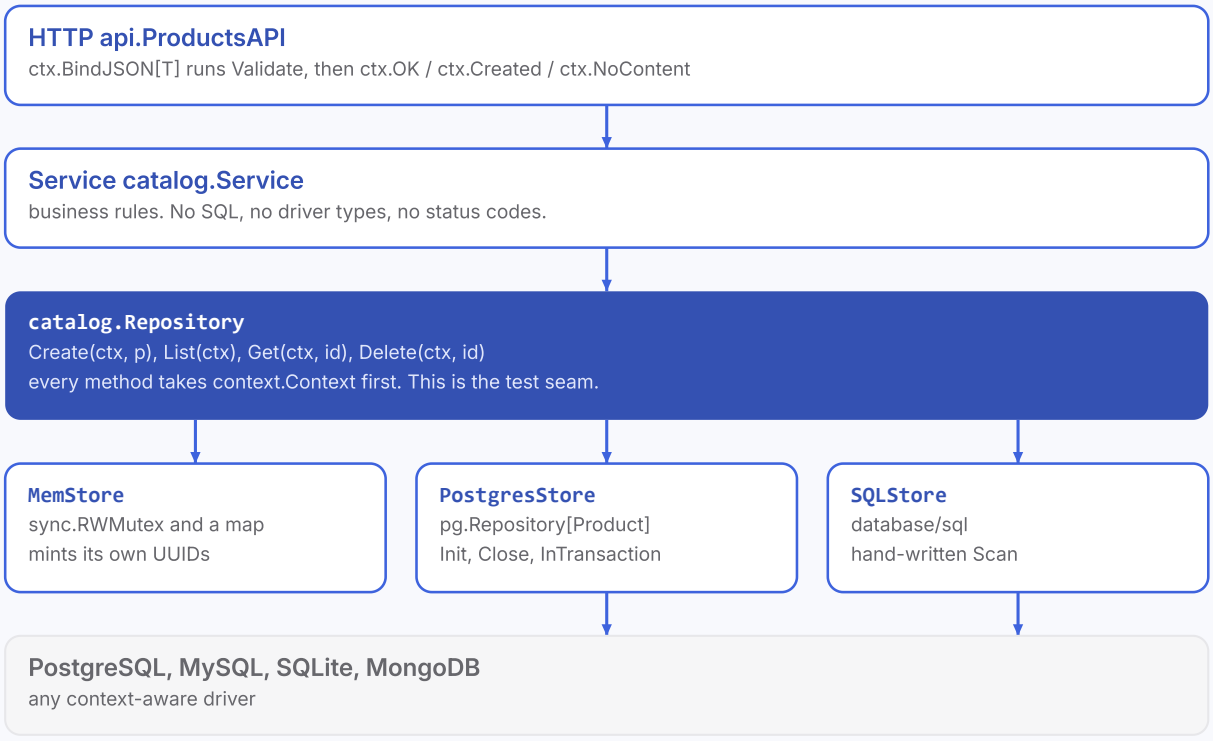
account must arrive in another; a transaction is how the database promises there is no in-between state.

- An **ORM-style layer** (object-relational mapper) translates between your language's structs and the database's rows so you do not hand-write the copying in either direction. In this chapter that role is played by `pg.Repository[T]`, which scans rows straight into your `Product` struct.

Everything in this chapter flows through one Go **interface**: a named set of method signatures that any type can satisfy. Handlers and services depend on that interface, never on a concrete driver, which is what buys the chapter's repeated promise: the real database, an in-memory fake for tests and any future replacement are interchangeable implementations behind the same four methods. What disappears from your code is storage detail spread across handlers (connection handles, driver error checks, SQL strings next to HTTP status codes), leaving one file per database that you can swap with one line of wiring.

The Repository Seam

Handlers never talk to a database. They call a service; the service talks to storage through an interface. That interface, the repository seam, is the single point where the application meets persistence, and it is what makes the rest of this chapter mechanical: to change databases you write one new type and change one line of wiring.



ON THE WAY BACK

A store is the only layer that knows a driver error, and it translates on the way out: pg.ErrNoRows and sql.ErrNoRows become ErrNotFound, pg.IsErrDuplicate becomes ErrSKUTaken. Above the seam there are domain sentinels and nothing else.

One map turns those sentinels into status codes: Errors(errors.NewOptions().MapErrors(NotFound, ErrNotFound).MapErrors(AlreadyExists, ErrSKUTaken)) gives 404 and 409. sql.ErrNoRows is already 404; importing rest/errors/pgerrors adds the pg equivalents. Anything unmapped is a logged 500, and the schema never reaches the wire.

The repository interface is the seam where the application meets persistence: services speak domain sentinels, only a store knows a driver error, and one error map turns those sentinels into status codes.

The running example is a small product catalog. First the domain:

GO

```go
// catalog/catalog.go
package catalog

import (
    "errors"
    "time"

    "github.com/kataras/iris/v14/rest/errors/validation"
)

// Sentinel errors: the domain speaks its own language. The mapping to
// HTTP status codes happens once, in main.go (see the Errors section).
var (
    ErrNotFound = errors.New("product not found")
    ErrSKUTaken = errors.New("sku already exists")
)
```

```
// Product is the stored entity. The pg struct tags describe the
// PostgreSQL columns for the store you will meet later in this chapter;
// every other store (and the JSON encoder) ignores them.
type Product struct {
    ID        string    `pg:"type=uuid,primary,default=gen_random_uuid()" json:"id"`
    CreatedAt time.Time `pg:"type=timestamp,default=clock_timestamp()" json:"created_at"`
    Name      string    `pg:"type=varchar(255)" json:"name"`
    SKU       string    `pg:"type=varchar(64),unique" json:"sku"`
    Price     int64     `pg:"type=bigint" json:"price"` // Cents.
}

// ProductInput is the request payload. ctx.BindJSON runs Validate
// automatically after decoding, so handlers never call it by hand.
type ProductInput struct {
    Name  string `json:"name"`
    SKU   string `json:"sku"`
    Price int64  `json:"price"`
}

func (in ProductInput) Validate() error {
    return validation.Join(
        validation.String("name", in.Name).NotEmpty().Length(1, 255),
        validation.String("sku", in.SKU).NotEmpty().Length(1, 64),
        validation.Number("price", in.Price).Positive(),
    )
}
```

If you dislike driver tags on a domain struct, define a separate storage entity inside the store implementation and map between the two. For most applications the single struct is the right trade: one type, one JSON shape, one schema definition.

The seam itself:

GO

```
// catalog/repository.go
package catalog

import "context"

// Repository is the storage seam: the service depends on this
// interface, production wires a database implementation, tests wire
// MemStore or a one-method stub. Every method takes a context.Context
// first — iris.Context implements it, so request cancellation reaches
// the database driver with zero extra code.
type Repository interface {
    Create(ctx context.Context, p Product) (id string, err error) // ErrSKUTaken
    List(ctx context.Context) ([]Product, error)
    Get(ctx context.Context, id string) (Product, error) // ErrNotFound
    Delete(ctx context.Context, id string) (bool, error)
}
```

Two details are load-bearing. The first parameter of every method is a plain `context.Context`: handlers pass their `iris.Context` straight through, and when a client disconnects, the cancellation propagates into `pgx`, `database/sql`, the MongoDB driver (any context-aware driver) and aborts the query. (When a write must outlive the request, an audit insert after the

response for example, pass `ctx.WithoutCancel()` instead: it detaches from cancellation but keeps request values.) The second detail is that ID generation lives inside the store, not the service: an in-memory store mints UUIDs itself, PostgreSQL uses a `gen_random_uuid()` column default, and the seam hides the difference by returning the new id.

The in-memory implementation is ordinary Go: a mutex and a map. Iris serves each request on its own **goroutine** (Go's lightweight thread), so two requests can touch this map at the same moment; the `sync.RWMutex` is the guard that prevents corruption, giving writers exclusive access while readers share it. This store is the development store and, later, the test suite's fake:

GO

```go
// catalog/memstore.go
package catalog

import (
    "context"
    "sort"
    "sync"
    "time"
    "uuid"
)

type MemStore struct {
    mu       sync.RWMutex
    products map[string]Product
}

var _ Repository = (*MemStore)(nil)

func NewMemStore() *MemStore {
    return &MemStore{products: make(map[string]Product)}
}

func (s *MemStore) Create(_ context.Context, p Product) (string, error) {
    s.mu.Lock()
    defer s.mu.Unlock()

    for _, existing := range s.products {
        if existing.SKU == p.SKU {
            return "", ErrSKUTaken
        }
    }

    p.ID = uuid.New().String()
    p.CreatedAt = time.Now().UTC()
    s.products[p.ID] = p
    return p.ID, nil
}

func (s *MemStore) List(_ context.Context) ([]Product, error) {
    s.mu.RLock()
    defer s.mu.RUnlock()

    list := make([]Product, 0, len(s.products))
    for _, p := range s.products {
        list = append(list, p)
    }
    sort.Slice(list, func(i, j int) bool {
```

```
        return list[i].CreatedAt.Before(list[j].CreatedAt)
    })
    return list, nil
}

func (s *MemStore) Get(_ context.Context, id string) (Product, error) {
    s.mu.RLock()
    defer s.mu.RUnlock()

    p, ok := s.products[id]
    if !ok {
        return Product{}, ErrNotFound
    }
    return p, nil
}

func (s *MemStore) Delete(_ context.Context, id string) (bool, error) {
    s.mu.Lock()
    defer s.mu.Unlock()

    if _, ok := s.products[id]; !ok {
        return false, nil
    }
    delete(s.products, id)
    return true, nil
}
```

Business rules live in a service that depends on the seam. Here it is thin; in a real application this is where invariants, authorization checks and cross-entity logic accumulate (see Chapter 8 for the layering rationale):

```go
// catalog/service.go
package catalog

import "context"

// Service holds the business rules. Its constructor declares what it
// needs; the iris dependency container calls it with the registered
// Repository.
type Service struct {
    repo Repository
}

func NewService(repo Repository) *Service {
    return &Service{repo: repo}
}

func (s *Service) Create(ctx context.Context, in ProductInput) (string, error) {
    return s.repo.Create(ctx, Product{
        Name:  in.Name,
        SKU:   in.SKU,
        Price: in.Price,
    })
}

func (s *Service) List(ctx context.Context) ([]Product, error) {
    return s.repo.List(ctx)
}
```

```
func (s *Service) Get(ctx context.Context, id string) (Product, error) {
    return s.repo.Get(ctx, id)
}

func (s *Service) Delete(ctx context.Context, id string) (bool, error) {
    return s.repo.Delete(ctx, id)
}
```

Wiring a Store Through Services

The HTTP layer stays thin: read input, call the service, hand the result and the error to a response helper. Note that every service call receives `ctx` (the `iris.Context`) as its `context.Context`:

GO

```
// api/products.go
package api

import (
    "github.com/kataras/iris/v14"
    "shop/catalog"
)

type ProductsAPI struct {
    svc *catalog.Service
}

func NewProductsAPI(svc *catalog.Service) *ProductsAPI {
    return &ProductsAPI{svc: svc}
}

func (api *ProductsAPI) Routing(r iris.Router) {
    r.Get("/", api.list)
    r.Post("/", api.create)
    r.Get("/{id:uuid}", api.get)
    r.Delete("/{id:uuid}", api.delete)
}

func (api *ProductsAPI) list(ctx iris.Context) {
    products, err := api.svc.List(ctx)
    ctx.List(products, err)
}

func (api *ProductsAPI) create(ctx iris.Context) {
    in, ok := ctx.BindJSON[catalog.ProductInput]()
    if !ok {
        return // 400 already written: malformed JSON or failed Validate.
    }

    id, err := api.svc.Create(ctx, in)
    ctx.Created(id, err) // 201 with {"id": "..."}.
}

func (api *ProductsAPI) get(ctx iris.Context) {
    p, err := api.svc.Get(ctx, ctx.Params().GetString("id"))
    ctx.OK(p, err) // ErrNotFound is mapped to 404 centrally.
```

```
}

func (api *ProductsAPI) delete(ctx iris.Context) {
    deleted, err := api.svc.Delete(ctx, ctx.Params().GetString("id"))
    ctx.NoContent(deleted, err) // 204 deleted, 304 not found.
}
```

These four handlers exist because a `catalog.Service` sits between them and the store, and that service owns the SKU invariant. When there is no such rule and the handlers would only forward to the repository, do not write them: `mvc.UseRepository(repo)` registers the same resource from the repository alone, with `mvc.Guard` for the write routes. See Chapter 24.

Everything assembles in the builder. The repository arrives as a parameter so `main` can pass the real store while tests pass fakes:

```go
// main.go
package main

import (
    "github.com/kataras/iris/v14"
    "github.com/kataras/iris/v14/rest/errors"

    "shop/api"
    "shop/catalog"
)

func newApp(repo catalog.Repository) *iris.Application {
    return iris.NewBuilder(iris.DefaultOptions).
        Prefix("/api").
        AllowOrigin("none").
        Compression(true).
        LogRequests(false).
        Health(true, "development", "shop").
        Errors(errors.NewOptions().
            MapErrors(errors.NotFound, catalog.ErrNotFound).
            MapErrors(errors.AlreadyExists, catalog.ErrSKUTaken)).
        Services(repo, catalog.NewService).
        RouterMiddlewares().
        Middlewares().
        API("/products", api.NewProductsAPI).
        Build()
}

func main() {
    app := newApp(catalog.NewMemStore())
    app.Listen(":8080")
}
```

`Services` does more than store values for injection: it routes them by shape, and two of those rules exist precisely for database stores:

- A registered value (or constructor output) with a `Close()` or `Close() error` method becomes a **shutdown closer** automatically: it runs when the application shuts down gracefully, after in-

flight requests drain. No manual `defer db.Close()` bookkeeping.

- A service implementing `iris.Initializer` (`Init(ctx context.Context) error`) runs exactly once inside `Build()`, after dependency injection and before the server starts. This is the place for schema creation, a connectivity ping or cache warm-up; a returned error aborts startup instead of failing on the first request. It is the startup counterpart of `Deferrables`.

The next section uses both.

PostgreSQL with kataras/pg

Two more terms before the code. A database **schema** is the set of table definitions (names, column types, rules) that must exist before the first query can run. A **constraint** is such a rule enforced by the database itself: the `unique` tag on `SKU` means the database refuses a second row with the same SKU, no matter which program inserts it, and reports the attempt as a duplicate-key error.

For PostgreSQL, `github.com/kataras/pg` is a separate module by the Iris author, the same library Iris' own `auth/identity` sub-package uses internally. It sits on top of the pgx pool and derives the schema from the `pg` struct tags you saw on `Product`: `pg.NewSchema()` registers entities, `DB.CreateSchema` issues the corresponding `CREATE TABLE IF NOT EXISTS` statements, and the generic `pg.Repository[T]` scans rows into `T` without hand-written `Scan` calls.

```sh
go get github.com/kataras/pg@latest
```

The store implements the same `Repository` seam. It also implements `Init` and `Close`, so registering it wires schema creation into `Build()` and pool shutdown into graceful termination:

```go
// catalog/pgstore.go
package catalog

import (
    "context"
    "errors"

    "github.com/kataras/pg"
)

type PostgresStore struct {
    products *pg.Repository[Product]
}

var _ Repository = (*PostgresStore)(nil)

func NewPostgresStore(db *pg.DB) *PostgresStore {
    return &PostgresStore{products: pg.NewRepository[Product](db)}
```

```
}

// Init implements iris.Initializer: it runs once inside Build(), after
// injection, and a non-nil error aborts startup.
func (s *PostgresStore) Init(ctx context.Context) error {
    return s.products.DB().CreateSchema(ctx)
}

// Close is picked up by Services automatically and runs on graceful
// shutdown, closing the underlying pgx pool.
func (s *PostgresStore) Close() {
    s.products.DB().Close()
}

func (s *PostgresStore) Create(ctx context.Context, p Product) (string, error) {
    var id string
    if err := s.products.InsertSingle(ctx, p, &id); err != nil {
        if _, ok := pg.IsErrDuplicate(err); ok {
            return "", ErrSKUTaken // The unique "sku" constraint.
        }
        return "", err
    }
    return id, nil
}

func (s *PostgresStore) List(ctx context.Context) ([]Product, error) {
    return s.products.Select(ctx,
        `SELECT * FROM products ORDER BY created_at`)
}

func (s *PostgresStore) Get(ctx context.Context, id string) (Product, error) {
    p, err := s.products.SelectByID(ctx, id)
    if err != nil {
        if errors.Is(err, pg.ErrNoRows) {
            return Product{}, ErrNotFound
        }
        return Product{}, err
    }
    return p, nil
}

func (s *PostgresStore) Delete(ctx context.Context, id string) (bool, error) {
    return s.products.DeleteByID(ctx, id)
}
```

Notice the error translation at the boundary: `pg.ErrNoRows` becomes the domain's `ErrNotFound` and a duplicate-key violation (detected with `pg.IsErrDuplicate`) becomes `ErrSKUTaken`. The service and handlers never see driver errors. The central map in `newApp` already turns those domain errors into 404 and 409.

Swapping the database is now the promised one-line change in `main`:

```
    "os"

    "github.com/kataras/pg"

    // Optional safety net: maps any pg driver error that escapes the
    // store untranslated (pg.ErrNoRows -> 404, duplicate key -> 409,
    // invalid input syntax -> 400). See the error mapping section.
    _ "github.com/kataras/iris/v14/rest/errors/pgerrors"

    "shop/catalog"
)

func main() {
    schema := pg.NewSchema()
    schema.MustRegister("products", catalog.Product{})

    db, err := pg.Open(context.Background(), schema,
        os.Getenv("DATABASE_URL"))
    if err != nil {
        log.Fatal(err)
    }

    app := newApp(catalog.NewPostgresStore(db))
    app.Listen(":8080")
}
```

`newApp` is unchanged. A working `DATABASE_URL` looks like:

```sh
DATABASE_URL="postgres://postgres:secret@localhost:5432/shop?sslmode=disable"
```

Pool sizing travels in the same connection string through the standard pgx parameters (`pool_max_conns`, `pool_min_conns`, `pool_max_conn_lifetime`, `pool_max_conn_idle_time`, `pool_health_check_period`), so operations can tune the pool per environment without a code change. Never hard-code the connection string: it is a credential, and the environment (or a secret manager feeding it) is its only home (Chapter 11).

Beyond what the store above uses, `pg.Repository[T]` covers the rest of day-to-day SQL with the same shape (`Insert`, `Update`, `UpdateOnlyColumns`, `Upsert`, `Exists`, `SelectSingle` and raw `Query` / `QueryRow` escapes) plus `InTransaction`, which the transactions section below builds on.

The Standard Library: database/sql

If you prefer to stay on the standard library (or you target MySQL, SQLite or SQL Server), the seam pattern is identical; you just write the scanning yourself. `database/sql` is stdlib; only the driver you register (pgx's stdlib adapter, `go-sql-driver/mysql`, `modernc.org/sqlite`, ...) is a third-party choice.

```go
// catalog/sqlstore.go
package catalog

import (
    "context"
    "database/sql"
    "errors"
)

type SQLStore struct {
    db *sql.DB
}

func NewSQLStore(db *sql.DB) *SQLStore {
    return &SQLStore{db: db}
}

func (s *SQLStore) Get(ctx context.Context, id string) (Product, error) {
    var p Product
    err := s.db.QueryRowContext(ctx,
        `SELECT id, created_at, name, sku, price
         FROM products WHERE id = $1`, id).
        Scan(&p.ID, &p.CreatedAt, &p.Name, &p.SKU, &p.Price)
    if errors.Is(err, sql.ErrNoRows) {
        return Product{}, ErrNotFound
    }
    return p, err
}
```

The remaining methods follow the same shape: `ExecContext` for writes, `QueryContext` plus a scan loop for lists, always parameterized queries (`$1`, `?`), never string concatenation, and always the `...Context` variants so request cancellation reaches the database.

The parameterization rule guards against **SQL injection**, the classic attack in which user input pasted into a query string becomes part of the query itself: a "name" like `'; DROP TABLE products;--` rewrites your statement. Placeholders send the query and its values to the database separately, so input can never change the query's structure.

`*sql.DB` is itself a connection pool. Configure it once, right after `sql.Open`, and register the store the same way as before (give it `Close` and, if you want a startup ping, an `Init` that calls `PingContext`):

```go
db.SetMaxOpenConns(25)
db.SetMaxIdleConns(25)
db.SetConnMaxLifetime(30 * time.Minute)
db.SetConnMaxIdleTime(5 * time.Minute)
```

The defaults (unlimited open connections, two idle) are rarely what a production service wants, but the right numbers depend on your database and workload. Measure before you tune.

One convenience specific to `database/sql`: the central error map already knows `sql.ErrNoRows` and renders it as 404 by default, so even an untranslated leak produces a correct response instead of a 500.

MongoDB, Redis and ORMs

The seam does not care what is behind it, so third-party stores follow the same recipe. Consult each library's own documentation for its API; this book does not duplicate it.

MongoDB. Implement `Repository` with the official Go driver (`go.mongodb.org/mongo-driver`), holding a collection handle in the store. Pass the request context into every driver call and translate the driver's "no documents" sentinel into `ErrNotFound` at the store boundary, exactly as the PostgreSQL store translates `pg.ErrNoRows`.

Redis. Redis is usually not the repository itself but a layer beside it: caching, rate limiting, session-like state. Keep the client (`github.com/redis/go-redis/v9` is the common choice) registered in `Services` so its lifecycle is managed, and see Chapter 14 for the cache-aside pattern that wraps a `Repository` without the service ever knowing.

ORMs and generators. GORM, ent and sqlc all work with Iris for the same reason everything else does: they live entirely inside your store implementation. Map the library's record-not-found error into your domain sentinel at the boundary, keep models out of your handlers, and register the handle through `Services`. If an ORM error can escape to a handler, add it to the central map with `MapErrors` alongside your domain errors.

Column Types That Survive JSON

A model crosses two boundaries with different vocabularies. SQL has `NULL`; JSON does not have `NaN`. `rest/types` holds the small scalar types that sit on both sides of that gap, each implementing `database/sql`'s `Scanner` and `Valuer` as well as the JSON encoders, so one field serves the column and the wire.

Two of them are worth naming here because they solve failures that only show up in production data.

`types.Number` is a `float64` with a JSON null. `encoding/json` refuses to encode `NaN` or an infinity, because JSON has no literal for them, and a computed average over an empty set produces one often enough that a single row takes the whole document down with it. `Number`

marshals all three as `null`, and a JSON `null` unmarshals back to `NaN`:

```go
type Reading struct {
    Sensor  string       `json:"sensor"`
    Average types.Number `json:"average"` // null when there is nothing to average.
}
```

The same rule holds on the database side: a `NULL` column scans to `NaN` and a `NaN` is written as `NULL`, so a nullable `float8` needs no `sql.NullFloat64` wrapper. The zero value is the number zero and not null; `types.NullNumber()` returns the null one and `IsNull()` reports it.

`types.Weekday` is a `time.Weekday` that travels as a lowercase name:

```go
type Shift struct {
    Day types.Weekday `json:"day"` // "monday", not 1.
}
```

A client reading a schedule gets a word it can print instead of a number whose base it has to guess, and the JSON stays readable in a log. The column is text. `types.ParseWeekday` accepts any case and the three-letter form (`"Monday"`, `"monday"`, `"MON"`, `"mon"`), the seven days are exported as constants, and the two conversions are `types.WeekdayOf` going in and `d.Std()` coming out. The numbering is `time.Weekday`'s, Sunday being zero, which means the zero value marshals as `"sunday"` rather than as an error: a schedule struct nobody filled in claims Sunday, so set the field deliberately.

Transactions Below the Handler

Transactions belong in the store (or, for multi-repository coordination, the service), never in handlers. A handler that begins a transaction couples HTTP concerns to storage concerns, risks holding a database transaction open across response rendering and slow clients, and bypasses the guarantee that every unit of work is atomic no matter who calls it. Express each atomic unit as one method on the layer that owns it, and let the error it returns flow through the usual pipeline.

With `kataras/pg`, `InTransaction` wraps a function in begin/commit/rollback: return nil to commit, return an error to roll back. The callback receives a transactional repository, so the body reads exactly like non-transactional code:

```go
// Reprice atomically loads a product and writes the new price.
func (s *PostgresStore) Reprice(ctx context.Context, id string,
    price int64) error {
    return s.products.InTransaction(ctx,
```

```
        func(tx *pg.Repository[Product]) error {
            p, err := tx.SelectByID(ctx, id)
            if err != nil {
                if errors.Is(err, pg.ErrNoRows) {
                    return ErrNotFound // Rolls back, maps to 404.
                }
                return err
            }

            p.Price = price
            _, err = tx.Update(ctx, p)
            return err // nil commits; non-nil rolls back.
        })
}
```

`InTransaction` is re-entrant: when the repository is already inside a transaction, the callback runs directly on the current one instead of opening a nested transaction. That makes store methods composable: a service can call `Reprice` on its own, or inside a larger `InTransaction` block that spans several operations, and atomicity follows the outermost boundary.

The same discipline with `database/sql` is a few more lines, but the shape is identical: one store method, one atomic unit:

GO

```
func (s *SQLStore) Reprice(ctx context.Context, id string,
    price int64) (err error) {
    tx, err := s.db.BeginTx(ctx, nil)
    if err != nil {
        return err
    }
    defer func() {
        if err != nil {
            tx.Rollback()
            return
        }
        err = tx.Commit()
    }()

    // All statements go through tx, with the request context.
    _, err = tx.ExecContext(ctx,
        `UPDATE products SET price = $1 WHERE id = $2`, price, id)
    return err
}
```

Mapping Database Errors to HTTP

Everything storage-related converges on the central error map from Chapter 9. The strategy has three rings:

Ring 1: domain errors, mapped once. Stores translate driver errors into domain sentinels (`ErrNotFound`, `ErrSKUTaken`), and the builder's `Errors` step maps those to canonical codes for every handler at once:

```go
Errors(errors.NewOptions().
    MapErrors(errors.NotFound, catalog.ErrNotFound).
    MapErrors(errors.AlreadyExists, catalog.ErrSKUTaken)).
```

`ctx.OK`, `ctx.Created` and `ctx.NoContent` all hand a non-nil error to this pipeline before rendering anything, so the handlers you wrote earlier get correct 404 and 409 responses without a single status-code check.

Ring 2: driver defaults. `errors.NewOptions()` starts from a default map that already includes `database/sql.ErrNoRows` → 404. For `kataras/pg` users the equivalent is opt-in, one blank import:

```go
import _ "github.com/kataras/iris/v14/rest/errors/pgerrors"
```

It registers a resolver that maps `pg.ErrNoRows` to 404, duplicate-key violations to 409, and both PostgreSQL "invalid input syntax" and "operator does not exist" errors to 400 with the `SQLSTATE` noise trimmed. Duplicate-key details (table, column, constraint names, the offending value) are logged server-side but never forwarded to the client: driver errors leak schema information, which is exactly why this mapping exists. The import is deliberately separate so applications that do not use PostgreSQL keep the driver out of their dependency graph.

Ring 3: call-site detail. When a handler or service needs to enrich an error before it enters the pipeline, wrap it at the call site; the mapping still applies to the underlying error:

```go
if err := api.svc.Archive(ctx, id); err != nil {
    errors.HandleErrorf(ctx, err, "archive product %s", id)
    return
}
```

For transformations that need request data (attaching the authenticated user to a constraint violation, for instance), register a context-aware mapper with `MapErrorFuncCtx` on the error code map; Chapter 9 covers it in depth.

The result is a strict layering: drivers speak driver errors, stores speak domain errors, handlers speak `(value, err)`, and exactly one place in the program knows that "product not found" is a 404.

Summary

- Iris integrates with any database through a pattern, not adapters: handlers call a service, the service depends on a **repository interface**, and each database is one implementation of that seam.
- Every repository method takes `context.Context` first. `iris.Context` implements it, so client disconnects cancel in-flight queries; use `ctx.WithoutCancel()` for writes that must outlive the request.
- Register stores through the builder's `Services` step: a `Close` method auto-registers graceful shutdown, and `iris.Initializer`'s `Init` runs schema creation or a connectivity ping inside `Build()`.
- For PostgreSQL, `github.com/kataras/pg` pairs naturally with Iris: schema from struct tags, generic `pg.Repository[T]`, re-entrant `InTransaction`, and the `rest/errors/pgerrors` import for automatic 404/409/400 mapping of driver errors.
- `database/sql` follows the identical seam with hand-written scanning; its `ErrNoRows` maps to 404 by default. MongoDB, Redis and ORMs slot in the same way: translate their sentinels at the store boundary.
- Transactions live in store or service methods as single atomic units, never in handlers; errors they return flow through the same central map as everything else.

Further Reading

- Accessing relational databases. The official Go guide to `database/sql`: connections, pooling, queries and transactions.
- Tutorial: Accessing a relational database: a hands-on Go walkthrough from empty project to working queries.
- kataras/pg package documentation: the full API of the PostgreSQL layer used in this chapter.
- pgx package documentation: the underlying PostgreSQL driver and pool, including the connection string parameters.
- OWASP SQL Injection Prevention Cheat Sheet: why parameterized queries are non-negotiable, with examples.

CHAPTER 14

Caching Strategies

Caching in an Iris application happens at three distinct layers: the client (HTTP caching headers and `304 Not Modified` answers), the server (whole responses stored and replayed without running your handlers), and the application (your own data kept in memory so handlers skip expensive work). Each layer saves a different cost, and they compose. This chapter covers the `middleware/cache` package for client caching, the context's conditional request helpers, the `middleware/cache/httpcache` package for server-side response caching, plain-Go application caches wired through the builder's service container, and the invalidation discipline that keeps all of them honest.

Background

A cache is a saved copy of a result, kept somewhere fast so the next request for the same thing does not repeat the work that produced it. The work might be a database query, a rendered template, a call to another service, or simply transferring bytes over the network to a client that already has them. Most applications answer the same questions over and over (the same product list, the same profile page, the same country codes), and doing that work once per unique answer instead of once per request is the entire idea. When a request is answered from the copy, that is a *hit*; when the copy is missing and the real work must run, a *miss*.

A cached copy can live in three kinds of places, and each saves a different cost:

- **In the browser** (or any HTTP client). The client keeps the response it already downloaded and reuses it without contacting you. This saves the network itself: no request arrives at all.
- **In your server's memory.** Your Go process holds data or whole rendered responses in RAM. Requests still arrive, but they are answered without running the expensive code behind them. This copy is private to one process and vanishes when it restarts.
- **In a shared store.** A separate server (Redis or DragonflyDB in this chapter) holds the copies. Every instance of your application reads and writes the same store, so a copy made by one

instance serves requests hitting another, and the copies survive your application's restarts.

Every copy comes with the same catch: the moment the underlying data changes, the copy is wrong. A copy that no longer matches reality is called *stale*. Two defenses exist. A *TTL* (time to live) is an expiry stamp: the copy is trusted for, say, five minutes and then discarded regardless. *Invalidation* is actively deleting or replacing the copy at the moment the data changes. Invalidation is famously the hard part, because the code that reads from the cache is usually far away from the code that changes the data, and any path that changes data but forgets to invalidate produces bugs that look like ghosts: correct code serving wrong answers.

Browser caching is steered by two response headers. `Cache-Control` is the server's instruction about reuse: `max-age=3600` means "you may treat this copy as fresh for an hour and not ask me again." An `ETag` is a short fingerprint of the response body. When freshness expires, the browser does not blindly redownload; it sends the fingerprint back and asks "has this changed?" If not, the server answers `304 Not Modified` with an empty body (a few bytes instead of the full payload) and the browser keeps using its copy. That question-and-answer is called a *conditional request*, and it is the mechanism behind most of this chapter's first half.

With the vocabulary in place (hit, miss, stale, TTL, invalidation, `Cache-Control`, `ETag`, conditional request), the next section decides which of the three places fits which problem, and the rest of the chapter shows the Iris tool for each.

Three Layers of Caching

Before reaching for a tool, decide which cost you are trying to avoid.

Client caching saves the network. The server attaches response headers (`Cache-Control`, `Expires`, `ETag`, `Last-Modified`, `Vary`) that tell the browser or an intermediate proxy how long a response stays fresh and how to revalidate it. While the copy is fresh, the client does not contact you at all; when it revalidates, you answer a conditional request with an empty `304 Not Modified` instead of the full body. The cheapest request is the one never made.

Response caching saves the server. The rendered response body and headers are stored (in Iris: in DragonflyDB or Redis) under a key derived from the request. An equivalent request within the TTL is answered from the store; your handler, your services and your database never run. The request still travels over the network, but it costs you almost nothing to answer.

Application caching saves specific expensive work. It is not an HTTP concept: a service keeps data (a category list, exchange rates, a parsed ruleset) in memory and refreshes it on its own schedule. Handlers run normally; they just read from a fast source.

What separates the three is where the request stops, and how much work that saves:



Each caching layer stops the request earlier than the last: a fresh client cache never sends it, the cache middleware answers 304 before your handler runs, a response cache replays the whole response, and an application cache only saves the expensive call inside it.

The rest of the chapter walks down this stack, then closes with invalidation, which is where every caching design succeeds or fails.

The Cache Middleware

The `middleware/cache` package sets HTTP caching response headers and answers conditional `GET` / `HEAD` / `QUERY` requests with `304 Not Modified`. It does not store responses; it teaches clients and proxies to. `cache.New(cache.Options{...})` returns a regular handler that you register per route, per group or globally.

One header in the family needs a definition before it appears in code. `Vary` tells caches which request headers change the answer. A response compressed with gzip must never be replayed to a client that cannot decompress it, so `Vary: Accept-Encoding` instructs caches to keep separate copies per encoding the client accepts. Forgetting `Vary` is how one user's variant of a response ends up served to another.

```go
package main

import (
    "time"

    "github.com/kataras/iris/v14"
    "github.com/kataras/iris/v14/middleware/cache"
)

func main() {
    app := iris.New()

    public := cache.New(cache.Options{
        ETag: true,
        CacheControl: cache.NewCacheControlBuilder().
            Public().
            MaxAge(time.Hour).      // browsers: fresh for one hour
            SMaxAge(2 * time.Hour). // shared caches: two hours
            MustRevalidate().
            Build(),
        LastModified: true,
        Expires:      24 * time.Hour,
        Pragma:       cache.NoPragmaCache,
        Vary:         cache.NewVaryBuilder().AcceptEncoding().Build(),
    })

    app.Get("/catalog", public, func(ctx iris.Context) {
        ctx.JSON(iris.Map{"items": []string{"a", "b", "c"}})
    })

    app.Listen(":8080")
}
```

`cache.Options` covers the full header family:

Field	Type	Effect
`ETag`	`bool`	Sets an `ETag` and answers matching `If-None-Match` with 304
`CacheControl`	`string`	The `Cache-Control` value; build it with `CacheControlBuilder`
`LastModified`	`bool`	Sets `Last-Modified` and answers matching `If-Modified-Since` with 304

`Expires`	`time.Duration`	Sets `Expires` to now plus the duration
`Pragma`	`string`	Sets `Pragma`; use `cache.NoPragmaCache` for HTTP/1.0 clients
`Vary`	`string`	The `Vary` value; build it with `VaryBuilder`
`ProxyAuthenticate`	`string`	Sets `Proxy-Authenticate`; build it with `ProxyAuthenticateBuilder`

`cache.DefaultOptions` is a ready public-caching preset: `ETag` and `LastModified` on, `public, max-age=3600, s-maxage=7200, must-revalidate, proxy-revalidate`, a 24-hour `Expires`, `Pragma: no-cache` and `Vary: Accept-Encoding`. Register it as `cache.New(cache.DefaultOptions)`.

The Builders

The three builders produce plain header strings, so you can also assemble them once at startup and reuse the values:

```go
// "private, max-age=300"
private := cache.NewCacheControlBuilder().
    Private().
    MaxAge(5 * time.Minute).
    Build()

// "no-cache, no-store"
never := cache.NewCacheControlBuilder().
    NoCache().
    NoStore().
    Build()

// "Accept-Encoding, Accept-Language, Authorization"
vary := cache.NewVaryBuilder().
    AcceptEncoding().
    AcceptLanguage().
    Add("Authorization").
    Build()
```

`CacheControlBuilder` offers `Public()`, `Private()`, `NoCache()`, `NoStore()`, `MaxAge(d)`, `SMaxAge(d)`, `MustRevalidate()` and `ProxyRevalidate()`. The two max ages take a `time.Duration`, like every other duration in `middleware/`, and the header carries whole seconds: `MaxAge(time.Hour)` writes `max-age=3600`. A bare number still compiles but means nanoseconds, so `MaxAge(3600)` panics at startup with a message that tells you to pass a `time.Duration`. `VaryBuilder` offers `Accept()`, `AcceptEncoding()`, `AcceptLanguage()`, `UserAgent()` and the free-form `Add(name)`. `ProxyAuthenticateBuilder` offers `Basic(realm)` and `Digest(realm, nonce, opaque)`. Each ends with `Build()`.

Use `Private()` for per-user content that browsers may keep but shared proxies must not, and `NoCache().NoStore()` for responses that must never be reused, such as one-time tokens.

The 304 Flow

On a `GET`, `HEAD` or `QUERY` request the middleware compares the request's `If-None-Match` header against the `ETag` it manages, and `If-Modified-Since` against its `Last-Modified` value. On a match it strips the content headers and responds `304 Not Modified` with an empty body; your handler never runs. Otherwise it sets the configured headers and calls `ctx.Next()`.

One honest caveat: the middleware derives its `ETag` and `Last-Modified` values from the current time, not from your payload, so its revalidation mainly collapses rapid repeat requests. Treat the middleware as a policy tool (`Cache-Control`, `Expires`, `Vary`) and its 304 handling as a bonus. When validators must reflect your data, for example a real modification time from your database, write the conditional flow yourself with the context helpers shown in Conditional Requests by Hand.

Caching a Group of Endpoints

Most APIs contain a cluster of endpoints whose payload changes only on deployment: country and currency lists, enum labels, plan descriptions, public configuration. These "constants" endpoints are the perfect client caching target, and the clean way to cache them is one middleware on one group rather than a copy per route.

In the controller style of the recommended stack (see Chapter 4), build the middleware inside `Routing` and pass it to `r.Router` together with a closure that declares the routes:

```go
package api

import (
    "time"

    "github.com/kataras/iris/v14"
    "github.com/kataras/iris/v14/middleware/cache"
)

// MetaAPI serves rarely-changing lookup data.
type MetaAPI struct{}

func NewMetaAPI() *MetaAPI { return &MetaAPI{} }

func (api *MetaAPI) Routing(r iris.Router) {
    cached := cache.New(cache.Options{
        ETag: true,
        CacheControl: cache.NewCacheControlBuilder().
            Public().
            MaxAge(24 * time.Hour). // refetched at most daily
            Build(),
        Vary: cache.NewVaryBuilder().AcceptEncoding().Build(),
```

```
    })

    r.Router("/", cached, func(r iris.Router) {
        r.Get("/countries", api.listCountries)
        r.Get("/currencies", api.listCurrencies)
        r.Get("/units", api.listUnits)
    })
}

func (api *MetaAPI) listCountries(ctx iris.Context) {
    ctx.JSON([]string{"DE", "GR", "JP", "US"})
}

func (api *MetaAPI) listCurrencies(ctx iris.Context) {
    ctx.JSON([]string{"EUR", "JPY", "USD"})
}

func (api *MetaAPI) listUnits(ctx iris.Context) {
    ctx.JSON([]string{"imperial", "metric"})
}
```

Mount it like any other controller, with the builder (`API("/meta", api.NewMetaAPI)`) or directly (`app.Router("/meta", api.NewMetaAPI)`).

Two details make the pattern work. `r.Router` accepts handlers among its arguments: a `context.Handler` argument is registered with `Use` on the group, so it applies to the matched routes registered after it. And the path `"/"` does not create a child group; it returns the group itself, so the middleware and the closure's routes attach to the controller's own prefix. The result reads as a sentence: "everything in this block is cached with this policy."

With a day-long `max-age`, each client downloads each list at most once a day, and a deployment naturally refreshes the world within that window. If that staleness bound is too loose for an endpoint, it does not belong in the group.

Conditional Requests by Hand

When the validator must come from your data, the context gives you the raw pieces: `SetLastModified`, `CheckIfModifiedSince`, `WriteNotModified` and the combined `WriteWithExpiration`.

GO

```
package main

import (
    "sync"
    "time"

    "github.com/kataras/iris/v14"
)
```

```
// Announcements is a tiny store that tracks its own modification time.
type Announcements struct {
    mu        sync.RWMutex
    items     []string
    updatedAt time.Time
}

func (a *Announcements) Snapshot() ([]string, time.Time) {
    a.mu.RLock()
    defer a.mu.RUnlock()
    return a.items, a.updatedAt
}

func (a *Announcements) Add(item string) {
    a.mu.Lock()
    defer a.mu.Unlock()
    a.items = append(a.items, item)
    a.updatedAt = time.Now()
}

func main() {
    app := iris.New()
    news := &Announcements{}
    news.Add("v14 released")

    app.Get("/announcements", func(ctx iris.Context) {
        items, updatedAt := news.Snapshot()

        modified, err := ctx.CheckIfModifiedSince(updatedAt)
        if !modified && err == nil {
            ctx.WriteNotModified() // 304, empty body
            return
        }

        ctx.SetLastModified(updatedAt)
        ctx.JSON(items)
    })

    app.Listen(":8080")
}
```

The `!modified && err == nil` idiom is deliberate. `CheckIfModifiedSince` returns `false` with a non-nil error when the check could not run at all: the method is not `GET`, `HEAD` or `QUERY`, the request carries no `If-Modified-Since` header, the modification time is zero, or the header failed to parse. Those errors wrap the context package's `ErrPreconditionFailed`; they mean "send the full response", not "not modified". Only `false` with a nil error licenses the 304.

`WriteNotModified` sends the 304 correctly: it removes `Content-Type`, `Content-Length` and, when an `ETag` is present, `Last-Modified`, then sets the status code. For byte payloads the whole dance collapses into one call:

```
// 304 when the client is current, otherwise Last-Modified plus the body.
ctx.WriteWithExpiration(reportPDF, generatedAt)
```

On the request side, `ctx.MaxAge()` parses the request's `Cache-Control` header and returns its max-age seconds as an `int64`, or `-1` when the header is absent or unparsable. It is occasionally useful for honoring a client's declared freshness tolerance in your own caching decisions.

Server-Side Response Caching

The `middleware/cache/httpcache` package stores whole rendered responses in DragonflyDB or Redis and replays them without executing your handler chain. This is the layer that saves compute: the aggregation query, the template render, the downstream API call all run at most once per TTL window. The middleware ships with Iris; the store it talks to is an external DragonflyDB or Redis server you operate.

If those names are new: Redis and DragonflyDB are key-value stores, small standalone servers that keep data in RAM under string keys and answer lookups in well under a millisecond. DragonflyDB speaks the Redis protocol, which is why one client and one `Addr` option cover both. Being a separate process is the feature: every instance of your application shares the one store, so this is the "shared store" layer from the Background section.

```go
package main

import (
    "log"
    "time"

    "github.com/kataras/iris/v14"
    "github.com/kataras/iris/v14/middleware/cache/httpcache"
)

func main() {
    store, err := httpcache.New(httpcache.Options{
        Addr: "localhost:6379",
        TTL:  5 * time.Minute,
    })
    if err != nil {
        log.Fatal(err)
    }
    defer store.Close()

    app := iris.New()

    app.Get("/reports/summary", store.Handler(), func(ctx iris.Context) {
        // This expensive aggregation runs at most once
        // per five-minute window.
        ctx.JSON(iris.Map{"total": 42, "generated": time.Now()})
    })

    app.Listen(":8080")
}
```

`store.Handler()` caches `GET`, `QUERY` and `POST` requests (the last two because list endpoints with rich filters carry them in a JSON body, which is why the body hash is part of the key) and stores only successful `200` responses. On a hit it writes the stored headers and body directly, plus a `Cache-Control: public, max-age=<remaining TTL>` header so clients stack their own caching on top. On a miss it records the response through a response recorder and stores it after your handlers finish.

Two kinds of `200` are never stored: one that sets a cookie, and one whose `Cache-Control` carries `private`, `no-store` or `no-cache`. Such a response runs its handler on every request and goes out with the header it was given, not with the `public` of a replay. The cookie and the directive count wherever they were set, in the route handler or in a handler registered before the cache handler. That has a cost to know about: a security headers middleware that sends `no-store` on every response switches the store off for every route under it, and so does a middleware that sets a cookie on every response. Nothing is logged. A route that should be cached under the first sets its own `Cache-Control`, for example `public, max-age=60`, which replaces the inherited one.

`httpcache.Options`:

Field	Effect
`Addr`, `Password`	DragonflyDB/Redis address and optional password
`TTL`	Entry lifetime; defaults to 10 minutes
`CommandTimeout`	Store command timeout; defaults to 15 seconds
`NoQuery`	Exclude the query string from the cache key
`NoBody`	Exclude the request body from the cache key
`Headers`	Header names hashed into the key, e.g. `Authorization`
`Cookies`	Cookie names hashed into the key, e.g. `session_id`
`StartupCleanup`	Flush the store on startup
`Disable`	Turn the whole cache into a no-op

Keys follow a readable structure:

```
httpcache:<route_tmpl>:<method>:<url_path>
    [:<query_hash>][:<body_hash>][:<headers_hash>][:<cookies_hash>]
```

The key is a single line, wrapped above for width. The route template (for example `/articles/{id:uuid}`) anchors the key, so all entries of one route share a prefix that invalidation can target. The hashes cover whatever the options include.

Two of those options are safety-critical. If a cached endpoint returns user-specific data, the user's identity must be part of the key: list the `Authorization` header in `Headers` or the session cookie in `Cookies`. The middleware holds you to that for the credentials it can see. A request that carries `Authorization`, `Proxy-Authorization` or cookies is looked up and stored only when the key includes every one of them: the header is named in `Headers`, and each cookie the request sends is named in `Cookies`. Naming `Cookie` in `Headers` puts the whole cookie header in the key instead. Any other request with credentials is not cached at all: its handler runs every time, nothing is stored, and the middleware logs once to say so. With both options empty that is every request with credentials. A credential in a custom header such as `X-Token` is invisible to that check until `Headers` names it, and until then the first user's response is replayed to everyone. The handler has a say too: a response it marks `Cache-Control: private` is never stored, whatever the key holds. When a response cannot tolerate that risk at all, mark it `private` or do not response-cache it.

Naming one credential is not enough, and that costs hits. Under `Cookies: []string{"sid"}` a request that sends `sid` and one more cookie, an analytics or a theme cookie, is not cached. Under `Headers: []string{"Authorization"}` neither is a request that sends any cookie. Name

those cookies in `Cookies` too, or accept that such requests run the handler.

With sessions the order of the two handlers decides. `sessions.Handler` writes the cookie of a new session after the handlers behind it have returned (Chapter 10). Register the cache handler before it and that response reaches the cache with its `Set-Cookie`, so it is not stored. Register it after `sessions.Handler` and the cookie is written once the response is already in the store: the page made for one new session is served to the next anonymous visitor. When the order cannot change, send `Cache-Control: private` on such a page. Behind a reverse proxy that caches, send it on every page that depends on the session, whatever the order: the answer to a request that only reads the session carries no `Set-Cookie` for the proxy to go by.

A key on the session cookie stores a page once per session. After a sign-out, a request that still sends the old cookie gets the stored page until the entry expires, because a hit is answered from the store and your handler never runs. Invalidate on sign-out, or mark the page `private`.

`Disable: true` returns a fully working no-op instance: `Handler` and `Pair` return nil handlers (the framework skips nil middleware), and every store method returns immediately. Development and test environments run without a store while the wiring stays identical.

Pairing Caching with Invalidation

A response cache is only trustworthy when mutations empty it. The package wires this mechanically: `Pair(r)` returns a caching handler and an invalidating handler bound to the same router group.

```go
package api

import (
    "github.com/kataras/iris/v14"
    "github.com/kataras/iris/v14/middleware/cache/httpcache"
)

type ArticlesAPI struct {
    store *httpcache.Cache
    // plus your service dependencies
}

func NewArticlesAPI(store *httpcache.Cache) *ArticlesAPI {
    return &ArticlesAPI{store: store}
}

func (api *ArticlesAPI) Routing(r iris.Router) {
    caching, invalidating := api.store.Pair(r)

    r.Get("/", caching, api.list)
    r.Get("/{id:uuid}", caching, api.read)
    r.Post("/", invalidating, api.create)
    r.Put("/{id:uuid}", invalidating, api.update)
    r.Delete("/{id:uuid}", invalidating, api.delete)
}
```

```
// list, read, create, update, delete are ordinary handlers.
```

At build time the invalidating handler collects the key prefixes of this group's routes that carry the caching handler. At request time it runs after your handler and, when a `POST`, `PUT`, `PATCH` or `DELETE` finishes with `200`, `201` or `204`, deletes every entry under those prefixes with batched scans. A successful create drops the stale list caches; a failed one leaves them alone.

The building blocks are available separately when you need different shapes: `InvalidateHandler(prefixes...)` invalidates explicit prefixes (all `httpcache` entries when called with none), `InvalidateRoutes(r)` is the invalidating half of `Pair`, and the store methods `Invalidate(ctx, key)`, `InvalidatePrefix(ctx, prefix)`, `InvalidateAll(ctx)`, `CountPrefix(ctx, prefix)` and `CountAll(ctx)` work from any code, such as an admin endpoint or a background job. `RequestKey(r)` and `ContextKey(ctx)` expose the key derivation itself.

With the builder stack, register the store as a service. It becomes injectable into `NewArticlesAPI`, and because `*httpcache.Cache` has a `Close() error` method, the container also registers it as a shutdown closer (see Chapter 8):

GO

```
app := iris.NewBuilder().
    Prefix("/api").
    AllowOrigin("none").
    Compression(true).
    LogRequests(false).
    NoHealth().
    Errors(errors.NewOptions()).
    Services(store).
    RouterMiddlewares().
    Middlewares().
    API("/articles", api.NewArticlesAPI).
    Build()
```

Client caching and response caching are complements, not alternatives: the response cache absorbs the requests that do arrive, while `Cache-Control` headers keep many requests from arriving at all. Rarely-changing public endpoints deserve both.

Application-Level Caching

Below HTTP, caching is plain Go. Iris adds no special data-cache API here, deliberately: a struct, a mutex and a TTL are easy to write, easy to test and impossible to misdocument. What the framework contributes is lifecycle. A cache registered through `Services` is injected wherever its type is needed, warmed exactly once at startup via `iris.Initializer`, and closed at shutdown when it has a `Close` method.

The example below uses a mutex, worth two sentences if you have not met one. Your HTTP handlers run concurrently, so many of them may read the cached list at the same moment a refresh is rewriting it; a mutex (mutual exclusion lock) forces such access to take turns so nobody observes a half-written value. The `sync.RWMutex` variant is the right fit for caches because it lets any number of readers proceed together and only makes them wait when a writer holds the lock.

A cache service that fronts a slow repository:

```go
package catalog

import (
    "context"
    "sync"
    "time"
)

// Repository is the slow source of truth (database, remote API).
type Repository interface {
    Categories(ctx context.Context) ([]string, error)
}

// Cache keeps the category list in memory and refreshes it when stale.
type Cache struct {
    repo Repository
    ttl  time.Duration

    mu        sync.RWMutex
    items     []string
    fetchedAt time.Time
}

func NewCache(repo Repository) *Cache {
    return &Cache{repo: repo, ttl: 5 * time.Minute}
}

// Init implements iris.Initializer. The builder calls it once inside
// Build(), after dependency injection and before the server accepts
// traffic, so the first request never sees a cold cache. A returned
// error aborts startup.
func (c *Cache) Init(ctx context.Context) error {
    return c.refresh(ctx)
}

// Categories returns the cached list, refreshing it when the TTL has
// passed. On refresh failure it serves the stale copy rather than fail.
func (c *Cache) Categories(ctx context.Context) ([]string, error) {
    c.mu.RLock()
    items := c.items
    fresh := time.Since(c.fetchedAt) < c.ttl
    c.mu.RUnlock()

    if fresh {
        return items, nil
    }

    if err := c.refresh(ctx); err != nil {
```

```
        if items != nil {
            return items, nil // stale beats broken
        }
        return nil, err
    }

    c.mu.RLock()
    items = c.items
    c.mu.RUnlock()
    return items, nil
}

func (c *Cache) refresh(ctx context.Context) error {
    items, err := c.repo.Categories(ctx)
    if err != nil {
        return err
    }

    c.mu.Lock()
    c.items = items
    c.fetchedAt = time.Now()
    c.mu.Unlock()
    return nil
}
```

Wire it in the builder's `Services` step; the container calls `NewCache` with the registered repository and later hands the `*catalog.Cache` to any controller constructor that asks for it:

```
app := iris.NewBuilder().
    // ...
    Errors(errors.NewOptions()).
    Services(repo, catalog.NewCache).
    RouterMiddlewares().
    Middlewares().
    API("/catalog", api.NewCatalogAPI).
    Build()
```

GO

The startup ordering is the point. `Init` runs after injection and before serving, in registration order, which makes warmup failures a deploy-time signal instead of a burst of slow first requests. `Deferrables` and `Close` methods are the mirror image at shutdown (Chapter 3 covers the full chain).

For bounded key-value caching without writing your own eviction, the framework's internal generic cache is importable: `github.com/kataras/iris/v14/core/pkg/cache`. `NewMemoryCache[T](maxCount)` returns a `CacheIterator[T]` with `Get`, `Set`, `Delete`, `Walk` and `First`; when the entry count exceeds `maxCount` (or a memory-derived default when you pass zero), the oldest entries are evicted in batches. Iris uses it for the geolocation middleware and the OAuth2 stores.

```
import "github.com/kataras/iris/v14/core/pkg/cache"

profiles := cache.NewMemoryCache[Profile](10_000)
```

GO

```
if err := profiles.Set(ctx, id, &profile); err != nil {
    return err
}

cached, found, err := profiles.Get(ctx, id) // (*Profile, bool, error)
```

Chapter 20 revisits this package in the context of measurement and tuning.

Distributed Caching

Third-party territory. When several application instances must share cached data, the cache moves to an external store: Redis, DragonflyDB or Memcached, accessed through their own Go client libraries (for example `github.com/redis/go-redis/v9`), which are third-party dependencies, not Iris APIs. The pattern does not change: wrap the client in a small service with the same shape as the in-memory `catalog.Cache` above, register it through `Services`, and keep the client type out of your handlers. Note that the built-in `httpcache` middleware is already distributed for its own layer: every instance pointed at the same store shares one response cache, and an invalidation from any instance is visible to all.

Cache Invalidation

Invalidation is a design discipline more than an API, so this section is advice rather than code.

Give every entry a TTL. Invalidation logic will have bugs; a TTL bounds how long any bug can serve stale data. Pick the TTL from a business question (“how stale may a price be?”) and treat explicit invalidation as an optimization on top, never as the only line of defense. The `httpcache` store enforces this shape by design: entries always expire.

Invalidate on the write path. The code that changes data is the only code that reliably knows data changed. The `Pair` pattern applies this mechanically for response caches; follow the same rule in your own services, where the method that writes also invalidates or updates the corresponding cache entry. Invalidation sprinkled anywhere else, in background sweepers or in readers, drifts out of sync with reality.

Prefer coarse invalidation. Dropping a whole route prefix after any mutation is crude but easy to prove correct, and a cache exists to make reads cheap, so rebuilding a handful of entries is usually noise. Precise per-entry bookkeeping (which lists contain article 42?) is where stale-data bugs live. Reach for it only when measurements show the coarse flush is actually hurting.

Client caches cannot be invalidated. Once a browser holds a response with `max-age=86400`, no server-side flush reaches it. So the `max-age` you send is a promise: choose it as your true staleness tolerance. When content must update promptly, send a short `max-age` (or `no-cache`) and rely on cheap 304 revalidation. For static assets, invalidate by changing the URL instead: fingerprinted file names with a long `max-age` give instant updates and permanent caching at once.

Do not cache what you cannot afford to leak. Responses that vary by identity need the identity in the cache key (`Headers`/`Cookies` in `httpcache.Options`, `Vary` plus `private` on the client side) or no response caching at all. `private` works on both sides: browsers keep the response to one user, and `httpcache` does not store it. An authorization bug through a cache is still an authorization bug.

Measure before layering. A cache with a low hit rate is pure complexity. Count hits and misses (the store's `CountPrefix` helps, as do the metrics of Chapter 17) before adding another layer.

Summary

- Caching has three layers, each saving a different cost: client caching saves the network, response caching saves the server, application caching saves specific expensive work. They compose.
- `middleware/cache` sets the header family (`Cache-Control`, `Expires`, `ETag`, `Last-Modified`, `Vary`, `Pragma`, `Proxy-Authenticate`) through `cache.New(cache.Options{...})` and the `CacheControlBuilder`, `VaryBuilder` and `ProxyAuthenticateBuilder` fluent builders, and answers conditional requests with `304 Not Modified`.
- Group rarely-changing "constants" endpoints under one cached router group: `r.Router("/", cached, func(r iris.Router) { ... })` keeps the policy in one place.
- For data-derived validators, use `ctx.CheckIfModifiedSince`, `ctx.SetLastModified`, `ctx.WriteNotModified` and `ctx.WriteWithExpiration`; remember that only `false` with a nil error means "not modified".
- `middleware/cache/httpcache` replays stored `200` responses for `GET`, `QUERY` and `POST` from DragonflyDB/Redis, and never stores one marked `private`, `no-store` or `no-cache`; `Pair` couples caching routes with automatic invalidation on successful mutations, and `Disable` makes the whole thing a no-op for development.
- Application caches are plain Go services: register them with `Services`, warm them once at startup through `iris.Initializer`, and let `Close` methods participate in graceful shutdown. The importable `core/pkg/cache` package provides a bounded generic memory cache.

- Invalidate on the write path, keep TTLs on everything, prefer coarse flushes, and treat client-side `max-age` as an unbreakable promise.

Further Reading

- MDN: HTTP caching. The browser-side story in plain language: freshness, revalidation, and how shared and private caches differ.
- RFC 9111: HTTP Caching: the authoritative specification every cache, browser or proxy, must follow.
- MDN: Cache-Control: every directive (`max-age`, `private`, `no-store`, `must-revalidate`) with examples of common recipes.
- MDN: ETag: validators and conditional requests, the machinery behind `304 Not Modified`.
- The Go sync package: `Mutex` and `RWMutex`, used by every hand-rolled in-memory cache.
- go-redis: the Go client library for the Redis-protocol stores this chapter's distributed layers rely on.

CHAPTER 15

Internationalization (i18n)

Iris ships internationalization as a regular middleware package, `github.com/kataras/iris/v14/middleware/i18n`. It loads translation files in INI, JSON, TOML or YAML format, matches the request language against your registered locales using BCP 47 tags, and exposes the matched locale to your handlers. This chapter covers only what the package actually provides: loading translations, per-request language detection, message rendering with plurals and variables, and the optional path/subdomain rewriting router.

Background

Internationalization (abbreviated i18n, for the eighteen letters between the “i” and the “n”) is structuring an application so it can address each user in their own language. The obstacle is rarely translation itself; it is where the words live. When user-facing strings are hardcoded, written literally inside handlers and templates, adding a second language means hunting down every string in the codebase, and translators, who are usually not programmers, would have to edit Go source to do their job. Worse, a fixed string like `"You have 3 new messages"` glued together in code embeds English grammar into program logic.

The standard cure is a *message catalog*: user-facing text moves out of code and into plain data files, one set per language, where every message has a *key*: a stable name like `welcome` or `FreeDay`. Code asks for the key; the catalog answers in whichever language was matched for the current request. That is the problem that disappears from your handlers: they no longer contain any user-facing prose at all, and shipping a new language becomes adding files rather than changing code.

Translation alone is not enough, which is why catalogs are more than key-to-string maps:

- **Plurals.** English has two forms (1 day, 5 days), but languages disagree wildly: Russian uses different forms for 1, 2–4 and 5+; Arabic distinguishes six cases. A catalog therefore stores several variants of a message and picks one based on a count at run time.

- **Variables and word order.** A sentence with a value in the middle ("You are 10 minutes late") cannot be built by concatenating halves, because other languages put the value elsewhere. Messages carry placeholders instead, and the placeholder can move freely per language.
- **Locale codes.** Languages are identified by standardized *language tags* (BCP 47, defined in RFC 5646) such as `el-GR`: a language code (`el`, Greek) optionally followed by a region code (`GR`, Greece). The region part matters when regions diverge: `en-US` and `en-GB` spell, format dates and count currency differently. A *locale* is the matched combination the application serves.

The remaining question is how a server knows which language a given request wants. Every browser volunteers an answer: the `Accept-Language` request header carries the user's ranked preferences, taken from their operating system or browser settings (for example `el-GR, el;q=0.9, en;q=0.8`, where `q` weights the ranking). That is a good default and a poor final word, since users borrow machines and override preferences, so real applications also honor explicit choices: a `?lang=` URL parameter, a saved cookie, a language subdomain (`el.example.com`) or a path prefix (`/el-GR/account`). The i18n middleware you meet below checks exactly these sources, in a documented order, and falls back to your default language when nothing matches.

Package Overview

The package has three cooperating pieces:

- `i18n.New()` returns an `*i18n.I18n` catalog. You load locale files into it once at startup with `Load`, `LoadFS`, `LoadKV` or `LoadAssets`.
- `i18n.NewHandler(locales, i18n.HandlerOptions{...})` returns a regular middleware handler. Register it with `Use`. On every request it detects the language and stores the matched locale on the context.
- `i18n.NewRouter(locales, i18n.RouterOptions{...})` is optional. Register it with `UseRouter` to strip language prefixes such as `/el-GR/...` (or language subdomains such as `el.example.com`) before routing happens.

Handlers then read translations through the package-level functions `i18n.GetLocale(ctx)` and `i18n.GetMessage(ctx, key, args...)`.

Loading Translations

`Load` accepts a glob pattern and an optional list of language codes. The language of each file is deduced from its path: a directory named after the language (`locales/el-GR/anything.yml`) or a file named after it (`locales/el-GR.yml`) both work.

```go
package main

import "github.com/kataras/iris/v14/middleware/i18n"

func main() {
    locales := i18n.New()

    // First language is the default/fallback one.
    err := locales.Load("./locales/*", "en-US", "el-GR")
    if err != nil {
        panic(err)
    }
}
```

Rules worth knowing:

- The first language in the list is the default. `SetDefault("el-GR")` can reorder that after loading, but usually you just list the default first.
- If you omit the language list entirely, the languages are discovered dynamically from the file paths.
- With a list, a file whose own name is a language code by itself (`locales/en-US/app.yml` : `app` , `api` and `tag` all are) belongs to the language of its directory. Without a list such a file name becomes a language of its own, so pass the list when you name files that way.
- With a list, `Load` , `LoadFS` and `LoadAssets` return a `locales not found` error when no file is loaded for any of the listed languages, which is what a pattern one directory level off produces.
- The file extension selects the parser: `.ini` , `.json` , `.toml` / `.tml` , and `.yaml` / `.yml` (the default). Multiple files for the same language are merged into one message catalog.
- Nested keys flatten with dots: a YAML map `user: {name: ...}` is fetched as `"user.name"` . INI sections behave the same way.

Embedded Files

Go can compile arbitrary files straight into the executable: the `//go:embed` directive tells the compiler to bundle the named files, and the resulting `embed.FS` value serves them through the standard `fs.FS` read-only filesystem interface. For translations this means the deployed binary is self-contained. There is no `locales/` directory to ship, forget or mismatch.

`LoadFS` works with any `fs.FS` , including `embed.FS` , so translations can be compiled into the binary:

```go
import "embed"

//go:embed locales/*
var localesFS embed.FS

func load(locales *i18n.I18n) error {
    return locales.LoadFS(localesFS, "locales/*", "en-US", "el-GR")
}
```

Maps

`LoadKV` loads translations from an in-memory `i18n.LangMap` (`map[string]map[string]any`), which is convenient in tests:

```go
langMap := i18n.LangMap{
    "en-US": map[string]any{
        "hello": "Hello",
    },
    "el-GR": map[string]any{
        "hello": "Γειά",
    },
}

err := locales.LoadKV(langMap, "en-US", "el-GR")
```

`LoadAssets(assetNames, asset, languages...)` exists as well, for go-bindata style generated assets.

Loader Options

The `Loader` field of `*i18n.I18n` is an `i18n.LoaderConfig`. Configure it before calling any `Load` method:

```go
locales := i18n.New()
locales.Loader.Strict = true  // fail unless every language loads.
locales.Loader.Left = "{{"    // template delimiters (defaults).
locales.Loader.Right = "}}"

locales.SetDefaultMessageFunc(
    func(langMatched, key string, args ...any) string {
        return "[missing: " + key + "]"
    })
```

Without a default message function, a lookup for an unknown key returns an empty string. `Strict` wants at least one file for every listed language and fails the load otherwise, with an error such as `locales expected to be 2 but 1 parsed`. Left off, a listed language without files is accepted and has no messages.

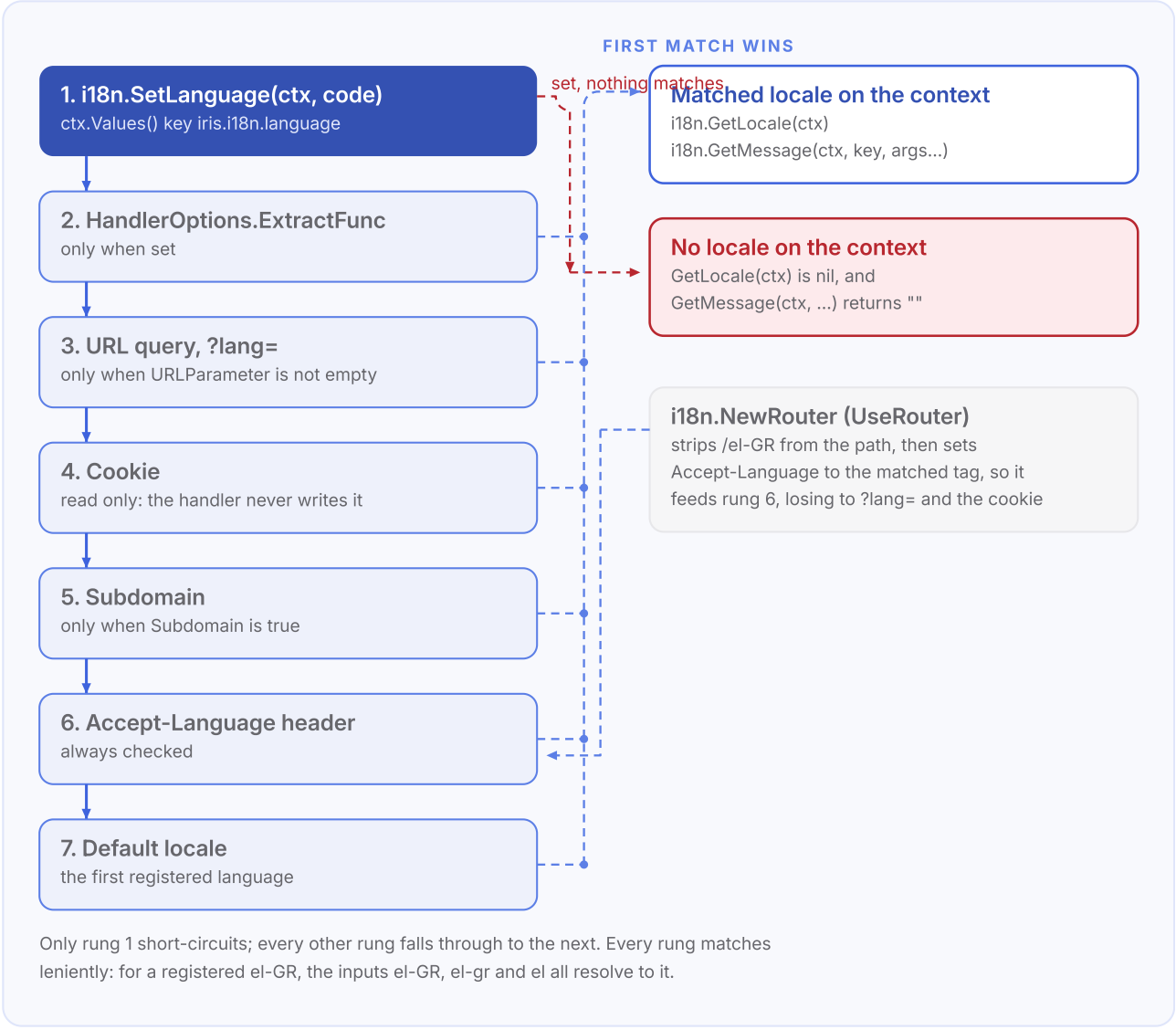
The Request Handler

`i18n.NewHandler` builds the middleware that performs language detection. All detection sources are opt-in through `i18n.HandlerOptions`:

```go
app.Use(i18n.NewHandler(locales, i18n.HandlerOptions{
    // ?lang=el-GR
    URLParameter: "lang",
    // read the language from a cookie named "lang".
    Cookie: "lang",
    // treat a subdomain as a language: el.example.com.
    Subdomain: true,
    // optional custom source, checked before the ones above.
    ExtractFunc: nil,
}))
```

The handler checks the sources in a fixed order and stops at the first registered language that matches:

1. A language force-set by server code (`i18n.SetLanguage`, see below).
2. `ExtractFunc`, if set.
3. The URL query parameter (`URLParameter`).
4. The cookie (`Cookie`). The handler only reads this cookie; writing it is your job or the router's (see `RouterOptions.Cookie`).
5. The subdomain, if `Subdomain` is true.
6. The `Accept-Language` request header (always checked).
7. Fallback: the default locale, i.e. the first registered language.



The i18n handler tries seven sources in order and the first match wins; only a language set on the context short-circuits, and it yields no locale at all when nothing registered matches it.

Matching is lenient: for a registered `el-GR`, the inputs `el-GR`, `el-gr` and plain `el` all resolve to the same locale.

One asymmetry in that list is worth knowing. Rungs 2 through 7 fall through: a source that is unset, empty or unmatched simply hands over to the next. Rung 1 does not. A language force-set with `i18n.SetLanguage` that matches nothing registered leaves the request with *no* locale at all, so `i18n.GetLocale(ctx)` returns nil and `i18n.GetMessage` returns an empty string. It does not fall back to the default.

Translating in Handlers

`i18n.GetLocale(ctx)` returns the matched `locale.Locale`, an interface with `Language()`, `Index()`, `Tag()` and `GetMessage(key, args...)`. `i18n.GetMessage(ctx, key, args...)` is a shortcut for `GetLocale(ctx).GetMessage(key, args...)`.

```go
app.Get("/", func(ctx iris.Context) {
    locale := i18n.GetLocale(ctx)
    hi := locale.GetMessage("hi", "iris")
    // Equivalent: hi := i18n.GetMessage(ctx, "hi", "iris")

    ctx.Writef("[%s] %s", locale.Language(), hi)
})
```

Plain messages are rendered with printf-style verbs, so a locale entry `hi = hello, %s` combined with `GetMessage("hi", "iris")` produces `hello, iris`. Indexed verbs such as `%[1]d` are supported too.

To override the detected language:

- `i18n.SetLocaleByLanguage(ctx, "el-GR")` switches the locale for the rest of the current request. Use it inside route handlers, after the i18n handler has run.
- `i18n.SetLanguage(ctx, "el-GR")` stores a language code that the i18n handler honors above every other source. It only has an effect when called before the handler runs, e.g. in a middleware registered earlier.

Pluralization and Variables

Plural Forms

A message becomes plural-aware when its value is a map whose keys are plural forms. The supported forms are `zero`, `one`, `two`, `other` and the comparisons `=N`, `<N`, `>N`:

```yaml
FreeDay:
  one: "You have a day off"
  other: "You have %[1]d free days"
  "=3": "You have three days and %[2]d minutes off."
```

The plural count is the first argument to `GetMessage` and it also feeds the printf verbs:

```go
i18n.GetMessage(ctx, "FreeDay", 1)      // You have a day off
i18n.GetMessage(ctx, "FreeDay", 5)      // You have 5 free days
i18n.GetMessage(ctx, "FreeDay", 3, 15)  // ... three days and 15 minutes off.
```

Calling a plural message without a count does not panic; the returned string is an error description, which makes the mistake visible during development.

Variables

A locale file can declare a `Vars` list. Each variable has its own plural forms and is referenced inside messages as `${Name}`. The count that selects the variable's form comes from the message arguments:

```yaml
Vars:
  - Minutes:
      one: "minute"
      other: "minutes"

YouLate: "You are %[1]d ${Minutes} late."
```

```go
i18n.GetMessage(ctx, "YouLate", 1)   // You are 1 minute late.
i18n.GetMessage(ctx, "YouLate", 10)  // You are 10 minutes late.
```

`Vars` declared at the top level of a file are shared by every message of that locale. A message can also carry its own local variables by using a map value with `Text` and `Vars` keys:

```yaml
HouseCount:
  Text: "She has %[1]d ${Houses}"
  Vars:
    - Houses:
        one: "house"
        other: "houses"
```

Template Messages

Go's standard library includes `text/template`, a small templating language: a string containing placeholders such as `{{.name}}` is parsed once, then rendered against a data value that fills the placeholders in. Compared to printf verbs, templates name their inputs, which reads better when a message takes several values.

A message whose value contains the configured delimiters (default `{{` and `}}`) is parsed as a Go `text/template`. The first `GetMessage` argument becomes the template data, usually a map:

```yaml
welcome: "Welcome back, {{.name}}!"
```

```go
i18n.GetMessage(ctx, "welcome", iris.Map{"name": "Makis"})
// Welcome back, Makis!
```

For plural template messages the reserved data key `PluralCount` selects the plural form. Custom per-locale template functions can be registered through the `Funcs` field of the loader configuration; every locale template also has a built-in `tr` function that resolves other keys of the same locale.

Path and Subdomain Rewriting

`i18n.NewRouter` lets URLs carry the language, e.g. `/el-GR/account` or `el.example.com/account`. It must run before routing, so register it with `UseRouter`, and keep the regular handler registered with `Use` as well:

```go
app.UseRouter(i18n.NewRouter(locales, i18n.RouterOptions{
    // write the detected language to this response cookie.
    Cookie: "lang",
    // also detect and strip language subdomains.
    Subdomain: true,
    // optional hook, called with the detected language tag.
    InjectFunc: nil,
}))

app.Use(i18n.NewHandler(locales, i18n.HandlerOptions{
    URLParameter: "lang",
    Cookie:       "lang",
}))
```

When the first path segment (or the subdomain, with `Subdomain: true`) matches a registered language, the router rewrites the request in place: it removes the language part, sets the request's `Accept-Language` header to the matched tag and, if `Cookie` is set, writes the language to a response cookie shared across subdomains. Routing then continues with the clean path, so a route defined as `/account` serves `/el-GR/account` too, and the i18n handler picks the language up from the rewritten header. Lenient matching applies here as well: `/el/account` and `/el-gr/account` both work.

Template Integration

Server-side views (see Chapter 6) integrate through plain template functions. The pattern used by the package itself registers a static `tr` function on the engine and passes per-request closures as view data:

```go
import (
    "github.com/kataras/blocks"
```

```
    "github.com/kataras/iris/v14"
    "github.com/kataras/iris/v14/middleware/i18n"
    "github.com/kataras/iris/v14/middleware/view"
)

func registerViews(app *iris.Application, locales *i18n.I18n) {
    templates := blocks.New("./views")
    // Static form, any language: {{tr "el-GR" "hi" "iris"}}.
    templates.AddFunc("tr",
        func(lang, key string, args ...any) string {
            if locale, ok := locales.GetLocale(lang); ok {
                return locale.GetMessage(key, args...)
            }
            return "[INVALID LOCALE] " + lang
        })
    templates.Load()
    app.UseRouter(view.Handler(templates))

    app.Get("/", func(ctx iris.Context) {
        view.Render(ctx, "index.html", "", iris.Map{
            // Request-aware form: {{call .tr "hi" "iris"}}.
            "tr": func(key string, args ...any) string {
                return i18n.GetMessage(ctx, key, args...)
            },
        })
    })
}
```

```HTML
<h1>{{call .tr "hi" "iris"}}</h1>
<p>{{tr "el-GR" "hi" "iris"}}</p>
```

Complete Example

A runnable program with two locale files, plural messages, URL-parameter and header detection plus path prefixes:

```
myapp/
├── locales/
│   ├── en-US.yml
│   └── el-GR.yml
└── main.go
```

locales/en-US.yml

```YAML
hi: "hello, %s"
FreeDay:
  one: "You have one day off."
  other: "You have %d days off."
```

locales/el-GR.yml

```yaml
hi: "γεια, %s"
FreeDay:
  one: "Έχεις μία ελεύθερη μέρα."
  other: "Έχεις %d ελεύθερες μέρες."
```

main.go

```go
package main

import (
    "github.com/kataras/iris/v14"
    "github.com/kataras/iris/v14/middleware/i18n"
)

func main() {
    app := iris.New()

    locales := i18n.New()
    if err := locales.Load("./locales/*", "en-US", "el-GR"); err != nil {
        panic(err)
    }

    app.UseRouter(i18n.NewRouter(locales, i18n.RouterOptions{
        Cookie: "lang",
    }))

    app.Use(i18n.NewHandler(locales, i18n.HandlerOptions{
        URLParameter: "lang",
        Cookie:       "lang",
    }))

    app.Get("/", func(ctx iris.Context) {
        locale := i18n.GetLocale(ctx)
        ctx.Writef("[%s] %s", locale.Language(),
            locale.GetMessage("hi", "iris"))
    })

    app.Get("/days/{count:int}", func(ctx iris.Context) {
        count := ctx.Params().GetInt("count")
        ctx.WriteString(i18n.GetMessage(ctx, "FreeDay", count))
    })

    app.Listen(":8080")
}
```

```sh
curl http://localhost:8080/


curl -H "Accept-Language: el" http://localhost:8080/


curl "http://localhost:8080/days/5?lang=el-GR"


curl http://localhost:8080/el-GR/days/1
```

Test i18n behavior the same way as any other handler chain, with `httptest.New(t, app)` and requests that vary the `lang` query parameter or the `Accept-Language` header; Chapter 16 covers the tooling.

Summary

- `i18n.New()` creates the catalog; `Load`, `LoadFS`, `LoadKV` and `LoadAssets` fill it from glob patterns, embedded filesystems, maps or generated assets. INI, JSON, TOML and YAML files are supported, and the language is deduced from the file path.
- `i18n.NewHandler` (registered with `Use`) detects the language per request in a fixed order: server-set value, `ExtractFunc`, URL parameter, cookie, subdomain, `Accept-Language` header, then the default locale.
- Handlers translate with `i18n.GetMessage(ctx, key, args...)` or through `i18n.GetLocale(ctx)`. Messages use printf verbs, Go templates, plural forms (`zero`, `one`, `two`, `other`, `=N`, `<N`, `>N`) and `${Var}` variables.
- `i18n.NewRouter` (registered with `UseRouter`) rewrites language path prefixes and subdomains before routing and can persist the choice in a cookie.
- View engines integrate via ordinary template functions such as a `tr` helper backed by `GetLocale` / `GetMessage`.

Further Reading

- RFC 5646: Tags for Identifying Languages: the BCP 47 language-tag grammar behind codes like `el-GR` and `en-US`.
- MDN: Accept-Language: how browsers express ranked language preferences, including `q` weights.
- MDN: Content negotiation: the wider HTTP mechanism that language selection is one instance of.
- The Go text/template package: the full syntax available inside template-style messages.
- The Go embed package: `//go:embed` and `embed.FS`, used to compile locale files into the binary.

CHAPTER 16

Testing

Testing is where the architecture of this book pays off. Because an Iris application is assembled by a `newApp` function that receives its configuration and its dependencies as parameters, a test can build the real application (real router, real middleware, real error map) and swap only the edges: an in-memory store instead of a database, a fixed password instead of a secret from the environment. The `httptest` package then drives that application entirely in memory: no ports, no network, no sleeps. A suite of full HTTP tests runs in milliseconds. This chapter covers the `httptest` API, the request-and-assertion chain, designing applications so they are testable in the first place, and the specific tests every service should have: guards, validation, the central error map, and the proof that internal errors never reach a client.

Background

An automated test is a small Go function that runs a piece of your application and checks the result against what you expect; the `go test` command runs every such function and reports which ones failed. Its value is not proving today's build works; you can click through the app for that. Its value is that the proof re-runs on every future change. When a change six months from now quietly breaks note creation, the suite says so within seconds, names the exact expectation that no longer holds, and points at the change that broke it. That failure mode (old behavior broken by new code) is called a *regression*, and catching regressions automatically is the reason test suites exist. Each individual check inside a test ("the status must be 201") is an *assertion*.

Tests are usually named by how much of the system they exercise: a *unit test* checks one function or type in isolation, an *integration test* checks that several pieces work when wired together, and an *end-to-end test* drives the assembled application the way a real client would. In one breath: unit tests prove each part works, integration tests prove the parts fit, end-to-end tests prove the whole does the job. This chapter leans on end-to-end-style HTTP tests because in Iris they are as cheap as unit tests, plus plain unit tests for domain logic.

Real applications depend on things a test does not want: a production database, a mail provider, the actual passage of time. A *test double* is a stand-in for such a dependency. Two kinds appear in this chapter: a *fake* is a genuinely working but lightweight implementation, like an in-memory map pretending to be a database; a *stub* is even simpler: it returns canned answers, often from a single method, such as "every query fails with this error". In many languages test doubles require mocking frameworks and code generation. In Go they are ordinary structs, because of how interfaces work: an interface names a set of methods, and any type with those methods satisfies it implicitly. If your handler depends on a `Repository` interface rather than a concrete database type, then any five-line struct with the same methods can step in, and the handler cannot tell the difference. That is the whole trick, and this chapter uses it constantly.

Finally, *in-memory HTTP testing*. The naive way to test a web server is to actually start it: listen on a port, send requests over the network, wait for responses. Every item in that sentence is a liability: ports collide when tests run in parallel, network buffers add timing you do not control, and waits turn into `time.Sleep` calls that make the suite slow and *flaky* (failing occasionally without any code change, the most corrosive property a test can have). The in-memory approach removes the transport entirely: the test hands a request object directly to the router, in the same process, through the same code path a real request would take once past the socket. No port, no network, no waiting. When the call returns, the handler has run. Tests become deterministic and fast enough to run on every save, which is what makes them actually get run.

How Testing Works in Iris

`httptest.New(t, app)` prepares an application for testing and returns an `*httpexpect.Expect` value, a fluent client from the httpexpect library that Iris integrates:

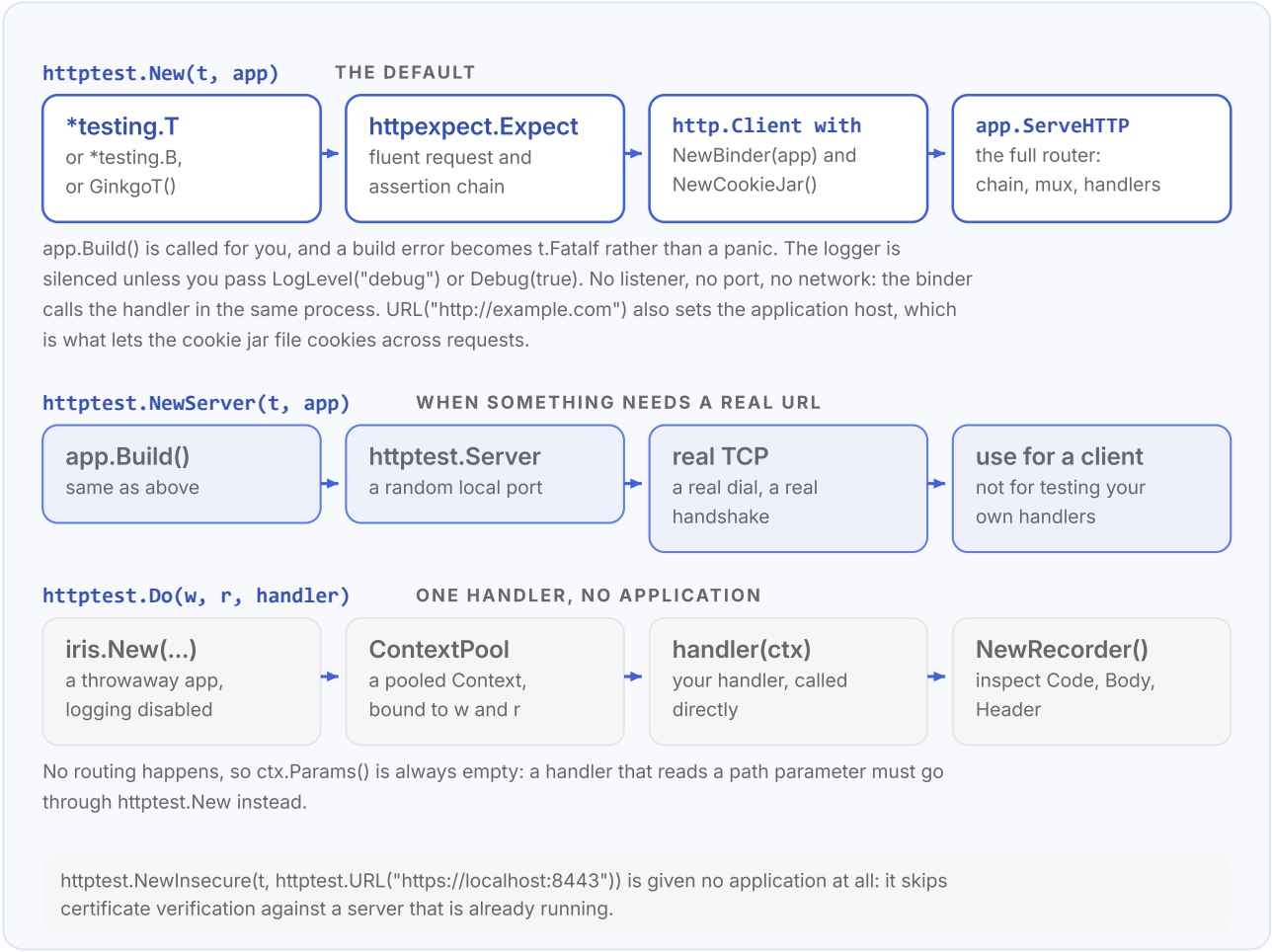
```go
import "github.com/kataras/iris/v14/httptest"

e := httptest.New(t, app)
```

Three things happen inside that call, and each one matters:

1. **The application is built.** `httptest.New` calls `app.Build()` for you and reports a build failure through `t.Fatalf`: a misconfigured route or an unresolvable dependency fails the test immediately with the real error message, instead of panicking.
2. **Requests bypass the network.** The returned client's transport is bound directly to the application. `e.GET("/x")` invokes the router in-process; nothing listens on a port, so tests are fast, parallel-safe and free of port conflicts.

3. **Logs are silenced.** The application's log level is set to `disable` by default, so `go test` output stays readable. Pass `httptest.LogLevel("debug")` or `httptest.Debug(true)` when you need to see what a failing test is doing.



httptest.New binds the client transport straight to your built application so requests never touch the network, NewServer puts a real port in front of it, and Do calls a single handler with no router at all.

The first argument is not literally a `*testing.T`: it is the `httptest.IrisTesty` interface, which `*testing.T`, `*testing.B` and Ginkgo's `GinkgoT()` all satisfy, so the same helper serves tests, benchmarks and BDD suites.

The `httptest` package also re-exports every HTTP status code (`httptest.StatusOK`, `httptest.StatusNotFound`, ...), so test files do not need to import `net/http` just for constants.

Your First Test

The smallest possible round trip:

```go
package main

import (
    "testing"

    "github.com/kataras/iris/v14"
    "github.com/kataras/iris/v14/httptest"
)

func TestHello(t *testing.T) {
    app := iris.New()
    app.Get("/hello", func(ctx iris.Context) {
        ctx.JSON(iris.Map{"message": "hello"})
    })

    e := httptest.New(t, app)
    e.GET("/hello").Expect().
        Status(httptest.StatusOK).
        JSON().Object().
        HasValue("message", "hello")
}
```

Every request follows the same shape: build the request (`e.GET(...)`, optionally with body, query and headers), execute it with `Expect()`, then chain assertions on the response. A failed assertion reports through the `t` you passed to `httptest.New`, with the expected and actual values.

Designing for Testability

The single most important testing decision is not in a test file at all. It is the signature of your application factory:

```go
func newApp(cfg Config, repo store.Repository) *iris.Application
```

`main` calls it with production values; tests call it with their own. Everything else follows from this:

- **Seams are parameters.** Anything a test may want to replace, such as the repository, a mail sender or a clock, arrives as an argument (usually an interface) and is handed to the builder's `Services` step, which the dependency container uses to satisfy controller constructors (Chapter 8).
- **Tests exercise the real application.** Because `newApp` is the same function `main` uses, every test request passes through the real router, the real middleware chain and the real central error map. There is no "test router" that behaves differently from production.

- **Configuration is a value.** A `Config` struct built in the test replaces files and environment variables; nothing global to reset between tests.

The Bookshelf service in the framework's `TUTORIAL.md` is a complete, compile-verified worked example of this pattern. This chapter uses a smaller application in the same shape.

The Application Under Test

A notes service: a domain package that owns the entities, the validation and the storage interface, and a `main` package that wires controllers behind `iris.NewBuilder` (Chapter 3).

GO

```go
// store/store.go
package store

import (
    "context"
    "errors"
    "sync"
    "uuid"

    "github.com/kataras/iris/v14/rest/errors/validation"
)

// ErrNotFound is a domain sentinel; the HTTP layer maps it to 404 once.
var ErrNotFound = errors.New("store: note not found")

type Note struct {
    ID   uuid.UUID `json:"id"`
    Text string    `json:"text"`
}

// NoteInput is the request payload. Implementing Validate makes
// ctx.BindJSON run it automatically after decoding (Chapter 5).
type NoteInput struct {
    Text string `json:"text"`
}

func (in NoteInput) Validate() error {
    return validation.Join(
        validation.String("text", in.Text).NotEmpty().Length(1, 280),
    )
}

// Repository is the storage seam: production wires a database
// implementation, tests wire MemStore or a one-method stub.
type Repository interface {
    Insert(ctx context.Context, n Note) error
    Notes(ctx context.Context) ([]Note, error)
    Note(ctx context.Context, id uuid.UUID) (Note, error) // ErrNotFound
}

// MemStore is the in-memory Repository: this chapter's storage and the
// test suite's default fake.
type MemStore struct {
```

```
    mu    sync.RWMutex
    notes map[uuid.UUID]Note
}

var _ Repository = (*MemStore)(nil)

func NewMemStore() *MemStore {
    return &MemStore{notes: make(map[uuid.UUID]Note)}
}

func (s *MemStore) Insert(_ context.Context, n Note) error {
    s.mu.Lock()
    defer s.mu.Unlock()

    s.notes[n.ID] = n
    return nil
}

func (s *MemStore) Notes(_ context.Context) ([]Note, error) {
    s.mu.RLock()
    defer s.mu.RUnlock()

    list := make([]Note, 0, len(s.notes))
    for _, n := range s.notes {
        list = append(list, n)
    }
    return list, nil
}

func (s *MemStore) Note(_ context.Context, id uuid.UUID) (Note, error) {
    s.mu.RLock()
    defer s.mu.RUnlock()

    n, ok := s.notes[id]
    if !ok {
        return Note{}, ErrNotFound
    }
    return n, nil
}
```

The HTTP layer: two controllers and the composition root.

GO

```
// main.go
package main

import (
    "os"
    "uuid"

    "github.com/kataras/iris/v14"
    "github.com/kataras/iris/v14/middleware/basicauth"
    "github.com/kataras/iris/v14/rest/errors"

    "notes/store"
)

// NotesAPI serves /api/notes. Its constructor declares its dependency;
// the container calls it with the Repository registered in Services.
type NotesAPI struct {
    repo store.Repository
```

```
}

func NewNotesAPI(repo store.Repository) *NotesAPI {
    return &NotesAPI{repo: repo}
}

func (api *NotesAPI) Routing(r iris.Router) {
    r.Get("/", api.list)
    r.Post("/", api.create)
    r.Get("/{id:uuid}", api.get)
}

func (api *NotesAPI) list(ctx iris.Context) {
    notes, err := api.repo.Notes(ctx)
    ctx.List(notes, err)
}

func (api *NotesAPI) create(ctx iris.Context) {
    input, ok := ctx.BindJSON[store.NoteInput]()
    if !ok {
        return // 400 already written: malformed JSON or failed Validate.
    }

    n := store.Note{ID: uuid.New(), Text: input.Text}
    ctx.Created(n.ID, api.repo.Insert(ctx, n))
}

func (api *NotesAPI) get(ctx iris.Context) {
    id := ctx.Params().GetUUID("id")

    n, err := api.repo.Note(ctx, id)
    ctx.OK(n, err) // store.ErrNotFound renders 404 (central map).
}

// AdminAPI is mounted behind basic auth in newApp.
type AdminAPI struct {
    repo store.Repository
}

func NewAdminAPI(repo store.Repository) *AdminAPI {
    return &AdminAPI{repo: repo}
}

func (api *AdminAPI) Routing(r iris.Router) {
    r.Get("/stats", api.stats)
}

func (api *AdminAPI) stats(ctx iris.Context) {
    notes, err := api.repo.Notes(ctx)
    ctx.OK(iris.Map{"notes": len(notes)}, err)
}

// Config carries what tests want to control. A real application loads
// it from YAML and the environment (Chapter 11).
type Config struct {
    AdminUser     string
    AdminPassword string
}

// newApp is the composition root and the test seam: main passes the
// real configuration and store, tests pass their own.
func newApp(cfg Config, repo store.Repository) *iris.Application {
```

```
    return iris.NewBuilder().
        Prefix("/api").
        AllowOrigin("none").
        Compression(false).
        LogRequests(false).
        NoHealth().
        Errors(errors.NewOptions().
            MapErrors(errors.NotFound, store.ErrNotFound)).
        Services(repo).
        RouterMiddlewares().
        Middlewares().
        API("/notes", NewNotesAPI).
        API("/admin", basicauth.Default(map[string]string{
            cfg.AdminUser: cfg.AdminPassword,
        }), NewAdminAPI).
        Build()
}

func main() {
    cfg := Config{
        AdminUser:     "admin",
        AdminPassword: os.Getenv("ADMIN_PASSWORD"),
    }

    app := newApp(cfg, store.NewMemStore())
    app.Listen(":8080")
}
```

Note what `newApp` does not do: it does not read files, does not touch `os.Getenv`, does not construct its own storage. Those are `main`'s job. That separation is the entire trick.

End-to-End Tests

The first test walks the happy path and the two most common failure paths in one function. Each test builds its own application and store, so tests share no state and can run in parallel.

GO

```
// main_test.go
package main

import (
    "testing"
    "uuid"

    "github.com/kataras/iris/v14/httptest"

    "notes/store"
)

func testConfig() Config {
    return Config{AdminUser: "admin", AdminPassword: "test-password"}
}

func TestNotesCRUD(t *testing.T) {
    app := newApp(testConfig(), store.NewMemStore())
```

```
    e := httptest.New(t, app)

    // Create; extract the generated id from the response.
    id := e.POST("/api/notes").
        WithJSON(store.NoteInput{Text: "buy milk"}).
        Expect().Status(httptest.StatusCreated).
        JSON().Object().Value("id").String().NotEmpty().Raw()

    // Read back.
    e.GET("/api/notes/"+id).Expect().Status(httptest.StatusOK).
        JSON().Object().HasValue("text", "buy milk")

    // The listing contains exactly one note.
    e.GET("/api/notes").Expect().Status(httptest.StatusOK).
        JSON().Array().Length().IsEqual(1)

    // Validation: ctx.BindJSON runs NoteInput.Validate -> 400.
    e.POST("/api/notes").
        WithJSON(store.NoteInput{Text: ""}).
        Expect().Status(httptest.StatusBadRequest)

    // The central error map turns store.ErrNotFound into 404.
    e.GET("/api/notes/" + uuid.New().String()).
        Expect().Status(httptest.StatusNotFound)
}
```

Notice `.Raw()`: most chain methods keep asserting, but `Raw()` steps out of the chain and hands you the underlying Go value: here the created id as a `string`, immediately reused to build the next request.

The Request and Assertion Chain

Requests are built by chaining, executed by `Expect()`, and verified by chaining again. The methods you will use constantly:

Building the request:

Method	Purpose
`e.GET/POST/PUT/PATCH/DELETE(path)`	start a request
`e.GET("/users/{id}", "42")`	fill path placeholders positionally
`WithJSON(v)`	JSON-encode `v` as the body, set the content type
`WithQuery("page", 2)`	add a query parameter
`WithHeader("Authorization", "Bearer "+tok)`	add a header
`WithBasicAuth(user, pass)`	add an `Authorization: Basic` header
`WithFormField(k, v)`	add a form field

`WithMultipart().WithFile(field, name)`	multipart file upload

Asserting on the response:

Method	Purpose
`Status(httptest.StatusOK)`	status code
`Body().IsEqual("ok")`, `Body().Contains("<h1>")`	raw body
`Header("X-Request-Id").NotEmpty()`	response header
`JSON().Object()`	parse the body as a JSON object
`JSON().Array()`	parse the body as a JSON array
`HasValue("name", "John")`	object has key with exact value
`ContainsKey("id")` / `NotContainsKey("password")`	key presence
`NotContainsValue(v)`	no key holds this value
`Value("address").Object()`	descend into a nested value
`Array().Length().IsEqual(3)` / `Array().NotEmpty()`	array size
`Array().Value(0).Object()`	descend into an element
`Raw()`	leave the chain, return the Go value

Typed value assertions hang off `Value`:

```go
obj := e.GET("/api/report").Expect().
    Status(httptest.StatusOK).JSON().Object()

obj.Value("count").Number().IsEqual(42)
obj.Value("active").Boolean().IsTrue()
obj.Value("tags").Array().NotEmpty()
```

By default all failed assertions in a test are collected and reported together. Pass `httptest.Strict(true)` to stop at the first failure instead, useful when later assertions only produce noise once the first one fails.

Fakes and One-Method Stubs

Because the repository is an interface, a test double is ordinary Go. The full in-memory `MemStore` covers most tests, but the sharpest tool is the one-method stub: embed the interface, override a single method, inherit everything else.

```go
import (
    stdContext "context"
    "fmt"
)

// failingRepo overrides one method of a working Repository; everything
// else is inherited. Interface embedding makes one-method fakes
// one-liners.
type failingRepo struct {
    store.Repository
}

func (failingRepo) Notes(stdContext.Context) ([]store.Note, error) {
    return nil, fmt.Errorf("pg: connection refused on 10.0.0.3:5432")
}

// Internal error details must never reach the client.
func TestInternalErrorsDoNotLeak(t *testing.T) {
    app := newApp(testConfig(), failingRepo{store.NewMemStore()})
    e := httptest.New(t, app)

    body := e.GET("/api/notes").
        Expect().Status(httptest.StatusInternalServerError).
        JSON().Object()
    body.HasValue("message", "server error")
    body.NotContainsValue("10.0.0.3") // stays server-side.
}
```

This is the test to internalize. Five lines of stub prove a property of the whole system: an unmapped error (a database outage, a bug) renders as a generic 500 while the real cause (with its connection string) is logged server-side, never serialized to the client (Chapter 9). No mocking framework, no code generation: the seam is an interface, so the stub is a struct literal.

Testing Guards and Authentication

Protected surfaces get three tests: no credentials, wrong credentials, valid credentials. With the in-memory client this is three requests:

```go
func TestAdminRequiresCredentials(t *testing.T) {
    app := newApp(testConfig(), store.NewMemStore())
    e := httptest.New(t, app)

    // No credentials: the basicauth middleware replies 401.
    e.GET("/api/admin/stats").Expect().
        Status(httptest.StatusUnauthorized)

    // Wrong credentials: still 401.
    e.GET("/api/admin/stats").WithBasicAuth("admin", "wrong").
        Expect().Status(httptest.StatusUnauthorized)

    // Valid credentials: through to the handler.
```

```
    e.GET("/api/admin/stats").WithBasicAuth("admin", "test-password").
        Expect().Status(httptest.StatusOK).
        JSON().Object().HasValue("notes", 0)
}
```

The password is not a secret here: `testConfig()` chose it. That is the `newApp(cfg, ...)` seam again: authentication is tested against the real middleware with credentials the test controls.

Token-protected routes follow the same pattern with a header instead:

```GO
e.GET("/api/profile").
    WithHeader("Authorization", "Bearer "+token).
    Expect().Status(httptest.StatusOK)
```

Generate `token` with the same signer configuration the application verifies with, a test-only key pair in the config, exactly like the test-only password (Chapter 10).

For guards you wrote yourself (the “shelf must exist”, “shelf must be unlocked” middlewares of Chapter 4), assert the pair of behaviors that defines a guard: the request it blocks and the request it lets through. A locked-resource guard, for example, gets one test proving writes return 403 and one proving reads still return 200.

Testing Validation and the Error Map

Do not stop at status codes: your API’s error body is a contract too (Chapter 9). Clients parse `http_error_code` and `validation`, so pin the wire shape in a test.

```GO
func TestValidationWireFormat(t *testing.T) {
    app := newApp(testConfig(), store.NewMemStore())
    e := httptest.New(t, app)

    body := e.POST("/api/notes").
        WithJSON(store.NoteInput{Text: ""}).
        Expect().Status(httptest.StatusBadRequest).
        JSON().Object()

    body.Value("http_error_code").Object().
        HasValue("canonical_name", "INVALID_ARGUMENT").
        HasValue("status", 400)
    body.Value("validation").Array().NotEmpty()
    body.Value("validation").Array().Value(0).Object().
        HasValue("field", "text")
}
```

The central error map deserves its own test per mapped sentinel: return the sentinel from the seam (or trigger it naturally, as here with an unknown id) and assert the canonical rendering:

```go
func TestNotFoundMapping(t *testing.T) {
    app := newApp(testConfig(), store.NewMemStore())
    e := httptest.New(t, app)

    e.GET("/api/notes/" + uuid.New().String()).
        Expect().Status(httptest.StatusNotFound).
        JSON().Object().
        Value("http_error_code").Object().
        HasValue("canonical_name", "NOT_FOUND").
        HasValue("status", 404)
}
```

These tests are cheap and catch a whole class of regressions: someone removing a `MapErrors` line, a sentinel silently replaced by a new error value, a handler that starts writing its own ad-hoc 404 body.

Testing Middleware in Isolation

To test one middleware, build a throwaway application around it: the middleware, one trivial handler, nothing else. The test then observes the middleware's effect from the outside, like a client would.

```go
package middleware_test

import (
    "testing"

    "github.com/kataras/iris/v14"
    "github.com/kataras/iris/v14/httptest"
)

func TestVersionHeader(t *testing.T) {
    app := iris.New()
    app.Use(func(ctx iris.Context) {
        ctx.SetHeader("X-API-Version", "v1")
        ctx.Next()
    })
    app.Get("/", func(ctx iris.Context) {
        ctx.Text("ok")
    })

    e := httptest.New(t, app)
    e.GET("/").Expect().Status(httptest.StatusOK).
        Header("X-API-Version").IsEqual("v1")

    // Root-level Use runs for every request, even unmatched ones —
    // the header appears on the 404 too (Chapter 7).
    e.GET("/missing").Expect().Status(httptest.StatusNotFound).
        Header("X-API-Version").IsEqual("v1")
}
```

A middleware that makes decisions gets the same blocked/allowed pair as any guard. Test the middleware you actually ship: the key check below is `apikey.New` from Chapter 12, not a hand-written one, so the test covers the code that will run in production:

```go
import (
    "testing"

    "github.com/kataras/iris/v14"
    "github.com/kataras/iris/v14/httptest"
    "github.com/kataras/iris/v14/middleware/apikey"
)

func TestAPIKeyGuard(t *testing.T) {
    app := iris.New()
    app.Use(apikey.New(apikey.Options{
        Header: "X-API-Key",
        Scheme: apikey.NoScheme, // a bare value, no "Bearer" prefix.
        Keys:   map[string]string{"reporting": "secret"},
    }))
    app.Get("/", func(ctx iris.Context) {
        principal, _ := apikey.Get(ctx)
        ctx.Text(principal)
    })

    e := httptest.New(t, app)

    // Blocked: no key, and a key that matches nothing.
    e.GET("/").Expect().Status(httptest.StatusUnauthorized)
    e.GET("/").WithHeader("X-API-Key", "nope").
        Expect().Status(httptest.StatusUnauthorized)

    // Allowed: the handler runs and can name the caller.
    e.GET("/").WithHeader("X-API-Key", "secret").
        Expect().Status(httptest.StatusOK).
        Body().IsEqual("reporting")
}
```

The third assertion is the one worth writing. A guard test that only checks 401 and 200 proves the door opens; asserting the body is `"reporting"` proves the request arrived carrying the caller's identity, which is what every downstream audit line and rate-limit bucket depends on.

Both tests are deterministic by construction: nothing waits, nothing depends on timing. Prefer designs like this over `time.Sleep`: a test that waits for wall-clock time is slow on every run and flaky on busy CI machines.

Unit Testing Domain Logic

Not everything needs an HTTP request. Domain rules (validation, calculations, state transitions) live in packages that import nothing from Iris, so they get plain Go tests, usually table-driven. “Table-driven” is the standard Go idiom for testing many cases without repeating the machinery: the cases are rows of data (a name, an input, an expected outcome) and one loop runs the same check over every row. Adding the next edge case is adding a line to the table, and `t.Run` gives each row its own name in the failure output:

```go
// store/store_test.go
package store

import (
    "strings"
    "testing"
)

func TestNoteInputValidate(t *testing.T) {
    tests := []struct {
        name    string
        input   NoteInput
        wantErr bool
    }{
        {"valid", NoteInput{Text: "buy milk"}, false},
        {"empty", NoteInput{Text: ""}, true},
        {"too long", NoteInput{Text: strings.Repeat("x", 281)}, true},
    }

    for _, tt := range tests {
        t.Run(tt.name, func(t *testing.T) {
            err := tt.input.Validate()
            if (err != nil) != tt.wantErr {
                t.Fatalf("Validate() error = %v, wantErr %v",
                    err, tt.wantErr)
            }
        })
    }
}
```

Keep the split intentional: unit tests pin the domain’s edge cases cheaply and exhaustively; the HTTP tests of the previous sections pin the wiring: that a failed `Validate` becomes a 400 with the right body. One wire-format test per input type is enough; the case matrix belongs here.

httptest Options and Variants

`httptest.New` accepts options after the application:

```go
e := httptest.New(t, app,
    httptest.Debug(true),           // print requests and responses
    httptest.LogLevel("debug"),     // re-enable application logs
    httptest.Strict(true),          // stop at the first failed assertion
    httptest.URL("http://example.com"), // base URL for the client
```

```
)
```

Option	Default	Effect
`Debug(bool)`	`false`	dumps each request/response through `t.Logf`
`LogLevel(string)`	`"disable"`	the application's logger level while testing
`Strict(bool)`	`false`	require-style reporting: fail fast
`URL(string)`	`""`	base URL; enables cookie handling for that host

The test client always carries a cookie jar, but the jar needs a host to file cookies under. When a flow depends on cookies surviving across requests (sessions, CSRF tokens), set a base URL (`httptest.URL("http://example.com")` or the shorthand `httptest.Domain`), and `Set-Cookie` responses are resent automatically on subsequent requests within the same `e`.

Four variants cover the remaining cases:

`httptest.NewController` builds a throwaway application around one controller, which is what a controller test wants and what `httptest.New` cannot give it: `New` takes a built application, and the unit under test is a `Routing` value.

```go
func TestNotesController(t *testing.T) {
    e := httptest.NewController(t, "/notes", &NotesAPI{Store: store.NewMemory()})

    e.GET("/notes").Expect().Status(httptest.StatusOK).
        JSON().Array().IsEmpty()
}
```

The variadic middleware is registered on the controller's own group, so it runs for its routes and not for the 404 and 405 chains: `httptest.NewController(t, "/notes", api, requireAdmin)`. A failure answers what a builder application answers, so the test asserts the body production sends. The path parameter type and context error handlers are installed, so a bad path parameter renders the `rest/errors` body. `errors.Handler` runs before the router, so a bare failure status, such as a `requireAdmin` that stops with 403, answers the JSON envelope and not an empty body. It is also the 404 handler, so an unknown path under the prefix answers the `NOT_FOUND` envelope. A method the controller does not serve answers that same 404, not a 405, because `Options.Router.FireMethodNotAllowed` is off by default and `NewController` takes no options. Nothing else the builder chain adds is there: no CORS, no access log, no recovery, no compression. When a test needs those, or needs `httptest.URL`, it builds the application itself and calls `New`.

`httptest.Do` executes a single handler with no application and no routing, using the standard recorder (the package aliases `httptest.NewRecorder` and `httptest.NewRequest` for convenience):

```go
func TestHandlerAlone(t *testing.T) {
    handler := func(ctx iris.Context) {
        ctx.JSON(iris.Map{"status": "ok"})
    }

    w := httptest.NewRecorder()
    r := httptest.NewRequest("GET", "/", nil)
    httptest.Do(w, r, handler)

    if w.Code != httptest.StatusOK {
        t.Fatalf("expected 200, got %d", w.Code)
    }
}
```

Because no routing happens, dynamic path parameters are never populated: `ctx.Params()` is empty. A handler that reads path parameters must be tested through a real application with `httptest.New`.

`httptest.NewServer(t, app)` builds the application and wraps it in a standard `net/http/httptest.Server`, a real listener on a random local port. Reach for it only when something genuinely needs a network endpoint, such as testing an HTTP client:

```go
srv := httptest.NewServer(t, app)
defer srv.Close()

resp, err := http.Get(srv.URL + "/api/notes")
```

`httptest.NewInsecure(t, ...)` returns a client that skips TLS certificate verification, for asserting against an already-running HTTPS server: `httptest.NewInsecure(t, httptest.URL("https://localhost:8443"))`.

Running the Suite

Nothing about Iris tests is special to the `go` tool:

```sh
go test ./...           # the whole module
go test -run CRUD ./... # a subset by test name
go test -race ./...     # with the race detector
go test -cover ./...    # with coverage
```

A word on `-race`: a *data race* is two goroutines (Go's lightweight concurrent threads) touching the same memory at the same time with at least one writing, and it corrupts data in ways that surface far from the cause, often only under production load. Go ships a *race detector* that

watches memory accesses while tests run and reports races with full stack traces the moment they happen.

Run `-race` in CI at minimum. Handlers execute concurrently in production, and shared fakes like `MemStore` are exactly where an unguarded map write hides; the race detector finds it while the suite is still milliseconds long.

Determinism is a design property, and the patterns of this chapter buy it for free, so keep them:

- **Fresh state per test.** Every test calls `newApp` with its own store. No shared globals means no ordering dependencies, and `t.Parallel()` is safe.
- **No ports.** `httptest.New` needs none; reserve `NewServer` for the rare test that truly requires a listener.
- **No sleeps.** The in-memory transport is synchronous: when `Expect()` returns, the handler has finished. If a test wants time to pass, inject a clock through the same seam as every other dependency instead of waiting for the real one.
- **In-memory stores by default.** Real databases belong to a separate, explicitly tagged integration suite (Chapter 13); the everyday suite must run anywhere `go test` does.

Summary

- `httptest.New(t, app)` builds the application (failures reach `t.Fatalf`), binds an httpexpect client directly to it (no ports, no network) and silences logs. Options: `Debug`, `LogLevel`, `Strict`, `URL`.
- Requests chain builders (`WithJSON`, `WithQuery`, `WithHeader`, `WithBasicAuth`), execute with `Expect()`, and assert with `Status`, `JSON().Object()/Array()`, `HasValue`, `Value`, `Length`, `Body`, `Header`; `Raw()` returns the underlying Go value.
- Testability is an architecture decision: `newApp(cfg, deps)` takes its seams as parameters, so tests run the real router, middleware and error map with fakes only at the edges.
- Interface embedding turns one-method stubs into one-liners; the `failingRepo` test proves internal error details never leak to clients.
- Test guards as pairs (blocked and allowed), pin the validation and error-map wire format, and test middleware by building a tiny application around it.
- Domain logic gets plain table-driven Go tests; `go test ./...` runs everything in milliseconds, `-race` in CI, no sleeps anywhere.

Further Reading

- Go tutorial: Add a test: the official first-test walkthrough; start here if `go test` itself is new.
- The Go testing package: `*testing.T`, subtests with `t.Run`, `t.Parallel`, benchmarks and fuzzing.
- The Go net/http/httptest package: the standard library's recorder and test server that Iris's `httptest` builds upon.
- httpexpect: the full assertion-chain API behind the client `httptest.New` returns.
- The Go Data Race Detector: what `-race` catches, its cost, and how to read its reports.

CHAPTER 17

Monitoring & Observability

An application you cannot observe is an application you cannot operate. Iris ships the observability essentials in-tree: a structured access log, request correlation IDs, an application logger, a live process dashboard, the Go profiling endpoints, per-request cost measurement, `Server-Timing` responses and a health endpoint that reports build information. This chapter wires each of them the way you would in production, then closes with an honest look at where third-party tools (Prometheus, OpenTelemetry) take over.

Background

The day you deploy, your relationship with your own code changes. On your machine, when something misbehaves, you attach a debugger, set a breakpoint and inspect variables at leisure. In production none of that exists: the process runs on a remote machine, serves real users and cannot be paused. By the time someone reports a problem, the moment that caused it is minutes or days gone. The only way to understand it is through what the application recorded *while it happened*. Observability is the discipline of emitting that evidence deliberately (before you need it) instead of hoping to reconstruct events from memory.

The evidence comes in three shapes, and the industry uses three words for them:

- **Logs** are timestamped records of individual events: "this request returned 500 at 03:12:07", "user 42 signed up". They answer *what exactly happened*. A *structured* log records each entry as key-value data (in practice, one JSON object per line) instead of free-form prose, so tools can filter and index it rather than merely display it.
- **Metrics** are numbers tracked over time: requests per second, memory in use, the error rate. A metric forgets individual events and keeps only the aggregate, which makes it tiny to store and ideal for graphs and alerts. Metrics answer *how the system is trending*.
- **Traces** follow one request across every service it touches, with timing for each step: this instance, then the database, then a downstream API. Traces answer *where the time went*

when a single operation spans several processes.

A rough rule: metrics tell you something is wrong, logs tell you what, traces tell you where.

Two more terms carry this chapter. A *health check* is a cheap endpoint whose only job is to be polled by machines. The poller is typically an *orchestrator*, supervising software such as a load balancer or Kubernetes that starts, stops and routes traffic to your instances. A failing *liveness* check tells the orchestrator "this process is dead, restart it"; a failing *readiness* check tells it "this instance cannot serve right now, send traffic elsewhere", a distinction the health section returns to.

Finally, request IDs. One production request leaves fragments everywhere: a line in the access log, several in the application log, entries on the database and on downstream services, all interleaved with the fragments of thousands of neighboring requests. When a user reports one failure, finding its fragments by timestamp is archaeology. A *request ID* (or correlation ID) is a unique value stamped onto every record a request produces, across every system it touches. Debugging at 3am then starts with one search: filter all logs by the ID from the user's error message, and the complete story of that single request reads out in order. Little in this chapter is glamorous, but this one habit routinely turns hours of guessing into minutes of reading.

Structured Access Logging

Chapter 12 introduced `middleware/accesslog` as a one-liner. In production the access log is the backbone of your observability: one record per request (latency, status, sizes, custom fields) written as JSON lines that any log shipper can tail, parse and index. The configuration below is tuned for exactly that:

```go
package main

import (
    "github.com/kataras/iris/v14"
    "github.com/kataras/iris/v14/middleware/accesslog"
    "github.com/kataras/iris/v14/middleware/requestid"
)

func newAccessLog() *accesslog.AccessLog {
    ac := accesslog.File("access_log.json")

    // Fire the log line in a goroutine on a copy of the request
    // context, so the response never waits on log I/O.
    ac.Async = true

    // Keep payloads out of production logs: privacy and volume.
    ac.RequestBody = false
    ac.ResponseBody = false
```

```
    // Body sizes are cheap to record. The full wire sizes
    // (BytesReceived/BytesSent) dump the whole request and response
    // to count headers too — expensive, leave them off.
    ac.BytesReceivedBody = true
    ac.BytesSentBody = true

    // One line per record, even when a panic is logged.
    ac.KeepMultiLineError = false
    ac.PanicLog = accesslog.LogHandler

    // Compact JSON: one object per line.
    ac.SetFormatter(&accesslog.JSON{})

    // Attach the correlation ID to every record.
    ac.AddFields(func(ctx iris.Context, f *accesslog.Fields) {
        f.Set("request_id", requestid.Get(ctx))
    })

    return ac
}

func main() {
    app := iris.New()

    ac := newAccessLog()
    defer ac.Close()

    app.UseRouter(ac.Handler)
    app.Use(requestid.New(requestid.ServerOnlyGenerator))

    app.Get("/", func(ctx iris.Context) {
        ctx.JSON(iris.Map{"hello": "world"})
    })

    app.Listen(":8080")
}
```

Register the handler with `UseRouter` so the log also captures requests that never match a route: 404s, CORS preflights, rejected requests. `accesslog.File(path)` opens a buffered file (flushed on `Close`), `accesslog.FileUnbuffered(path)` trades throughput for immediate writes, and `accesslog.New(w io.Writer)` accepts any writer, the hook for log rotation below.

The Record

With the JSON formatter each line carries these keys (empty ones are omitted): `timestamp` (unix milliseconds), `latency`, `code`, `method`, `path`, `ip`, `query`, `params`, `fields`, `request`, `response`, `bytes_received`, `bytes_sent`. The `query`, `params` and `fields` keys hold arrays of `{"key": ..., "value": ...}` entries, the same shape in compact and indented output. Without a formatter the default text format is:

```
Time|Latency|Code|Method|Path|IP|Path Params Query Fields|Bytes Received|Bytes Sent|Request|Response|
```

Three formatters ship with the package, all set through `SetFormatter`:

```go
// JSON lines. A non-empty Indent pretty-prints instead;
// HumanTime replaces the unix timestamp with TimeFormat text.
ac.SetFormatter(&accesslog.JSON{})

// CSV, optionally with a header row on a fresh file.
ac.SetFormatter(&accesslog.CSV{Header: true})

// Full control via text/template.
ac.SetFormatter(&accesslog.Template{
    Text: "{{.Now.Format .TimeFormat}} {{.Code}} {{.Method}} " +
        "{{.Path}} {{.Latency}}\n",
})
```

The default text format, `CSV` and `Template` write one line per record. In every value a client controls (the method, the path, the IP, the query, the path parameters, the fields and both bodies) a line break or a tab becomes a space and any other control character is dropped, so a request to a path with `%0A` in it cannot start a second log line. A multi-line error or panic trace is one line there too, whatever `KeepMultiLineError` says. The option still matters for JSON, which keeps a record on one line either way: set to false, the line breaks of an error text are spaces in the logged string, not `\n` escapes. Two limits in a template: a field or path parameter value that is not a string is replaced by its sanitized text when that text held a control character, and anything read through `.Ctx` is printed as it is.

A custom formatter is any type implementing `accesslog.Formatter` (`SetOutput(io.Writer)` plus `Format(*accesslog.Log) (bool, error)`). It receives the values as the request sent them.

Custom Fields

Instance-level fields are computed for every request through `AddFields`, as in the `request_id` example above. Handlers add per-request fields through `accesslog.GetFields`:

```go
func chargeHandler(ctx iris.Context) {
    fields := accesslog.GetFields(ctx)
    fields.Set("tenant", ctx.Params().GetString("tenant"))
    fields.Set("items_charged", 3)

    ctx.JSON(iris.Map{"ok": true})
}
```

Fields land in the record's `fields` array as `{"key": ..., "value": ...}` entries, so your log pipeline can index them like any other attribute.

Skipping Noise and Logging Only Failures

Health checks and static assets can drown a log. Skip them per route with `accesslog.SkipHandler` as a middleware, or conditionally with `accesslog.Skip(ctx)`:

```go
app.Get("/ready", accesslog.SkipHandler, readyHandler)

app.Get("/assets/{file:path}", func(ctx iris.Context) {
    accesslog.Skip(ctx)
    // serve the asset...
})
```

Setting `ac.OnlyFailures = true` logs only requests that end with a status code of 400 or above, a useful mode for a second, dedicated error log alongside your metrics.

Shutdown Semantics

`Close` flushes buffers, waits up to ten seconds for pending asynchronous lines (`CloseContext` accepts a custom deadline) and closes the underlying file. In manual apps, `defer ac.Close()`; the middleware also registers its own `Close` on the interrupt signal. With the builder, `LogRequestsWith` registers the closer for you:

```go
ac := newAccessLog()

app := iris.NewBuilder().
    Prefix("/api").
    AllowOrigin("none").
    Compression(true).
    LogRequestsWith(ac). // your tuned instance, closed on shutdown
    Health(true, "production", "ops@example.com").
    Errors(errors.NewOptions()).
    Deferrables().
    Services( /* your services */ ).
    RouterMiddlewares(requestid.New(requestid.ServerOnlyGenerator)).
    Middlewares().
    API("/orders", NewOrdersAPI).
    Build()
```

`LogRequests(true)` installs the framework's tuned default instead: convenient in development, but it logs request and response bodies, which is rarely what production wants. Prefer `LogRequestsWith` with a configuration like the one above.

Log Rotation

The access log writes to whatever `io.Writer` you give it, so rotation plugs in at the writer level. With the third-party `gopkg.in/natefinch/lumberjack.v2` package (clearly a third-party choice, any rotating writer works):

```go
import "gopkg.in/natefinch/lumberjack.v2"

ac := accesslog.New(&lumberjack.Logger{
    Filename:   "/var/log/myapp/access_log.json",
    MaxSize:    100, // megabytes per file
```

```
    MaxBackups: 7,
    MaxAge:     28, // days
    Compress:   true,
})
```

If an external rotator (logrotate, a sidecar) truncates the file instead, the middleware exposes `ac.TruncateFile(0)` to clear the current file in place without reopening it.

Live Tail with the Broker

The broker broadcasts every `accesslog.Log` to registered listeners, the building block for a live log view over SSE or WebSocket, or for an in-process alerting hook. Create it before the server starts:

```go
broker := ac.Broker()

go func() {
    logs := broker.NewListener() // read-only channel of accesslog.Log
    for log := range logs {
        if log.Code >= 500 {
            // push to an alert channel, a websocket, a counter...
        }
    }
}()
```

Call `broker.CloseListener(logs)` when a consumer goes away; `Close` on the access log terminates all listeners.

The bare `go func()` above is the shortest way to show the channel. In a builder application, wrap the loop in an `iris.Worker` instead (the `CPUAlerter` further down this chapter is the shape) and pass it to `Services`: the listener then starts after the initializers, stops on the shutdown context, and `Shutdown` waits for it, none of which a loose goroutine gives you.

Request Correlation

One request produces many log lines: the access log record, application log entries, database logs, entries on downstream services. A correlation ID ties them together. The `requestid` middleware assigns one per request, stores it on the context and reflects it in the `X-Request-Id` response header:

```go
import "github.com/kataras/iris/v14/middleware/requestid"

app.Use(requestid.New())

app.Get("/", func(ctx iris.Context) {
```

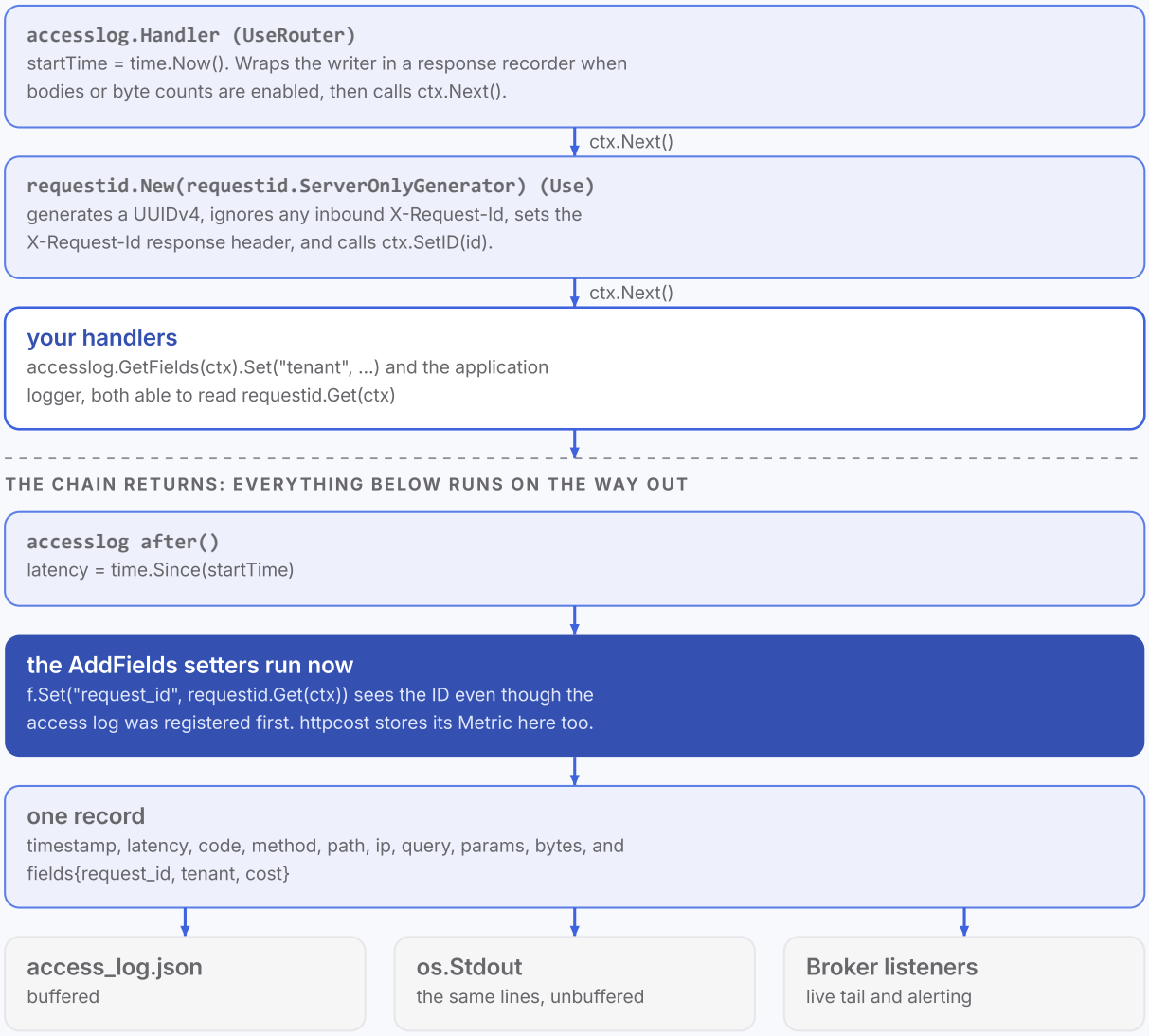
```
    id := requestid.Get(ctx) // "" if the middleware did not run
    ctx.JSON(iris.Map{"request_id": id})
})
```

The default generator honors a client-supplied `X-Request-Id` header and generates a UUIDv4 only when it is missing. Behind a trusted proxy that assigns IDs, that is exactly right. Facing untrusted clients it is a log injection vector: anyone can choose the value that lands in your logs. When IDs must be unforgeable, generate them server-side only:

```
app.Use(requestid.New(requestid.ServerOnlyGenerator))
```
GO

Because the ID travels in the response header, a client (or your own outgoing HTTP calls, by setting the same header on downstream requests) can quote it in bug reports, and support can grep every system for it. The access log picks it up through the `AddFields` setter shown earlier; your application logger should include it too, which is where the next section comes in.

That setter is worth a second look, because at first glance it should not work. The access log is registered with `UseRouter`, *before* `requestid.New`, so when its handler runs there is no ID yet. The reason it sees one anyway is that field setters do not run on the way in. They run in the access log's second pass, after the chain has returned:



The access log starts its timer before the chain and writes its record after it, and because its field setters run in that second pass they can read a request ID that a later middleware generated.

Application Logging

The access log records what happened to requests; the application logger records what your code decided along the way. Every Iris application carries a golog logger, reachable as `app.Logger()` at startup and `ctx.Application().Logger()` inside handlers:

```go
package main
```

```
import (
    "os"

    "github.com/kataras/golog"
    "github.com/kataras/iris/v14"
)

func main() {
    app := iris.New()

    logger := app.Logger()
    logger.SetLevel("info") // "fatal","error","warn","info","debug"
    logger.SetTimeFormat("2006-01-02 15:04:05")

    // Keep stdout and add a file output.
    f, err := os.OpenFile("app.log",
        os.O_CREATE|os.O_WRONLY|os.O_APPEND, 0o600)
    if err != nil {
        logger.Fatalf("open log file: %v", err)
    }
    defer f.Close()
    logger.AddOutput(f)

    app.Get("/signup", func(ctx iris.Context) {
        ctx.Application().Logger().Infof("user signed up",
            golog.Fields{
                "plan":   "free",
                "source": "landing-page",
            })
        ctx.JSON(iris.Map{"ok": true})
    })

    app.Listen(":8080")
}
```

The leveled methods come in pairs (`Debug`/`Debugf`, `Info`/`Infof`, `Warn`/`Warnf`, `Error`/`Errorf`, `Fatal`/`Fatalf`), and any `golog.Fields` value passed among the arguments becomes structured fields on the entry rather than format arguments. To emit the application log itself as JSON (so it flows through the same pipeline as the access log), set the formatter:

```
app.Logger().SetFormat("json")
```
GO

`SetOutput(w)` replaces all outputs with one writer, `AddOutput(w...)` appends. At `SetLevel("debug")` Iris also prints route registrations and other framework internals: useful in development, noisy in production; `"info"` is the production default worth keeping.

The Monitor Dashboard

The `middleware/monitor` package samples process and operating system statistics (CPU, RAM, TCP connections, load average) on a fixed interval and serves them two ways: as JSON and as a self-refreshing HTML dashboard. The dashboard page fetches its numbers with a `POST` request to its own URL, so the two handlers mount on the same path:

```go
package main

import (
    "time"

    "github.com/kataras/iris/v14"
    "github.com/kataras/iris/v14/middleware/basicauth"
    "github.com/kataras/iris/v14/middleware/monitor"
)

func main() {
    app := iris.New()

    m := monitor.New(monitor.Options{
        RefreshInterval:     2 * time.Second, // stats collection
        ViewRefreshInterval: 2 * time.Second, // dashboard refresh
        ViewTitle:           "My App",
    })
    iris.RegisterOnInterrupt(m.Stop)

    admin := app.Router("/admin", basicauth.Default(map[string]string{
        "admin": "change-me",
    }))
    admin.Get("/monitor", m.View)   // HTML dashboard
    admin.Post("/monitor", m.Stats) // JSON stats (used by the view)

    app.Listen(":8080")
}
```

Three practical notes. First, `RefreshInterval` left unset defaults to `monitor.DefaultRefreshInterval` (2 seconds), so a zero-value `monitor.Options{}` is safe; set it yourself when the collection cadence matters. Second, the dashboard exposes live host metrics: mount it behind authentication, as above, never publicly. Third, stop the collector on shutdown: `iris.RegisterOnInterrupt(m.Stop)` in manual apps, `Deferrables(m.Stop)` in builder apps.

`Options` also accepts `PID` (defaults to the current process) and `ViewAnimationInterval` for the chart animation.

Reading the Stats Programmatically

The same numbers the dashboard renders are available in code, which turns the monitor into a simple in-process alerting source. A sampling loop is long-running work, so it belongs in an `iris.Worker` rather than a goroutine started next to the server: the builder starts `Run` after the

initializers, cancels its context at shutdown, and waits for it to return (Chapter 3). A bare `go func()` gets none of that, and a `time.Tick` inside one leaks its ticker for the life of the process.

```go
// CPUAlerter watches the numbers the dashboard renders and warns when
// the host is hot.
type CPUAlerter struct {
    Monitor  *monitor.Monitor
    Logger   *golog.Logger
    Interval time.Duration
}

func (a *CPUAlerter) Run(ctx context.Context) error {
    ticker := time.NewTicker(a.Interval)
    defer ticker.Stop()

    for {
        select {
        case <-ctx.Done():
            return nil // shutdown: stop sampling.
        case <-ticker.C:
            if stats := a.Monitor.Holder.GetStats(); stats.OSCPU > 90 {
                a.Logger.Warnf("high CPU: %.1f%%", stats.OSCPU)
            }
        }
    }
}

// Stop releases what Run acquired. The ticker is Run's own, so there is
// nothing left to release here.
func (a *CPUAlerter) Stop(context.Context) error { return nil }
```

Register it with the monitor it reads:

```go
Deferrables(m.Stop).
Services(&CPUAlerter{
    Monitor:  m,
    Logger:   golog.Default,
    Interval: 30 * time.Second,
}).
```

`monitor.Stats` carries `PIDCPU`, `PIDRAM`, `PIDConns` for the process and `OSCPU`, `OSRAM`, `OSTotalRAM`, `OSLoadAvg`, `OSConns` for the host. Each value is also published through the standard `expvar` package under `pid_cpu`, `pid_ram`, `pid_conns`, `os_cpu`, `os_ram`, `os_total_ram`, `os_load_avg` and `os_conns`; see the third-party section for exposing them over HTTP.

Profiling with pprof

A profiler watches a running program and reports where its resources actually go, attributed to functions: which code burns the CPU, which allocates the memory, where goroutines sit blocked. Go's profiler, pprof, is built into the runtime and works by sampling (briefly recording call stacks many times per second), so it is safe to leave available in production and enable on demand.

When the monitor tells you *that* CPU or memory is high, the Go profiler tells you *why*. The `middleware/pprof` package mounts the standard `net/http/pprof` endpoints as a routing controller. Profiles reveal source file names and execution details, so guard them exactly like the monitor:

```go
package main

import (
    "github.com/kataras/iris/v14"
    "github.com/kataras/iris/v14/middleware/basicauth"
    "github.com/kataras/iris/v14/middleware/pprof"
)

func main() {
    app := iris.New()

    auth := basicauth.Default(map[string]string{"ops": "change-me"})
    app.Router("/debug/pprof", auth, pprof.New())

    app.Listen(":8080")
}
```

`GET /debug/pprof` renders an index of the available profiles (heap, goroutine, allocs, block, mutex, threadcreate, cmdline, profile, trace). The interesting ones connect straight to the Go toolchain:

```sh
go tool pprof http://ops:change-me@localhost:8080/debug/pprof/profile


go tool pprof http://ops:change-me@localhost:8080/debug/pprof/heap


curl -u ops:change-me -o trace.out \
    "http://localhost:8080/debug/pprof/trace?seconds=5"
go tool trace trace.out
```

An alternative to permanent credentials is registering the routes only when a debug flag or environment variable is set, so production builds do not carry the endpoints at all. Chapter 20 walks through interpreting the profiles.

Per-Request Cost Measurement

The `middleware/httpcost` package answers a question most stacks cannot: what does one request cost? It measures elapsed time, CPU time, memory allocation delta and bytes written, then folds them into a single number using per-unit prices you define, the basis for usage-based billing, tenant showback or simply spotting expensive endpoints:

```go
package main

import (
    "github.com/kataras/iris/v14"
    "github.com/kataras/iris/v14/middleware/httpcost"
)

func main() {
    app := iris.New()

    app.Use(httpcost.New(httpcost.Options{
        CostPerSecond: 0.01,  // per wall-clock second
        CostPerMB:     0.001, // per MB of RAM and network
        CostPerCPU:    0.02,  // per CPU-second
    }))

    app.Get("/reports", func(ctx iris.Context) {
        ctx.JSON(iris.Map{"report": "..."})
    })

    app.Listen(":8080")
}
```

`httpcost.DefaultOptions` carries those same values; setting a price to zero disables that dimension, and at least one must remain enabled. The middleware stores an `*httpcost.Metric` on the context *after* the handler chain returns, so read it from something that wraps the chain; the access log's field setters are the natural place:

```go
ac.AddFields(func(ctx iris.Context, f *accesslog.Fields) {
    if metric, ok := httpcost.Get(ctx); ok {
        f.Set("cost", metric.Cost)
        f.Set("cpu_time", metric.CpuTime)      // seconds
        f.Set("memory_used", metric.MemoryUsed) // bytes
    }
})
```

The metric also carries `StartTime`, `Duration` and `NetworkUsed` (the bytes written to the response). Be honest about precision: CPU and memory are sampled process- and system-wide, so under concurrency each request is attributed a share of everything that ran at the

same time. Treat the numbers as consistent approximations, not an invoice-grade meter. Outside HTTP, the same measurement runs over any function:

```go
cost := httpcost.Compute(func() {
    // an import job, a report build...
})
```

Server-Timing

The `middleware/servertiming` package emits the standard `Server-Timing` response header, which browser devtools render as a waterfall of your server-side phases, free instrumentation for anyone with the network tab open:

```go
import "github.com/kataras/iris/v14/middleware/servertiming"

app.Use(servertiming.Handler)

app.Get("/dashboard", func(ctx iris.Context) {
    servertiming.Start(ctx, "db", "Load records")
    // ... query the database ...
    servertiming.End(ctx, "db")

    servertiming.Start(ctx, "render", "Build payload")
    // ... assemble the response ...
    servertiming.End(ctx, "render")

    ctx.JSON(iris.Map{"data": "..."})
})
```

`servertiming.Handler` records the total as an `app` metric on its own. Named metrics are optional refinement: every `Start` must be matched by an `End` with the same name, and the header is written once all started metrics have ended, before any body reaches the client. Timings are visible to whoever receives the response, so keep metric names and descriptions free of internal detail on public endpoints.

Health Checks and Build Information

Load balancers, container orchestrators and uptime monitors all need a cheap endpoint that answers “is this process alive, and which build is it?”. With the builder, one step provides it:

```go
Health(true, "production", "ops@example.com")
```

This registers `GET /health` at the root (deliberately outside the `Prefix`), handled by the `middleware/modrevision` package. The response is plain text: the server name and environment, the git revision and build time when available, cloud container details when detected, and a final `OK`. The revision and time come from `core/pkg/buildinfo`, which reads the `vcs.revision` and `vcs.time` stamps Go embeds when you `go build` inside a version-controlled checkout; `go run` or `-buildvcs=false` builds simply omit them.

Outside the builder, mount the same handler yourself with full control over its options:

```go
import "github.com/kataras/iris/v14/middleware/modrevision"

app.Get("/health", modrevision.New(modrevision.Options{
    ServerName:     "Orders API",
    Env:            "production",
    Developer:      "ops@example.com",
    UnixTime:       false,
    CloudDetection: true,
}))
```

The endpoint reveals environment and contact information by design. If that is more exposure than your security policy allows, use `NoHealth()` in the builder chain and register a terser endpoint of your own.

Liveness Versus Readiness

`/health` is a liveness check: it answers as long as the process serves HTTP, dependencies aside, exactly what a restart-deciding orchestrator should watch. A readiness check asks a different question (“should this instance receive traffic right now?”) and must consult dependencies. That is plain handler code:

```go
import (
    "database/sql"

    "github.com/kataras/iris/v14/rest/errors"
)

func readyHandler(db *sql.DB) iris.Handler {
    return func(ctx iris.Context) {
        if err := db.Ping(); err != nil {
            // 503 UNAVAILABLE, with the ping error logged and not sent.
            errors.Unavailable.LogErr(ctx, err)
            return
        }
        ctx.OK(iris.Map{"status": "ok"}, nil)
    }
}

app.Get("/ready", accesslog.SkipHandler, readyHandler(db))
```

Keep readiness checks fast and side-effect free (they run every few seconds) and skip them in the access log as shown, or they become the loudest client you have.

Third-Party Metrics and Tracing

Everything above is in-tree. Two observability pillars deliberately are not: time-series metrics (Prometheus and friends) and distributed tracing (OpenTelemetry). Iris ships no exporter or integration layer for them, and this section shows no invented one. What Iris does provide is `iris.FromStd`, which mounts any standard `net/http` handler. That is all the glue these ecosystems need.

Prometheus is the de facto standard metrics system: a separate server that stores time series (numbers with timestamps) and evaluates alert rules over them. It works by *pull*: on a fixed interval it fetches ("scrapes") a plain-text endpoint on your application, conventionally `/metrics`, where your process publishes its current counters and gauges. Your application never sends anything; it only answers when scraped.

Exposing a Prometheus scrape endpoint with the official client library (`github.com/prometheus/client_golang`):

```go
import (
    "github.com/prometheus/client_golang/prometheus/promhttp"

    "github.com/kataras/iris/v14"
)

app.Any("/metrics", iris.FromStd(promhttp.Handler()))
```

Instrumentation itself is ordinary middleware: register counters and histograms with the Prometheus client and observe them in an `app.Use` handler around `ctx.Next()`, using `ctx.GetStatusCode()`, `ctx.Path()` and your own timer. No Iris-specific API is involved, so the client library's documentation applies as-is.

Go's standard library has a minimal cousin of this idea: `expvar` lets any package publish named variables, and its handler serves them all as one JSON document over HTTP. The endpoint mounts the same way, and it is worth mounting when the monitor middleware runs, because the monitor publishes its gauges through `expvar`:

```go
import "expvar"

app.Any("/debug/vars", iris.FromStd(expvar.Handler()))
```

For distributed tracing, the OpenTelemetry Go SDK instruments at the `net/http` layer; anything it produces as an `http.Handler` mounts through `iris.FromStd` the same way, and its context propagation flows through the standard request context that Iris hands to your code. Protect `/metrics` and `/debug/vars` like the monitor and profiler: behind authentication or reachable only from your internal network.

Summary

- `accesslog` is the backbone: JSON lines to a file, `Async` on, bodies off, custom fields for the request ID and anything else your pipeline indexes. Register with `UseRouter`, close on shutdown, plug a rotating writer into `accesslog.New` for rotation. `LogRequestsWith(ac)` is the builder slot for a tuned instance.
- `requestid` correlates every log line of a request; `ServerOnlyGenerator` keeps the IDs unforgeable.
- `app.Logger()` is a leveled golog logger; `golog.Fields` adds structure and `SetFormat("json")` matches the access log's pipeline.
- `monitor` serves live process/OS stats as JSON and as a dashboard: `GET` and `POST` on the same path, behind admin authentication, with a positive `RefreshInterval` and `Stop` on shutdown.
- `pprof` mounts the Go profiler at `/debug/pprof`; `httpcost` prices each request; `servertiming` surfaces server phases in browser devtools.
- `Health(true, env, dev)` registers `GET /health` at the root via `modrevision`, reporting the git revision and build time embedded by `go build`; readiness checks that ping dependencies are plain handlers.
- Metrics and tracing belong to Prometheus and OpenTelemetry; `iris.FromStd(promhttp.Handler())` is the only glue required.

Further Reading

- OWASP Logging Cheat Sheet: what to log, what never to log, and how to keep attacker-controlled input from corrupting your logs.
- The Go net/http/pprof package: the standard profiling endpoints that `middleware/pprof` mounts.
- The Go expvar package: the standard mechanism behind `/debug/vars` and the monitor's published gauges.

- MDN: Server-Timing: the header format and how browser devtools display it.
- Prometheus Go client: counters, gauges and histograms for the `/metrics` endpoint shown in this chapter.
- OpenTelemetry Go SDK: the tracing SDK whose `net/http` instrumentation mounts through `iris.FromStd`.

CHAPTER 18

Security Best Practices

Many of Iris v14's defaults are secure ones: session cookies carry `Secure` and `SameSite=Lax`, JWT verification reads only the `Authorization` header, and sign-in redirects are confined to same-site paths. A few are permissive instead, for compatibility, and each has a one-line opt-in. This chapter tours both, then covers the hardening you enable yourself: transport security, request bounding, input validation, private errors, session and cookie policy, authentication hardening, CSRF defense, static file safety, and secrets hygiene.

Background

Web security begins with one assumption: every part of an incoming request is chosen by the sender, and the sender may be hostile. The method, the path, every header, every cookie value, the body, an uploaded file's name and content: all of it is attacker-controlled until your code proves otherwise. "Validate at the edge" and "bound the request", two phrases this chapter repeats, are that assumption turned into practice.

Most attacks on a web application fall into a handful of families:

- **Injection**: attacker input is spliced into a command, most famously a SQL query, so that what you meant to treat as data is executed as code instead.
- **Cross-site scripting (XSS)**: an attacker's JavaScript ends up inside a page you serve, running in other users' browsers with access to their session.
- **Cross-site request forgery (CSRF)**: a hostile page tricks a signed-in victim's browser into sending a state-changing request to your application, riding on cookies the browser attaches automatically.
- **Credential attacks**: passwords leaked from other sites are replayed against your sign-in form, or your own password database is stolen and brute-forced offline.
- **Denial of service (DoS)**: a client exhausts your memory, CPU or connections (a gigabyte body, a flood of requests) until real users cannot get through.

Two terms recur throughout the chapter. A **cookie** is a small piece of data the server asks the browser to store; the browser then attaches it to every later request to the same site automatically, which is what makes sessions work, and also what makes CSRF possible. **TLS** (the protocol behind `https://`) encrypts the connection between browser and server and proves the server's identity, so nothing on the network path can read or tamper with the traffic.

Because no single control is perfect, security relies on **defense in depth**: independent layers arranged so that one failing does not collapse the rest. A typed `{id:uuid}` route parameter, a query that keeps data separate from SQL text, and a least-privilege database account all guard against the same injection, and any one of them alone would still hold.

Finally, defaults matter more than options. A protection that must be switched on is a protection that is sometimes forgotten; a protection that is on from the start fails safe. That is why this chapter opens with what Iris already does for you, and why every escape hatch is a deliberate, visible line of code rather than silent configuration drift.

Secure by Default

Several protections are active in every Iris application without any configuration:

Area	Default behavior
Session cookies	`sessions.NewOptions` sets `Secure: true` and `SameSite: Lax`
Response cookies	`ctx.AddCookie` sets `HttpOnly` and `SameSite=Lax`
JWT extraction	`jwt.NewVerifier` reads the `Authorization` header only
CORS credentials	`Access-Control-Allow-Credentials: true` is never sent with a wildcard origin. It **is** sent with a reflected specific origin, which is what `cors.New()` produces: see CORS
Sign-in redirects	`auth.Auth.SigninHandler` follows only same-site relative paths
OAuth2 codes	authorization and device codes are 128-bit URL-safe opaque tokens
Builder apps	panics are recovered and failures render generic error bodies

Each default has a deliberate, one-line escape hatch for the cases that genuinely need different behavior:

- **Plain HTTP in local development.** A `Secure` cookie is never sent over `http://localhost`, so sessions appear to "not stick". Set `opts.Secure = false` on your `sessions.Options` for local runs only. In production behind a TLS-terminating proxy there is nothing to change: the browser connection is HTTPS even when the application sees HTTP.
- **Tokens in the URL.** The `?token=` query extractor is off because URLs leak into access logs, proxy logs, browser history and `Referer` headers. Call `verifier.WithQueryToken()` only when a header cannot be set, for example browser-initiated file downloads.
- **Credentials with a wildcard origin.** Browsers reject that combination, so the CORS middleware does not emit it. Credentialed CORS works normally for specific origins. `cors.New().AllowCredentials()` opts in for non-browser clients that need it anyway. Note the limit of this guarantee: it covers the literal `*` only, and `cors.New()` defaults to reflecting whatever origin asked. Use `cors.Strict()` and name your origins, as the CORS section describes.
- **External sign-in redirects.** `redirect_to` / `redirect_uri` values that are absolute URLs, protocol-relative (`//host`) or backslash-obfuscated are ignored and the token is returned as JSON instead, which closes an open-redirect vector. Perform external redirects in your own handler after validating the destination against your own allowlist.

The rest of the chapter covers the protections you enable yourself.

Transport Security

TLS needs two things: a private key and a **certificate**, a signed statement from a certificate authority (CA) that the key really belongs to your domain, which is what the browser verifies before showing the padlock. You can buy certificates, generate them internally, or obtain them for free from Let's Encrypt, an automated public CA. Everything in this section is about getting those two pieces into the server.

Serve TLS directly with `iris.ServerOptions`:

```go
package main

import (
    "github.com/kataras/iris/v14"
)

func main() {
    app := iris.New()

    app.Get("/", func(ctx iris.Context) {
        ctx.WriteString("served over TLS")
    })
```

```
    app.Run(iris.ServerOptions{
        Addr:     ":443",
        CertFile: "cert.pem",
        KeyFile:  "key.pem",
    })
}
```

For automatic certificates from Let's Encrypt, set `Letsencrypt: true` and list the domains the certificate covers. The first `Hosts` entry is the root domain:

```go
app.Run(iris.ServerOptions{
    Addr:        ":443",
    Letsencrypt: true,
    Hosts:       []string{"example.com", "www.example.com"},
})
```

Let's Encrypt validates each domain with the tls-alpn-01 challenge, which this same listener answers. Its validator connects to port 443, so that port has to reach the server; no port 80 handler is involved.

When `Letsencrypt` is enabled and `CertFile` / `KeyFile` are also provided, the files act as a fallback if the Let's Encrypt challenge fails, which is useful for local testing. For full control over protocol versions and cipher suites, supply your own `TLSConfig *tls.Config` (with certificates loaded into it) and leave `CertFile`, `KeyFile` and `Letsencrypt` unset.

Two practical notes:

- `app.Run` starts a single listener. There is no automatic port-80 listener redirecting HTTP to HTTPS; if you need that redirect, run a separate plain HTTP server or let your reverse proxy handle it.
- Terminating TLS at a proxy (nginx, a cloud load balancer) is equally valid. Secure cookies keep working because the browser's connection is HTTPS.

Bounding the Request

Untrusted request bodies are unbounded unless you cap them. Set `BodyMaxMemory` to limit every body-reading call (`Body`, `ReadJSON`, `ReadJSONStream` and `ReadMultipartRelated`) application-wide. It also bounds decompression-bomb reads when the compress middleware is active:

```go
package main

import "github.com/kataras/iris/v14"
```

```
func main() {
    app := iris.New(iris.Options{
        Context: iris.ContextOptions{
            BodyMaxMemory: 10 * iris.MB,
        },
    })
    // ...
}
```

A partial `iris.Options` literal merges with the defaults, so this configures only the body limit. Reading past the limit fails with `ErrRequestEntityTooLarge`, which the `rest/errors` envelope renders as 413 `TOO_LARGE` with the message “request body too large”. The default of `0` means unlimited. For per-route limits, use the `bodylimit` middleware:

```go
import "github.com/kataras/iris/v14/middleware/bodylimit"

upload := app.Router("/upload")
upload.Use(bodylimit.New(2 * iris.MB))
```

If you accept compressed request bodies, bound the decompressed size too. `compress.Handler` expands a body with no ceiling of its own, and a compression ratio of a thousand to one is ordinary for repetitive data, so a one-megabyte upload can become a gigabyte in memory before any handler sees it. `compress.Safe` is the same middleware with a 32 MB cap (`compress.DefaultMaxDecompressedSize`), and `compress.Limited` takes your own:

```go
import "github.com/kataras/iris/v14/middleware/compress"

app.Use(compress.Safe)         // 32 MB decompressed
app.Use(compress.Limited(8 * iris.MB))
```

Reading past the cap fails with `compress.ErrDecompressedBodyTooLarge`. A `BodyMaxMemory` or a `bodylimit` registered after the compression middleware bounds the same stream, because by then the body being read is already the decompressed one. With `bodylimit`, `ReadJSON` and `ctx.BindJSON` stop at the limit like `Body` does; a form that decompresses past it is not bound, and `ReadForm` reports no error. `compress.Handler` logs a warning at first use when nothing bounds it.

Rate limiting

A rate limiter puts a ceiling on how fast any one client can call you, so a single misbehaving or malicious client cannot monopolize the server. The standard model is the **token bucket**: each client has a bucket holding up to `burst` tokens, every request spends one, and tokens drip back at a fixed rate: when the bucket is empty, requests are rejected until it refills.

The `rate` middleware implements a token bucket per client (keyed by remote IP by default). `rate.Limit(limit, burst, rate.Options{})` takes requests per second, a burst size and a settings value whose zero form keeps every default; `rate.Every` converts an interval into the rate:

```go
import (
    "time"

    "github.com/kataras/iris/v14/middleware/rate"
)

// 10 requests per second with a burst of 10.
app.Use(rate.Limit(rate.Every(time.Second), 10, rate.Options{}))
```

In builder-built applications the exceed handler is pre-wired to the `rest/errors` JSON renderer: clients over the limit receive a structured 429 response under the `RESOURCE_EXHAUSTED` code with a "try again in" hint. Override it per limiter with the `Options.ExceedHandler` field.

Give expensive or abusable endpoints their own, stricter budget:

```go
app.Post("/signin",
    rate.Limit(rate.Every(10*time.Second), 3, rate.Options{}),
    signinHandler,
)
```

Two useful settings: `Options.PurgeEvery` and `Options.PurgeMaxLifetime` clean up idle client entries (each falls back to its own default when the other is left zero), and `rate.SetIdentifier`, called from a middleware registered before the limiter, switches the client key from the IP to, for example, an API key.

Request IDs

Correlation IDs end up in logs and support tickets, so do not let clients choose them. `requestid.ServerOnlyGenerator` ignores any client-supplied `X-Request-Id` header and always generates a fresh ID on the server:

```go
import "github.com/kataras/iris/v14/middleware/requestid"

app.Use(requestid.New(requestid.ServerOnlyGenerator))
```

IP allowlists

For endpoints that should only be reachable from known networks, the `ipaccess` middleware matches exact IPs, CIDR ranges and regular expressions. Blocked patterns are checked first; anything that matches no allowed pattern receives 403. An empty allowed list means "no allowlist", so `ipaccess.New(nil, blocked)` expresses a blocklist-only policy:

```go
import "github.com/kataras/iris/v14/middleware/ipaccess"

admin := app.Router("/admin")
admin.Use(ipaccess.Must(
    []string{"10.0.0.0/8", "203.0.113.7"},
    nil,
))
```

A pattern that is neither an IP nor a CIDR is compiled as a regular expression, anchored at the start. The anchoring matters: write `10.0.0.*` expecting a glob and you get a regexp where `.` means any character, and with no leading anchor it would also match `110.0.0.99`. The end stays open so a prefix pattern such as `^198\.51\.100\.` still covers a whole /24; add `$` when you want an exact match. Prefer a CIDR when you mean a range.

The check runs against `ctx.RemoteAddr()`. With no `RemoteAddrHeaders` that is the connection address, and behind a reverse proxy the connection is the proxy's, so either enforce IP rules at the proxy or configure `RemoteAddrHeaders`. The header is then read from the right, where the proxy appends the address it got the request from, and the first public address is the one matched. An entry the client sent sits to the left of it and is not read, so a proxy that appends (nginx's `$proxy_add_x_forwarded_for`) is enough against a forged `X-Forwarded-For`. The exception is a client inside your own network: it is appended as a private address, which is skipped, and the entry on its left is the client's own writing. A proxy that overwrites the header closes that.

Reading from the right decides what a pattern can match. A private address in the header is never the client, so `10.0.0.0/8` above does not match a client that an internal proxy reports as `10.0.0.5`: the address of the connection, the proxy, is matched. Put the proxy's address in the patterns, or change what is read: `RemoteAddrHeadersForce` takes the first address of a header that has no public one, which the client can write, and an empty `RemoteAddrPrivateSubnets` takes the right-most one, which the nearest proxy appended. A proxy with a public address in front of your own, a CDN for example, is the address that is matched until its range is added to `RemoteAddrPrivateSubnets`. Name only headers your proxy writes: a header is read when the ones named before it give no public address. And a client that reaches the server with no proxy in front still writes the whole header, so such a server has to be reachable through the proxy only.

Input Validation

Validate at the edge, in the request type itself. `ctx.BindJSON[T]()` decodes the body and, when the type implements `iris.Validatable` (`Validate() error`), runs validation automatically; the handler never sees an invalid value and never calls `Validate` manually:

```go
import (
    "github.com/kataras/iris/v14"
    "github.com/kataras/iris/v14/rest/errors/validation"
)

type CreateUserRequest struct {
    Name  string `json:"name"`
    Email string `json:"email"`
    Age   int    `json:"age"`
}

func (r CreateUserRequest) Validate() error {
    return validation.Join(
        validation.String("name", r.Name).NotEmpty().Length(3, 50),
        validation.String("email", r.Email).Email(),
        validation.Number("age", r.Age).InRange(13, 120),
    )
}

func createUser(ctx iris.Context) {
    input, ok := ctx.BindJSON[CreateUserRequest]()
    if !ok {
        return // a detailed 4xx response is already written.
    }
    // input is well-formed and validated.
}
```

On failure the client receives a structured, field-by-field validation error; see Chapter 9 for the response shape.

Typed path parameters

Route macros are a validation layer too. A parameter typed as `uuid`, `int` or any other macro rejects malformed values before your handler runs:

```go
app.Get("/users/{id:uuid}", getUser)
app.Get("/posts/{page:int min(1)}", listPosts)
```

In builder-built applications a malformed typed parameter renders a 400 response under the `INVALID_ARGUMENT` code with an “invalid path parameter” message. A handler that declares `{id:uuid}` can therefore pass `id` to the database layer without re-checking its format.

SQL injection

Never build SQL by string concatenation. Use parameterized queries, so the driver keeps data separate from the statement:

```go
row := db.QueryRow(
    "SELECT id, name FROM users WHERE id = $1", id,
)
```

Typed macros upstream make this safer still: an `{id:uuid}` parameter cannot carry `' OR '1'='1` into your query in the first place. See Chapter 13 for database patterns.

Keeping Internals Private

Error responses must never leak connection strings, hostnames, SQL, file paths or stack traces. The `rest/errors` pipeline (Chapter 9) is built around that rule: unmapped internal failures render as a generic 500 while the full cause is logged server-side.

At the response site, log-and-hide explicitly:

```go
import "github.com/kataras/iris/v14/rest/errors"

func getReport(ctx iris.Context) {
    report, err := service.BuildReport(ctx)
    if err != nil {
        // Logs the full error server-side;
        // the client sees a generic message.
        errors.Internal.LogErr(ctx, err)
        return
    }
    ctx.JSON(report)
}
```

`errors.Internal.ErrPrivate(ctx, err)` does the same without choosing the log level for you. When the sensitive detail is created far from the response (in a repository or service layer), wrap it at the source with `errors.Private`:

```go
return errors.Private(
    fmt.Errorf("query failed on host %s: %w", host, err),
)
```

A `Private`-wrapped error keeps its message for server-side logs but is never shown to the client, no matter where it later surfaces in the errors pipeline. Combined with one central `errors.NewOptions()` error map (which maps `sql.ErrNoRows` to 404 out of the box), the failure modes of your application are enumerable and none of them exposes internals.

Sessions and Cookies

Session cookies are `Secure` and `SameSite=Lax` from the start; your job is to pick strong keys, a sensible lifetime, and to rotate the session ID at sign-in.

```go
import (
    "net/http"
    "time"

    "github.com/kataras/iris/v14"
    "github.com/kataras/iris/v14/middleware/sessions"
)

type UserSession struct {
    UserID string
    Role   string
}

// hashKey: 64 bytes, blockKey: 32 bytes; load both from the
// environment, never from source code or a committed file.
func newSessionHandler(hashKey, blockKey []byte) iris.Handler {
    opts := sessions.NewOptions(hashKey, blockKey)
    opts.MaxAge = 2 * time.Hour // default is 24 hours.
    // Secure=true and SameSite=Lax are already set. Escalate to
    // Strict when the application is sensitive enough to accept
    // losing cross-site top-level navigation with the cookie:
    opts.SameSite = http.SameSiteStrictMode

    db := sessions.NewCookieDatabase[UserSession](opts)
    return sessions.Handler(db, "session")
}
```

For local development over `http://localhost`, set `opts.Secure = false`, and only there.

`MaxAge` counts from the last change to the session, not from the last request. `sessions.Handler` saves a session and sends its cookie again only when a request changed it, so the two hours above start at sign-in and reading pages does not extend them. Where a visit should extend the session, call `session.Commit(ctx)` in the handler that only reads it (Chapter 10). A deployment that changes `Secure`, `Path` or `MaxAge` in these options waits for the same thing: it reaches a visitor at their next change of the session, not at their next request. That holds for the three databases of the package. The session of a custom `Database` is saved after every request.

A page that only reads the session therefore goes out with no `Set-Cookie`, and that header was what kept it out of shared caches. Behind a reverse proxy that caches, one visitor's page can be served to another, so send `Cache-Control: private` on every page that depends on the session. `httpcache` caches a request with cookies only when its key names every one of them, and it is safe as long as its handler is registered before `sessions.Handler`. With the session cookie in the

key it stores a page per session, and after a sign-out a request that still sends the old cookie gets the stored page until it expires: invalidate on sign-out, or mark the page private (Chapter 14).

Changing the keys signs every visitor out, and that is all it does. A cookie signed with the old keys, like one older than `MaxAge`, is treated as no cookie: the request gets a new session and the response a new cookie.

Rotate the session ID at sign-in

Session fixation works by making a victim authenticate under an identifier the attacker already knows. Defeat it by rotating the identifier whenever the privilege level changes, most importantly right after sign-in:

```go
func signin(ctx iris.Context) {
    // ...verify credentials first...

    session := sessions.Get[UserSession](ctx)
    session.Set(UserSession{UserID: user.ID, Role: user.Role})

    if err := session.RegenerateID(ctx); errors.HandleError(ctx, err) {
        return
    }

    ctx.OK(iris.Map{"message": "signed in"}, nil)
}
```

`RegenerateID` deletes the old server-side entry and issues a new ID. It is a no-op for cookie-based databases, whose data lives inside the signed cookie and has no server-side identifier to rotate. Keep the order of the listing: `RegenerateID` stores the session and `sessions.Handler` commits nothing after it, so data set after the call is never stored on Redis or Postgres. On sign-out, call `session.Delete(ctx)` to expire the session entirely.

Cookies you set yourself

`ctx.AddCookie` writes cookies with `HttpOnly` and `SameSite=Lax` set. Add the `Secure` attribute and adjust `SameSite` per call:

```go
ctx.AddCookie("prefs", "compact",
    iris.CookieSecureForce,
    iris.CookieSameSite(http.SameSiteStrictMode),
)
```

`iris.CookieSecure` sets the attribute only when the current connection is TLS; `iris.CookieSecureForce` sets it unconditionally, which is what you want behind a TLS-terminating proxy. `ctx.AddCookieOptions` applies a set of options to every cookie set later in the same

request.

Setting a cookie twice in one response does not send it twice. `ctx.AddCookie`, `ctx.SetCookie` and `ctx.RemoveCookie` replace a `Set-Cookie` line that already carries the same name, path and domain, the way `SetHeader` replaces a header, so a middleware that refreshes a cookie a handler already set leaves one line for the browser to read. A cookie with the same name on another path or domain is still its own line.

Authentication Hardening

JWT

Keep access tokens short-lived and verify them from the `Authorization` header, which is the verifier's only extraction source by default:

```go
import (
    "os"
    "time"

    "github.com/kataras/iris/v14"
    "github.com/kataras/iris/v14/middleware/jwt"
)

type UserClaims struct {
    jwt.Claims
    Username string `json:"username"`
    Role     string `json:"role"`
}

func protectAPI(app *iris.Application, key []byte) {
    verifier := jwt.NewVerifier(jwt.HS256, key)
    verifier.WithDefaultBlocklist() // enable token revocation.

    api := app.Router("/api")
    api.Use(verifier.Verify(func() any { return new(UserClaims) }))
}
```

- Sign with a short max age (`jwt.NewSigner(jwt.HS256, key, 15*time.Minute)`) and issue a refresh token alongside via the signer's `NewTokenPair`. A stolen access token then expires in minutes. Chapter 10 shows the full flow.
- The blocklist makes sign-out meaningful: revoked tokens fail verification even before they expire.
- The signing key comes from the environment (`os.Getenv`), never from a source file. See Secrets Hygiene.

Basic authentication

Basic auth transmits credentials base64-encoded (encoded, not encrypted), so it is only acceptable over TLS. Enforce that in the middleware itself and store bcrypt hashes rather than plaintext passwords:

```go
import (
    "time"

    "github.com/kataras/iris/v14/middleware/basicauth"
)

auth := basicauth.New(basicauth.Options[basicauth.Map]{
    Realm:     "Restricted",
    Allow:     basicauth.AllowUsersFile[basicauth.Map]("users.yml", basicauth.BCRYPT),
    HTTPSOnly: true, // reject any non-HTTPS request outright.
    MaxAge:    2 * time.Hour,
    MaxTries:  3, // 403 after three failures.
})

admin := app.Router("/admin")
admin.Use(auth)
```

Password hashing cost

Passwords are never stored, only their hashes: one-way fingerprints that let you verify a password without keeping it. General-purpose hashes such as SHA-256 are unsuitable because they are fast: an attacker holding a stolen database can test billions of guesses per second. **bcrypt** is designed to be slow, and its cost factor lets you make it slower still as hardware improves.

The `auth/identity` password hasher uses bcrypt. Its work factor is the package-level `password.Cost`, defaulting to `bcrypt.DefaultCost`. Raising it makes offline brute-force proportionally more expensive at the price of more CPU per hash; hashes created with a different cost remain verifiable because the cost is embedded in the hash itself:

```go
import "github.com/kataras/iris/v14/auth/identity/password"

func init() {
    password.Cost = 12 // default is bcrypt.DefaultCost (10).
}
```

Finally, recall from Secure by Default that the auth SDK confines sign-in redirects to same-site paths and that `auth/oauth2` issues high-entropy opaque codes; neither needs configuration, so do not re-implement them less safely.

CORS

An **origin** is the scheme-host-port triple of a URL: `https://app.example.com` and `https://api.example.com` are different origins. Browsers enforce the same-origin policy: a script running on one origin cannot read responses from another. **CORS** (Cross-Origin Resource Sharing) is the server's controlled opt-out, response headers that tell the browser which foreign origins may read your API's responses. For requests with side effects, the browser first sends a **preflight**, an `OPTIONS` request asking permission before the real one goes out.

Cross-origin policy is an allowlist. Name your exact origins; a wildcard is only for genuinely public, credential-free APIs:

```go
import (
    "time"

    "github.com/kataras/iris/v14/middleware/cors"
)

app.UseRouter(cors.New().
    ExtractOriginFunc(cors.StrictOriginExtractor).
    AllowOrigins(
        "https://app.example.com",
        "https://admin.example.com",
    ).
    AllowHeaders("Content-Type", "Authorization").
    MaxAge(10 * time.Minute).
    Handler())
```

`StrictOriginExtractor` refuses to treat requests without a well-formed `Origin` header as same-origin-safe.

Name your origins. `cors.New()` on its own allows every origin, echoes the requesting one back, and advertises `Access-Control-Allow-Credentials: true` alongside it. That combination is what lets a page on any domain run

```js
fetch("https://your-api/me", {credentials: "include"})
```

and read the response as the logged-in user. The wildcard `*` is handled (browsers reject `*` with credentials, and the middleware suppresses the header there unless you call `AllowCredentials()`), but a reflected specific origin is not, and a reflected origin is what the default produces.

`cors.Strict()` is `cors.New()` with credentials off until you ask for them:

```go
app.UseRouter(cors.Strict().
    AllowOrigin("https://app.example.com").
    AllowCredentials().
    Handler())
```

Wildcards in an origin pattern match one label, so `https://www.example.*` accepts `https://www.example.com` and `https://www.example.org` but not `https://www.example.evil.com`. A pattern written without the dot, `https://www.example*`, is a plain prefix match and will accept `https://www.exampleshop.com`; use it only when that is what you mean.

If you use the builder, the `AllowOrigin` step accepts the same comma-separated origin line and wires the handler at the right position in the chain; pass your production origins there instead of `"*"`.

CSRF Protection

Iris does not bundle a CSRF-token middleware, and for most v14 applications you do not need one. Defense is layered:

SameSite cookies are the primary defense. Session cookies default to `SameSite=Lax`, which stops the browser from attaching them to cross-site subrequests and cross-site form POSTs, the classic CSRF vector. Two consequences follow. First, never mutate state on GET: Lax still sends the cookie on top-level GET navigations, so a state-changing GET would remain forgeable. Second, escalate to `SameSiteStrictMode` when the application warrants it (see Sessions and Cookies).

Header-carried tokens are immune. A JWT sent in the `Authorization` header is attached by your JavaScript, not by the browser, so a cross-site attacker cannot cause it to be sent. Token-authenticated APIs get CSRF resistance for free.

Watch what method override widens. `methodoverride.New` accepts `_method` from the query string and the form body by default, so a cross-site form POST can reach a route you registered as `DELETE`. Only `POST` is overridable, so a GET can never become a DELETE, but if you rely on "a browser cannot make a cross-site DELETE" as part of your reasoning, restrict the override to headers, which a cross-site form cannot set:

```go
import "github.com/kataras/iris/v14/middleware/methodoverride"

app.UseRouter(methodoverride.New(methodoverride.Options{HeadersOnly: true}))
```

Require a custom header on cookie-authenticated APIs. Cross-site HTML forms cannot set custom headers, and a cross-origin script that tries triggers a CORS preflight that your strict CORS policy rejects. A small middleware turns that into a hard guarantee:

```go
func requireCustomHeader(ctx iris.Context) {
    switch ctx.Method() {
    case "GET", "HEAD", "OPTIONS":
        ctx.Next()
        return
    }

    if ctx.Header("X-Requested-With") == "" {
        errors.PermissionDenied.Message(ctx, "custom request header required")
        return
    }

    ctx.Next()
}

api := app.Router("/api")
api.Use(requireCustomHeader)
```

Your clients then send `X-Requested-With: XMLHttpRequest` (or any custom header you choose) on every state-changing request.

Server-rendered forms. If you serve classic HTML forms and want synchronizer tokens on top of SameSite, implement them with the sessions API from this chapter: generate a random token with `crypto/rand`, store it in the typed session data, embed it in a hidden form field, and compare the posted value against the session on submission. It is a dozen lines of your own code, with no framework package required.

Static File Safety

The file server serves what is in the directory, including dotfiles such as `.git`, `.env` and `.htpasswd`, which have no business being downloadable. Block them with `fileserver.BlockHidden` as the `AssetValidator`:

```go
import "github.com/kataras/iris/v14/controller/fileserver"

app.Router("/static", fileserver.New("./assets", fileserver.Options{
    IndexName:      "index.html",
    AssetValidator: fileserver.BlockHidden,
}))
```

`BlockHidden` denies any request whose path contains a dot-prefixed segment. Note that this includes conventional directories like `.well-known`; if you need one of those served, compose your own validator on top of it. `AssetValidator` is a general per-request hook (return `false` to deny), so it also serves for authorization checks on downloads. Better still, keep secrets and VCS metadata out of the served directory entirely: serve a dedicated `assets` directory, not the project root.

As a backstop, the file server logs a warning at startup when the directory you point it at holds one of the well-known sensitive dotfiles and no `AssetValidator` is set. It is a reminder, not a defence: the files are still served.

Symbolic links are the other way out of the directory. A directory given as a string is served through `http.Dir`, which checks the request path as text but follows a link inside the directory that points somewhere else, so a link to `/etc` or to your home directory is served like any other folder. To serve only what is under the directory, open it as an `os.Root` and pass its file system:

```go
root, err := os.OpenRoot("./assets")
if err != nil {
    log.Fatal(err)
}

app.Router("/static", fileserver.New(root.FS(), fileserver.Options{
    IndexName:      "index.html",
    AssetValidator: fileserver.BlockHidden,
}))
```

A link that leaves the root now answers 404. That costs two things. The root holds the directory open for the life of the process, so a deploy that replaces the directory keeps serving the old one until the process restarts. And a link that leaves the directory is no longer served at all, which breaks a layout that relies on one on purpose, such as `./assets/shared` linked to a directory outside it.

Security Headers

Security headers are instructions to the browser about how your responses may be used. The two most consequential: **Content-Security-Policy (CSP)** declares which sources of scripts, styles and images a page may load, so that even successfully injected markup cannot execute an attacker's script, a strong second line of defense against XSS. **Strict-Transport-Security (HSTS)** tells the browser to reach your domain only over HTTPS from then on, closing the window in which a first plaintext request could be intercepted.

Iris does not inject opinionated security headers; set the ones that match your application in a small router-level middleware so they apply to every response, including 404s:

```go
func securityHeaders(ctx iris.Context) {
    ctx.SetHeader("X-Content-Type-Options", "nosniff")
    ctx.SetHeader("X-Frame-Options", "DENY")
    ctx.SetHeader("Referrer-Policy", "strict-origin-when-cross-origin")
    ctx.SetHeader("Content-Security-Policy", "default-src 'self'")

    if ctx.IsSSL() {
        ctx.SetHeader("Strict-Transport-Security",
            "max-age=31536000; includeSubDomains")
    }

    ctx.Next()
}

app.UseRouter(securityHeaders)
```

Tailor the Content-Security-Policy to what the application actually loads: an API that serves only JSON can keep the strict `default-src 'self'`, while a server-rendered site needs entries for its scripts, styles and images. Send Strict-Transport-Security only over TLS; a plaintext response carrying HSTS is ignored by browsers anyway.

Secrets Hygiene

Signing keys, session keys, database passwords and admin credentials live in the environment (or the secret manager that feeds it), never in source code and never in committed configuration files. Committed YAML holds addresses, timeouts and feature flags; the environment holds everything you would not paste into a code review.

Read secrets at startup and fail fast when one is missing:

```go
import (
    "fmt"
    "os"
)

func loadSecrets(cfg *Config) error {
    cfg.AdminPassword = os.Getenv("ADMIN_PASSWORD")
    if cfg.AdminPassword == "" {
        return fmt.Errorf(
            "ADMIN_PASSWORD environment variable is required")
    }
    return nil
}
```

A missing secret at boot is a deployment error with an obvious message; a missing secret discovered at request time is an outage. Two adjacent habits: treat access logs as sensitive too (they can contain tokens and personal data; rotate them and keep them out of version control), and prefer environment-first configuration end to end, as described in Chapter 11.

Production Checklist

Every line below is one or two lines of code away from an application built with the patterns in this book:

Concern	Action
Request bodies	`BodyMaxMemory` in `iris.Options`, or `bodylimit.New` per route
Compressed bodies	`compress.Safe` (or `compress.Limited`) instead of `compress.Handler`
Response caching	`cache.Strict(opts)` on routes that serve both anonymous and signed-in callers
Rate limiting	`rate.Limit(limit, burst, rate.Options{})`; builder apps render the JSON 429 automatically
CORS	`cors.Strict()` with exact origins; `cors.New()` reflects any origin with credentials
Method override	`methodoverride.Options{HeadersOnly: true}` if you rely on verb reachability
Cookies	defaults are `Secure` + `SameSite=Lax`; add `iris.CookieSecureForce` behind a TLS proxy
Session fixation	`session.RegenerateID(ctx)` right after sign-in
Dotfiles	`fileserver.BlockHidden` as the `AssetValidator`
Request IDs	`requestid.New(requestid.ServerOnlyGenerator)`
Password hashing	raise `password.Cost` above the bcrypt default
TLS	`app.Run(iris.ServerOptions{...})` with certificates or `Letsencrypt: true`
Access log	`LogRequests(true)`; set `AccessLog.Redact` or `RequestBody = false`; rotate and gitignore the log file
Health endpoint	`/health` reports env and developer; mount your own if that is too much exposure
Secrets	environment only; fail fast at startup when missing
Error responses	central `errors.NewOptions()` map; `errors.Private` for sensitive causes
Recovery	installed by the builder; do not add a second one
Dependencies	keep them current; run `govulncheck ./...` in CI

Summary

- Iris v14 ships secure defaults: `Secure` + `SameSite=Lax` session cookies, header-only JWT extraction, same-site-only sign-in redirects, `SameSite=Lax` on `ctx.AddCookie`, and high-entropy OAuth2 codes, each with a deliberate opt-out.
- A few defaults favour compatibility instead, and each has an opt-in: `cors.New()` reflects any origin with credentials (`cors.Strict()`), request bodies and decompression are unbounded (`BodyMaxMemory`, `compress.Safe`), `cache.New` applies a `public` policy to credentialed requests too (`cache.Strict`), and the access log records request bodies (`AccessLog.Redact`).
- Bound every request: `BodyMaxMemory` (or `bodylimit`) caps bodies at 413, `rate.Limit` throttles clients with a pre-wired JSON 429 in builder apps, `requestid.ServerOnlyGenerator` stops ID forgery, and `ipaccess` restricts network reach.
- Validate at the edge: `ctx.BindJSON[T]()` runs `Validate() error` automatically, and typed route macros reject malformed parameters with a 400 before your handler runs.
- Keep internals private: the errors pipeline renders generic 500s while logging full causes; `errors.Private` guarantees a sensitive error never reaches a client.
- Harden authentication: `RegenerateID` at sign-in, short-lived JWTs with a blocklist, basic auth over TLS only, a raised `password.Cost`.
- CSRF defense is SameSite cookies first, header-carried tokens second, and a custom-header requirement for cookie-authenticated APIs, with no dedicated middleware needed.
- Secrets live in the environment, are read at startup, and missing ones fail the boot, not the request.

Further Reading

- OWASP Top Ten: the canonical survey of the most common web application attack families and how they are exploited.
- OWASP CSRF Prevention Cheat Sheet: the defense layers behind this chapter's CSRF guidance, including SameSite cookies and custom request headers.
- OWASP Password Storage Cheat Sheet: current guidance on password hashing algorithms and work factors.
- MDN: Content Security Policy: how CSP directives are written and what each one restricts.

- MDN: Using HTTP cookies: the `Secure`, `HttpOnly` and `SameSite` attributes explained in depth.
- Go vulnerability management: how `govulncheck` finds known-vulnerable dependencies in your builds.

CHAPTER 19

Deployment & DevOps

An Iris application compiles to a single static binary with no runtime dependencies, which makes deployment refreshingly boring: build the binary, hand it its configuration through the environment, and put a health check behind whatever supervises it. This chapter covers the Iris-specific parts first (production server options, graceful shutdown, proxy headers, the builder's health endpoint) and keeps the generic DevOps material (Docker, Kubernetes, systemd, CI) short and honest.

Background

Deployment, stripped of tooling, is three problems: get your program onto a machine, keep it running when it crashes or the machine reboots, and replace it with a newer version without dropping requests. Every technology in this chapter (Docker, Kubernetes, systemd, a reverse proxy) automates part of one of those three.

Go makes the first problem unusually small. A **static binary** is an executable that carries everything it needs: no interpreter, no separate runtime, no shared libraries expected on the target machine. Copying one file is the entire installation.

A **container** packages a process together with its complete filesystem (binary, certificates, configuration defaults) into an **image**, an immutable snapshot that runs identically on your laptop, a CI runner and a production host. That immutability is the point: "works on my machine" stops being an argument, because the machine ships with the program. Docker is the tool most people use to build and run images, and a **Dockerfile** is the recipe that builds one.

A **reverse proxy** is a server (nginx, Caddy, a cloud load balancer) that sits in front of your application and forwards requests to it. It typically terminates TLS (holds the certificates and decrypts traffic) and can balance load across several application instances while shielding them

from the open internet. The cost is indirection: the application now sees the proxy's address instead of the client's, which is why forwarded headers exist, and why they must be explicitly trusted.

An **orchestrator** such as Kubernetes automates the "keep it running" half at fleet scale: it starts a declared number of container replicas, restarts the ones that fail health checks, spreads them across machines, and rolls out new image versions gradually. On a single host, **systemd** (the standard Linux service manager) does the same job with far less machinery: start the binary at boot, restart it on failure.

Two smaller terms round out the vocabulary. A **signal** (`SIGTERM`, `SIGINT`) is the operating system's way of asking a process to stop; graceful shutdown means catching that request and finishing in-flight work before exiting. **Continuous integration (CI)** is the habit of building and testing every change on a clean machine automatically, so a broken build is caught at push time rather than deploy time.

The good news running through this chapter: because an Iris application compiles to one static binary configured from the environment, every one of these targets (container, Kubernetes pod, systemd unit) reduces to "put the file there and hand it variables."

Building for Production

Disable cgo and the build produces a fully static Linux binary that runs in any container base image, including an empty one:

```sh
CGO_ENABLED=0 GOOS=linux go build -trimpath \
    -ldflags="-s -w" -o bin/server .
```

- `CGO_ENABLED=0` removes the libc dependency.
- `-trimpath` strips your local filesystem paths from the binary.
- `-ldflags="-s -w"` drops the symbol table and DWARF debug data, shrinking the binary considerably. Stack traces still work.

You do not need to inject a version with `-X` flags. Go stamps VCS metadata into every binary built inside a git checkout, and Iris reads it through `runtime/debug.ReadBuildInfo`: the commit hash and commit time appear as `iris.BuildRevision` and `iris.BuildTime`, are printed at startup (unless `DisableStartupLog` is set), and are served by the builder's `/health` endpoint described later. Build from a checkout that includes `.git` and the information travels with the binary for free.

Serving: Listen and Run

`app.Listen(addr)` is the short form for development and simple deployments. It is exactly equivalent to calling `app.Run` with sensible defaults:

```go
app.Listen(":8080")

// is the same as:
app.Run(iris.ServerOptions{
    Addr:           ":8080",
    Network:        "tcp",
    KeepAlive:      3 * time.Minute,
    SocketSharding: true,
})
```

For production you usually want `app.Run(iris.ServerOptions{...})` directly, because that is where timeouts, TLS and shutdown behavior live.

Both calls block until the server stops. After a graceful shutdown they return nil, and only once it has finished: the requests in flight have been answered, or the shutdown's deadline has passed, and the closers that `app.Shutdown` runs after the server have returned. A `main` that ends with `app.Listen` therefore cuts off neither a request nor a closer. A failed start, or a server that later dies with an error, is logged at fatal level and the process exits with status 1, so systemd or a container runtime sees the failure and restarts the unit. That is why the listings in this book call `app.Listen` and `app.Run` as statements. A process that runs more than one Iris server, or a test that opens a real listener, sets `iris.Options{DisableFatalRun: true}`; the failure is then logged at error level and returned to the caller.

Server timeouts

A server without timeouts is one slow client away from resource exhaustion. `iris.ServerOptions.Timeout` takes a `*iris.TimeoutOptions`, built fluently with `iris.Timeout()`:

```go
package main

import (
    "time"

    "github.com/kataras/iris/v14"
)

func main() {
    app := iris.New()

    app.Get("/", func(ctx iris.Context) {
```

```
        ctx.JSON(iris.Map{"status": "ok"})
    })

    app.Run(iris.ServerOptions{
        Addr: ":8080",
        Timeout: iris.Timeout().
            WithServerTimeout(60 * time.Second).
            WithShutdownOnInterruptTimeout(20 * time.Second),
    })
}
```

`WithServerTimeout(total)` treats the argument as the total lifecycle budget and derives the read-header, write and idle timeouts from it. The body-read timeout is deliberately not set globally, so handlers that accept large uploads keep control. Every field can also be set individually:

Method	Field it sets
`WithServerTimeout(d)`	`ServerTimeout`, derives header/write/idle
`WithReadTimeout(d)`	`ReadTimeout` (headers and body)
`WithReadHeaderTimeout(d)`	`ReadHeaderTimeout`
`WithWriteTimeout(d)`	`WriteTimeout`
`WithIdleTimeout(d)`	`IdleTimeout`
`WithHandlerTimeout(d, html...)`	Wraps the router in `http.TimeoutHandler`
`WithShutdownOnInterruptTimeout(d)`	Drain window on interrupt, and the deadline of the context `iris.RegisterOnInterrupt` functions receive (15s from `iris.Timeout()`; zero or negative sets no limit)

The 15 seconds belong to `iris.Timeout()` and to a server with no `Timeout` at all. A `&iris.TimeoutOptions{...}` literal that leaves `ShutdownOnInterruptTimeout` out carries zero, and so does a configuration file whose `Timeout` section omits the field. Zero sets no deadline: Ctrl+C then waits for the last request however long it takes, so set the field when the drain must be bounded. The same value bounds the cleanup you hook to the signal, which the Graceful Shutdown section below shows.

TLS

When Iris terminates TLS itself, `ServerOptions` offers two paths:

```
app.Run(iris.ServerOptions{
    Addr:        ":443",
    Hosts:       []string{"example.com"},
    Letsencrypt: true,
})
```

`Letsencrypt` is a plain boolean. The first `Hosts` entry is the root domain used for issuance; certificates are cached in a local `./letsencrypt` directory and renewed automatically 14 days before expiration. The server proves that it owns the domain with the ACME tls-alpn-01 challenge, answered on this same TLS listener. Let's Encrypt always connects to port 443 for it, so port 443 of every host has to reach this server. No port 80 handler takes part, and none is served. If `CertFile` / `KeyFile` are set alongside it, the server presents the issued certificate and falls back to that pair when issuance fails, which is convenient for local testing. A custom `TLSConfig *tls.Config` field exists for full control; it replaces both paths, and combining it with either panics at startup, on purpose.

In most production topologies, however, a reverse proxy or load balancer terminates TLS and Iris listens on plain HTTP behind it; see Running Behind a Reverse Proxy.

Tunnels: ngrok and SSH

For a demo, or to receive a webhook on a laptop, expose the local server through a tunnel instead of deploying it. `Ngrok` opens an ngrok tunnel and `SSHTunnel` a reverse SSH tunnel through serveo.net, localhost.run or a host of your own. Use one of them alone: the tunnel terminates TLS, so `CertFile`, `KeyFile`, `Letsencrypt` and `TLSConfig` stay unset.

ngrok is opt-in. Its code lives in the `core/server/ngrok` package, so a program that never opens a tunnel does not link the ngrok SDK. Add the blank import next to the options and the package registers itself:

```go
import _ "github.com/kataras/iris/v14/core/server/ngrok"

app.Run(iris.ServerOptions{
    Addr: ":8080",
    Ngrok: &iris.NgrokOptions{
        AuthToken: "your-ngrok-token", // or the NGROK_AUTHTOKEN variable
        Region:    "us",
    },
})
```

Without the import the server does not start. It fails with `server: ngrok is configured but its listener is not linked in`, followed by the import line to add, rather than quietly serving without a tunnel. The options themselves did not change, so a YAML file that sets `Ngrok` keeps working once the import is there.

`SSHTunnelOptions` lives in the `core/server` package and has no root-package alias. It runs `ssh -T -R <RemotePort>:<LocalHost>:<LocalPort> [user@]<Domain>` for you:

```go
import "github.com/kataras/iris/v14/core/server"

app.Run(iris.ServerOptions{
    Addr: ":8080",
    SSHTunnel: &server.SSHTunnelOptions{
        Domain:     "serveo.net",
        RemotePort: 80,
        LocalHost:  "localhost",
        LocalPort:  8080,
    },
})
```

Graceful Shutdown

Iris ships with graceful shutdown enabled. Unless you set `ServerOptions.DisableInterruptHandler`, the server installs a handler for `SIGINT` and `SIGTERM` that stops accepting new connections, drains in-flight requests for up to `ShutdownOnInterruptTimeout` (15 seconds by default), and then exits. `docker stop`, a Kubernetes pod termination and a plain Ctrl+C all take this path with zero code on your part.

The exit is the handler's own: it ends the process once every function on its list has returned, with status 0 unless one of them returned an error, and the server's drain is the entry that is always there. That is also the moment `app.Listen` returns. Whatever `main` does after `Listen` races that exit and can be cut at any statement, so it is no place for cleanup.

Cleanup work is a separate concern. `app.Shutdown(ctx)` runs every registered closer: the server's own shutdown is prepended automatically by `Run`, followed by everything you registered through the builder's `Deferrables(...)` step, `app.AddCloser(...)`, or `Close()` methods that `Services(...)` auto-detected (Chapter 3). To run those closers on interrupt as well, register the shutdown yourself with `iris.RegisterOnInterrupt`:

```go
package main

import (
    "context"
    "time"

    "github.com/kataras/iris/v14"
)

func main() {
    app := newApp() // your iris.NewBuilder(...).Build() chain.

    iris.RegisterOnInterrupt(func() {
```

```
        ctx, cancel := context.WithTimeout(
            context.Background(), 10*time.Second)
        defer cancel()
        app.Shutdown(ctx)
    })

    app.Listen(":8080")
}
```

Two details of that listing carry weight. The context given to `app.Shutdown` is the deadline of the drain, and every closer that takes a context receives it. This function is registered before the server's own interrupt entry, so it runs first and its ten seconds decide, not `ShutdownOnInterruptTimeout`. The shorter `iris.RegisterOnInterrupt(app.Shutdown)` uses the server's option instead: the signal handler passes `app.Shutdown` a context whose deadline is the `ShutdownOnInterruptTimeout` of the running server, 15 seconds unless you set another. A request still running at the deadline is cut, a stream that ends only when its client leaves included. Zero or a negative value there means no deadline, and one request that never ends then keeps the process alive. With more than one server in the process, the one that asked for the most wins: the largest value among them, and no deadline once one of them asked for none. A server started with `DisableInterruptHandler` hands its option to nobody, so the functions get no deadline then.

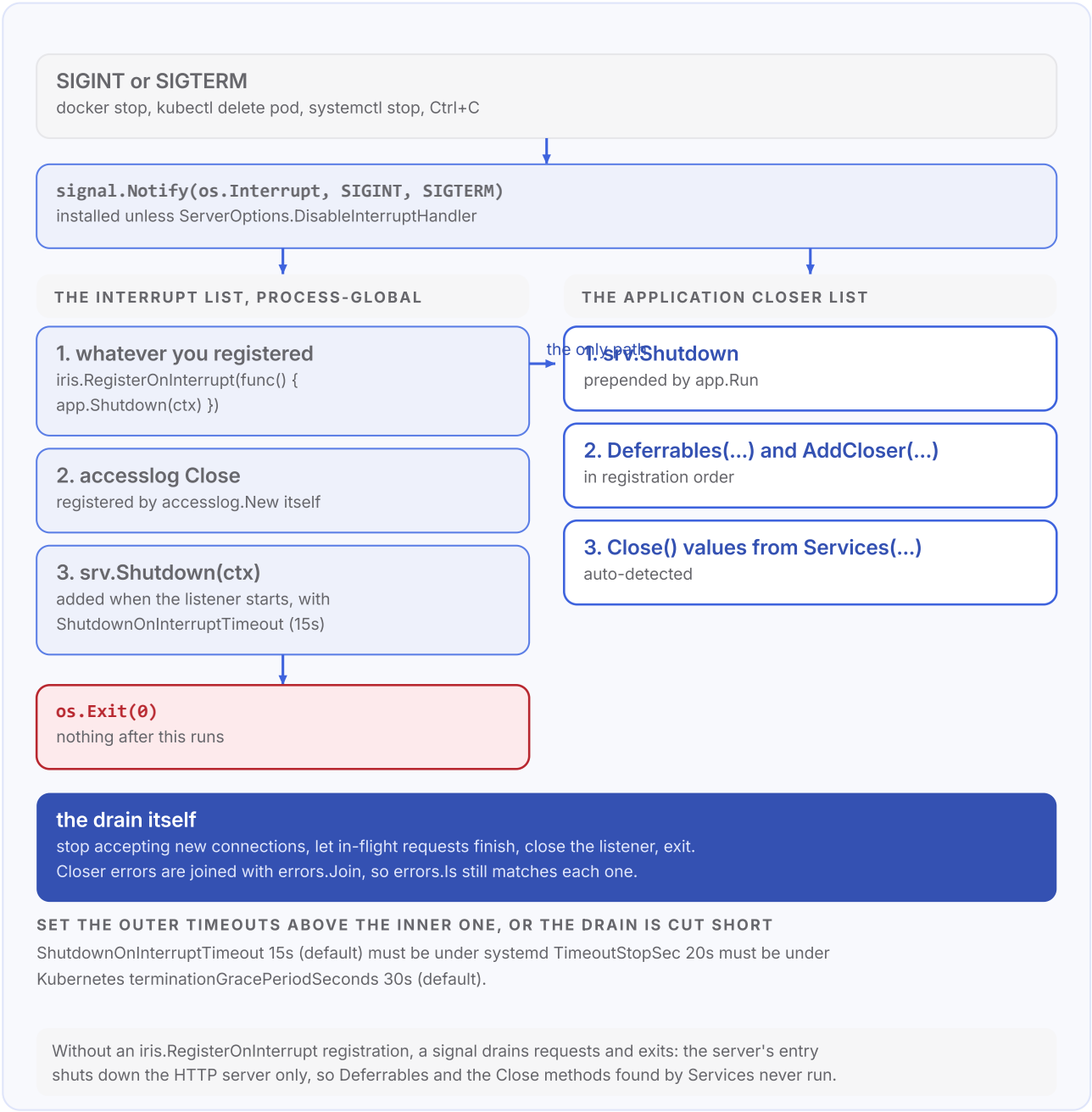
In both forms it is one deadline for the whole list, counted from the signal, and the closers after the server get what the drain left of it. When the drain uses all of it they receive a context that has expired already: the `Stop` of a worker is called with no time left, and the shutdown reports `Run did not return` for it. Pick a value that covers the slowest request you mean to wait for and the cleanup after it.

`app.Listen` as the last statement is the other one. `Listen` and `Run` return once `app.Shutdown` has run its whole list, the closers after the server included, so a `main` that ends with `Listen` cuts no closer short and needs nothing after it to wait on. That holds for a shutdown your own code starts as well: when another goroutine calls `app.Shutdown`, `Listen` returns after the last closer and `main` goes on with the statements that follow. Only a signal is different, because the process exits when the interrupt functions are done, and statements after `Listen` may not run then.

The wait must not run the other way. A closer or an interrupt function that waits for `Listen` or `Run` to return, on a channel `main` closes after `Listen` for example, blocks for good, because they are waiting for it. With nothing else running the Go runtime stops the program with `all goroutines are asleep - deadlock!`, and it hangs otherwise.

`RegisterOnInterrupt` accepts the same forms as `AddCloser`: `func()`, `func() error`, `io.Closer`, and context-aware variants. Errors from individual closers are aggregated with `errors.Join`, so `errors.Is` still matches each underlying error. An interrupt function that returns an error ends the process with status 1, after a `Received an error during shutdown` line. `app.Shutdown` registered directly hands back what its closers returned, so a closer that failed shows in the exit status. The listing above drops that error and exits with 0.

The reason that registration matters is worth seeing in full. There are two separate lists, and a signal drains only one of them before calling `os.Exit`. The server's own entry on that list shuts down the HTTP server and nothing else, so without a registration of your own, your `Deferrables` and your services' `Close` methods never run:



A signal drains the interrupt list and then calls os.Exit, and the server's own entry shuts down the HTTP server only, so your Deferrables and Close methods run only if something on that list calls app.Shutdown.

Containerizing with Docker

A **multi-stage build** keeps the shipped image small: the first stage uses the full Go toolchain image to compile the binary, and the final stage copies only that binary into a minimal base. The compilers, caches and source code never reach production.

A static Go binary wants a minimal image. The multi-stage build below compiles in the official Go image and ships on `distroless/static`, which adds CA root certificates and a non-root user on top of nothing at all, the two things `scratch` lacks. If you prefer `scratch`, copy `/etc/ssl/certs/ca-certificates.crt` from the build stage yourself.

DOCKERFILE

```
FROM golang:1.26 AS build
WORKDIR /src


COPY go.mod go.sum ./
RUN go mod download



COPY . .
RUN CGO_ENABLED=0 GOOS=linux go build -trimpath \
    -ldflags="-s -w" -o /out/bookshelf .

FROM gcr.io/distroless/static-debian12:nonroot
COPY --from=build /out/bookshelf /bookshelf
EXPOSE 8080
ENTRYPOINT ["/bookshelf"]
```

The exec-form `ENTRYPOINT` makes the binary PID 1, so it receives the `SIGTERM` that `docker stop` sends and drains gracefully as described above.

Configuration enters through the environment: the whole YAML document in one variable, following the pattern from the Bookshelf tutorial and Chapter 11:

SH

```
docker build -t bookshelf .
docker run --rm -p 8080:8080 \
    -e ADMIN_PASSWORD=change-me \
    -e BOOKSHELF_CONFIG='
LogLevel: info
Addr: ":8080"
Env: production
LogRequests: false
' bookshelf
```

One container-specific note: the builder's `LogRequests(true)` step writes `access_log.json` to the working directory in addition to stdout. In a container (especially one with a read-only root filesystem), pass a stdout-only access log through `LogRequestsWith` instead:

```go
import (
    "os"

    "github.com/kataras/iris/v14/middleware/accesslog"
)

ac := accesslog.New(os.Stdout)
ac.SetFormatter(&accesslog.JSON{})

// In the builder chain, in place of LogRequests(true):
//   .LogRequestsWith(ac).
```

Running Behind a Reverse Proxy

The most common production topology puts nginx, Caddy or a cloud load balancer in front of Iris: the proxy terminates TLS on 443 and forwards plain HTTP to the application on a local port. Two things must then be configured: the proxy must forward the client's identity in headers, and Iris must be told to trust those headers.

The proxy side, labeled clearly as **nginx configuration** (not Go):

```nginx
server {
    listen 443 ssl;
    server_name example.com;

    ssl_certificate     /etc/ssl/certs/example.com.pem;
    ssl_certificate_key /etc/ssl/private/example.com.key;

    location / {
        proxy_pass http://127.0.0.1:8080;
        proxy_set_header Host $host;
        proxy_set_header X-Real-Ip $remote_addr;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        proxy_set_header X-Forwarded-Proto $scheme;
    }
}
```

The same with Caddy, which obtains certificates automatically and sets the `X-Forwarded-*` headers by default (**Caddyfile**):

```
example.com {
    reverse_proxy 127.0.0.1:8080
}
```

On the Iris side, proxy trust lives in `iris.ContextOptions`. By default no `X-` headers are trusted. A client could otherwise spoof its IP or claim HTTPS. Opt in only when a proxy you control sets them:

```go
app := iris.New(iris.Defaults(func(opts *iris.Options) {
    // ctx.RemoteAddr() resolves the client IP from these headers.
    opts.Context.RemoteAddrHeaders = []string{
        "X-Forwarded-For", "X-Real-Ip",
    }
    // ctx.IsSSL() reports true when this header matches.
    opts.Context.SSLProxyHeaders = map[string]string{
        "X-Forwarded-Proto": "https",
    }
    // ctx.Host() may read the original host from this header.
    opts.Context.HostProxyHeaders = map[string]bool{
        "X-Forwarded-Host": true,
    }
}))
```

With this in place, `ctx.RemoteAddr()` returns the real client address instead of the proxy's, `ctx.IsSSL()` reflects the original scheme, and rate limiting, logging and audit trails stay meaningful. The same fields are settable from YAML configuration under the `Context:` key (Chapter 11).

`$proxy_add_x_forwarded_for` appends the address nginx received the request from to whatever `X-Forwarded-For` the client sent, and `ctx.RemoteAddr()` reads the header from the right for that reason. The last entry is the one nginx wrote. When it is a public address it is the result, whatever the client put in front of it. A private one is taken for another proxy of yours, and the reading moves one entry to the left.

That last entry has to be an address. When nginx itself accepts the request on a unix socket, `$remote_addr` is `unix:`, and an entry that is not an address ends the reading of its header with no result. Both headers of the example end that way then, and `ctx.RemoteAddr()` returns the address of the connection, which is the proxy's.

Health Checks and Kubernetes

The builder registers a health endpoint as one of its compiler-enforced steps (Chapter 3):

```go
iris.NewBuilder(cfg.Options).
    Prefix("/api").
    // ...
    Health(true, cfg.Env, cfg.Developer).
    // ...
```

`Health(true, env, developer)` registers `GET /health` at the application root, deliberately outside the `Prefix`, because load balancers and orchestrators probe fixed paths. The response is plain text with status 200: the server name and environment, the VCS build revision and time stamped into the binary, the developer name, cloud-container detection when applicable, and a final `OK` line. Probes only care about the status code; humans get the build information as a bonus. Applications built without the builder can register the same handler manually from `github.com/kataras/iris/v14/middleware/modrevision`.

Kubernetes distinguishes two probes. The **liveness probe** asks "is this process healthy enough to keep running?" Repeated failures make the orchestrator restart the container. The **readiness probe** asks "should traffic be routed here right now?" Failures merely take the pod out of the load balancer until it recovers. Pointing both at `/health` is a sensible default for a stateless API.

Wiring Kubernetes probes to it is a few lines. A minimal but complete deployment (**Kubernetes configuration**):

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: bookshelf
spec:
  replicas: 2
  selector:
    matchLabels:
      app: bookshelf
  template:
    metadata:
      labels:
        app: bookshelf
    spec:
      containers:
      - name: bookshelf
        image: registry.example.com/bookshelf:1.4.0
        ports:
        - containerPort: 8080
        env:
        - name: BOOKSHELF_CONFIG
          valueFrom:
            configMapKeyRef:
              name: bookshelf-config
              key: config.yml
        - name: ADMIN_PASSWORD
          valueFrom:
            secretKeyRef:
              name: bookshelf-secrets
              key: admin-password
        livenessProbe:
          httpGet:
            path: /health
            port: 8080
        readinessProbe:
          httpGet:
            path: /health
```

```
            port: 8080
          periodSeconds: 5
---
apiVersion: v1
kind: Service
metadata:
  name: bookshelf
spec:
  selector:
    app: bookshelf
  ports:
  - port: 80
    targetPort: 8080
```

On pod termination Kubernetes sends `SIGTERM`, which triggers the graceful drain from earlier. Keep the pod's `terminationGracePeriodSeconds` (30 by default) above your `ShutdownOnInterruptTimeout` so draining is never cut short.

Environment Configuration

The pattern that ties every deployment target together (Docker, systemd, Kubernetes) is a single YAML document delivered through a single environment variable, with secrets in their own variables. Iris supports it natively: `(*iris.Options).Bind` reads the `SERVER_CONFIG` variable first and falls back to a file, so one container variable can carry the entire framework configuration while a committed file serves development:

```go
var opts iris.Options
if err := opts.Bind("config.yml"); err != nil {
    log.Fatal(err)
}
app := iris.New(opts)
```

Applications with settings of their own embed `iris.Options` inline in a custom `Config` struct and read their own variable, exactly as the Bookshelf tutorial does with `BOOKSHELF_CONFIG` and `ADMIN_PASSWORD`. Partial documents are safe: options passed to `iris.New` merge with the defaults, so the YAML only needs to state what differs. Chapter 11 covers the full pattern, including multi-environment layouts.

Running Under systemd

On a plain Linux host, systemd is all the process supervision an Iris binary needs (**systemd unit**):

```ini
[Unit]
Description=Bookshelf API
After=network-online.target
Wants=network-online.target

[Service]
User=bookshelf
Group=bookshelf
WorkingDirectory=/opt/bookshelf
ExecStart=/opt/bookshelf/bookshelf
EnvironmentFile=/etc/bookshelf/env
Restart=on-failure
TimeoutStopSec=20

[Install]
WantedBy=multi-user.target
```

The `EnvironmentFile` holds `BOOKSHELF_CONFIG=...` and the secret variables, readable only by root and the service user. `systemctl stop` sends `SIGTERM`, so the graceful drain applies here too; `TimeoutStopSec` is set above the default 15-second drain window for the same reason as the Kubernetes grace period.

```sh
sudo systemctl daemon-reload
sudo systemctl enable --now bookshelf
```

Continuous Integration

CI for an Iris service is ordinary Go CI: vet, test with the race detector, and scan dependencies. The suite from Chapter 16 runs entirely in-process via `httptest`, so no services or containers are needed for the tests themselves (**GitHub Actions workflow**):

```yaml
name: CI
on:
  push:
    branches: [main]
  pull_request:

jobs:
  test:
    runs-on: ubuntu-latest
    steps:
    - uses: actions/checkout@v6
    - uses: actions/setup-go@v6
      with:
        go-version: 1.26.x
    - run: go vet ./...
    - run: go test -race ./...
```

```
    - run: go run golang.org/x/vuln/cmd/govulncheck@latest ./...
```

Add the `docker build`/push step of your registry of choice after the test job; the Dockerfile from this chapter needs no CI-specific changes.

Summary

- Build with `CGO_ENABLED=0 go build -trimpath -ldflags="-s -w"` for a static binary; Go's VCS stamping gives `/health` and the startup log their revision and build time with no `-X` flags.
- `app.Listen(addr)` is shorthand for `app.Run(iris.ServerOptions{...})`; production servers set `Timeout` via `iris.Timeout()` and choose between `CertFile`/`KeyFile`, `Letsencrypt: true`, or a TLS-terminating proxy in front. A failed start logs and exits with status 1; `iris.Options.DisableFatalRun` returns the error instead.
- Graceful shutdown is on by default: `SIGINT`/`SIGTERM` drains in-flight requests within `ShutdownOnInterruptTimeout`; `app.Shutdown(ctx)` runs closers and `Deferrables`, and `iris.RegisterOnInterrupt` hooks them into the interrupt path.
- Behind a proxy, set `Context.RemoteAddrHeaders`, `SSLProxyHeaders` and `HostProxyHeaders`; nothing is trusted by default.
- The builder's `Health(true, env, developer)` step serves `GET /health` at the root; point liveness and readiness probes at it.
- One YAML document in one environment variable configures everything; `Options.Bind` and the embedded-options pattern from Chapter 11 implement it.

Further Reading

- Docker: Get started: the official introduction to images, containers and Dockerfiles.
- Kubernetes: Overview: what an orchestrator is and which problems Kubernetes actually solves.
- Kubernetes: Liveness, Readiness and Startup Probes: configuring the probes this chapter points at `/health`.
- MDN: X-Forwarded-For: the forwarded-client-address header and its trust caveats.
- Let's Encrypt: How It Works. The automated certificate issuance behind `Letsencrypt: true`.

CHAPTER 20

Performance Optimization

Performance work starts with a measurement, not a hunch. Most web services are slowed down by a handful of causes: oversized responses, unbounded request bodies, chatty database access, accidental serialization of work. Every one of them shows up clearly in a profile or a benchmark before it shows up in prose. This chapter shows how to measure an Iris application, what the framework already optimizes so you do not re-optimize it, and where deliberate choices (compression, streaming, pagination, caching, concurrency) buy real wins.

Background

Two numbers describe a service's speed, and they are not the same number. **Latency** is how long one request takes from arrival to last byte: what an individual user feels. **Throughput** is how many requests the service completes per second: what capacity planning cares about. They often pull in opposite directions: batching work raises throughput but makes each request wait, and shedding load protects latency by lowering throughput. Decide which one you are optimizing before you change anything, because "faster" without a noun is not a goal.

Intuition about where a program spends its time is wrong more often than it is right. The code that looks slow and the code that is slow are usually different code. A **profiler** settles the argument: it samples what the running program is doing many times per second and produces a map of where CPU time and memory actually go, function by function and call path by call path. With that map you fix the few places that dominate the cost; without it you polish code that was never hot. That is why every section of this chapter begins, one way or another, with a measurement.

One Go-specific concept pays for itself here. When your handler creates values that outlive its stack frame, the runtime **allocates** them on the heap, and the **garbage collector (GC)** later spends CPU finding and reclaiming the ones no longer in use. The more a server allocates per request, the more often the GC runs and the more processor time it borrows from request

handling, which is why Go benchmarks report allocations per operation next to time per operation, and why the framework pools and reuses its per-request machinery instead of reallocating it.

The last background idea is the danger of optimizing the wrong thing. Speeding up a step that accounts for two percent of request time can never win more than two percent, no matter how brilliant the code, and every optimization costs complexity that you keep paying after the speed is forgotten. A change that does not move a measured number is not an optimization; it is a rewrite with extra risk. The chapter's ordering (measure, then learn what Iris already does, then reach for the levers) exists to keep your effort pointed at costs that are real.

Measure First

Three tools cover almost every performance question: pprof answers "where does CPU and memory go", `go test -bench` answers "did my change help", and the `httpcost` middleware answers "what does one request cost in production".

Profiling with pprof

Go ships its profiler in the standard library under the name pprof. A **CPU profile** samples the call stacks that are executing, many times per second over a window: functions that appear in many samples are where processor time goes. A **heap profile** records which call sites allocated the memory that is still live at the moment of the snapshot. The `go tool pprof` command reads both and renders tables and call graphs from them.

The `middleware/pprof` package mounts Go's standard profiling endpoints under a route group. Profiles expose internals of your process. Never leave them open to the public. Mount them behind authentication (or bind them to an internal-only listener):

```go
package main

import (
    "github.com/kataras/iris/v14"
    "github.com/kataras/iris/v14/middleware/basicauth"
    "github.com/kataras/iris/v14/middleware/pprof"
)

func main() {
    app := iris.New()

    auth := basicauth.Default(map[string]string{
        "ops": "a-strong-password",
    })
    app.Router("/debug/pprof", auth, pprof.New())
```

```
    app.Get("/", func(ctx iris.Context) {
        ctx.JSON(iris.Map{"message": "hello"})
    })

    app.Listen(":8080")
}
```

`pprof.New()` returns a routing controller; `app.Router` registers the `auth` handler as middleware for the group and mounts the controller under it. Visiting `/debug/pprof` lists the available profiles; the `go tool pprof` command consumes them directly (credentials go in the URL):

```sh
go tool pprof "http://ops:a-strong-password@localhost:8080/debug/pprof/profile?seconds=30"


go tool pprof "http://ops:a-strong-password@localhost:8080/debug/pprof/heap"
```

Inside the tool, `top` shows the heaviest functions and `web` renders a call graph. Profile under realistic load. An idle server profiles as nothing but scheduler noise.

Benchmarking handlers

For “did this change help”, write a Go benchmark against the built application. The application itself serves `http.Handler` after `Build()`, so the standard library’s recorder drives it without any network in the way, the same technique the framework’s own benchmark suites use:

```go
package main

import (
    "net/http"
    "net/http/httptest"
    "testing"

    "github.com/kataras/iris/v14"
)

func newBenchApp(b *testing.B) *iris.Application {
    app := iris.New()
    app.Get("/books/{id:int}", func(ctx iris.Context) {
        id := ctx.Params().GetInt("id")
        ctx.JSON(iris.Map{"id": id, "title": "A Book"})
    })

    if err := app.Build(); err != nil {
        b.Fatal(err)
    }

    return app
}

func BenchmarkGetBook(b *testing.B) {
    app := newBenchApp(b)
    req := httptest.NewRequest("GET", "/books/42", nil)
```

```
    b.ReportAllocs()
    b.ResetTimer()
    for i := 0; i < b.N; i++ {
        w := httptest.NewRecorder()
        app.ServeHTTP(w, req)
        if w.Code != http.StatusOK {
            b.Fatalf("unexpected status %d", w.Code)
        }
    }
}
```

```sh
go test -bench=BenchmarkGetBook -benchmem -count=6
```

`-benchmem` reports bytes and allocations per operation alongside time per operation. Reusing one request value is fine for GET; for POST benchmarks rebuild the body reader each iteration, because a body is consumed on first read.

Read benchmark output as deltas, not absolutes: numbers depend on the machine, its load, and the Go version, so the only meaningful comparison is before-versus-after on the same box. Run with `-count=6` or more and compare the two sets with benchstat, which reports whether a difference is statistically significant. A change that does not move ns/op, B/op, or allocs/op is not an optimization, whatever the code looks like.

Per-request cost with httpcost

Benchmarks run in a lab; the `middleware/httpcost` package measures requests in the running application. It samples time, CPU, and memory around the handler chain and folds them into a single configurable cost figure:

```go
package main

import (
    "github.com/kataras/iris/v14"
    "github.com/kataras/iris/v14/middleware/httpcost"
)

func main() {
    app := iris.New()

    // Outer middleware reads the metric after the chain completes.
    app.Use(func(ctx iris.Context) {
        ctx.Next() // run httpcost and the handlers first.

        if metric, ok := httpcost.Get(ctx); ok {
            ctx.Application().Logger().Infof(
                "%s %s: duration=%s cpu=%.4fs net=%dB cost=%.6f",
                ctx.Method(), ctx.Path(),
                metric.Duration, metric.CpuTime,
                metric.NetworkUsed, metric.Cost)
        }
```

```
    })
    app.Use(httpcost.New(httpcost.DefaultOptions))

    app.Get("/heavy", func(ctx iris.Context) {
        // ... some expensive work ...
        ctx.JSON(iris.Map{"done": true})
    })

    app.Listen(":8080")
}
```

The `Metric` carries `Duration`, `CpuTime` (seconds), `MemoryUsed` and `NetworkUsed` (bytes), and the weighted `Cost`. The weights come from `httpcost.Options` (`CostPerSecond`, `CostPerCPU`, `CostPerMB`), and any dimension can be disabled by setting it to zero. There is also `httpcost.Compute(job func()) float64` for measuring an arbitrary function outside a request.

Be honest about what the numbers mean: CPU and memory are process-wide deltas taken before and after the request, so under concurrent load a request is charged a share of its neighbors' work. The figures are excellent for spotting outliers and trends, not for exact accounting.

What Iris Already Does

Knowing what the framework optimizes tells you where *not* to spend effort.

Contexts are pooled. For every request, Iris takes a `*Context` and a response writer from `sync.Pool`-backed pools, serves the request, and returns both. The stores behind `ctx.Params()` and `ctx.Values()` are slices, not maps, and are reset (keeping their capacity) when the context is released, so steady-state traffic reuses the same storage instead of reallocating it. The corollary is a hard rule: never retain an `iris.Context` past the handler's return; it will be reused by another request (see Concurrency in Handlers for the safe pattern).

Routes are compiled once. `Build()` turns the registered routes into prefix trees, one per HTTP method and subdomain. Matching a request is a segment-by-segment walk of that tree: static segments are looked up, dynamic ones capture parameters, and the built-in typed parameters (`{id:int}`, `{id:uuid}`, ...) are validated by plain Go evaluator functions selected at registration time. There is no regular expression engine and no route-list scan at request time, so matching cost does not degrade as the route table grows. Requests read the trees with no lock, which is why a route registered after the build is not served. `Options.Router.EnableDynamicHandler` lets `app.BuildRouter()` compile the routes again while the server runs, and the price is a read lock on every request for as long as the option is on. Leave it off unless your routes really change at run time.

Accessors are typed, not reflective. `ctx.Params().GetInt("id")` and `ctx.Query().GetString("q")` read from those slice-backed entries and convert with `strconv`: no reflection, no intermediate maps. Missing values return the documented zero results (`""` for strings, `-1` for integers) rather than allocating error values on every probe.

JSON is streamed. With default options, `ctx.JSON` encodes straight into the response writer with a `json.Encoder`; the payload is never first assembled into an intermediate buffer by the framework.

The practical consequence: hand-rolled context pools, “fast” routing layers in front of Iris, and custom JSON encoder pools re-implement work the framework already does. Profile before adding any of them.

The rest of this chapter is the other half of that picture. Each lever below attaches to a stage the framework has already made cheap, which is why the measurement comes first and the lever second:

Measure first, always: pprof.New() behind auth for where CPU and heap actually go, go test -bench -benchmem -count=6 with benchstat for whether a change helped, and httpcost.New for the cost in production.

ALREADY OPTIMISED: REBUILDING ANY OF THIS IS WASTED EFFORT

Context and response writer from sync.Pool
Params() and Values() are slices, reset with their capacity kept

route match: one prefix tree per method and subdomain, built in Build()
typed parameters are evaluated by plain Go functions, with no regexp engine

typed accessors: ctx.Params().GetInt, ctx.Query().GetString
strconv, with no reflection and no intermediate maps

ctx.JSON: a json.Encoder straight into the response writer
with the default options the payload is never buffered first

YOUR LEVERS, EACH ATTACHED TO THE STAGE ABOVE IT

BodyMaxMemory, or bodylimit.New(n) per route
413 before a byte of the body is read

ctx.Paginated, and ctx.ReadJSONStream for large arrays
one page in the query and one page in the response; memory stays proportional to one element

keep the default iris.JSON options on hot paths
ASCII, Secure or Prefix switch to a buffered path. Indent does not.

compress.Handler on compressible groups only, and ctx.StreamWriter
wasted CPU on JPEG, PNG and ZIP; a flush per callback, so batch tens of kilobytes

NOT ON THE REQUEST PATH AT ALL

Caching is the request you never process at all (Chapter 14).
ctx.WithoutCancel() for work that must outlive the response, and never the pooled Context itself.
A bounded worker pool with select/default: a 503 now beats unbounded latency later.
GOGC and GOMEMLIMIT, one knob at a time, under load, with a number on the screen.

The framework already pools the Context, compiles the routes, avoids reflection in the typed accessors and streams JSON, so the levers worth pulling are the ones attached to your own code, and only after you have measured.

Response Efficiency

Bytes on the wire are usually the largest share of user-visible latency. Three levers control them.

Compression

The `middleware/compress` package negotiates the response encoding from the client's `Accept-Encoding` header (gzip, deflate, brotli, snappy, s2 and zstd are built in) and transparently decompresses request bodies that arrive with a `Content-Encoding`:

```go
import "github.com/kataras/iris/v14/middleware/compress"

app.Use(compress.Handler)
```

Applications assembled with `iris.NewBuilder()` enable the same middleware with the `Compression(true)` step (Chapter 3).

Compression is a trade: CPU per request in exchange for fewer bytes. It pays off on text (JSON, HTML, CSV) and is wasted on content that is already compressed (JPEG, PNG, ZIP, most video), where it burns CPU to save nothing. If your application serves both kinds, register the middleware on the route groups that produce compressible content instead of the root. And as always: measure. On a fast internal network, compressing small payloads can cost more latency than it saves; behind slow client links it is one of the cheapest wins available.

JSON encoding options

`ctx.JSON(v)` accepts an optional `iris.JSON` options value with `Indent`, `Prefix`, `EscapeHTML`, `ASCII`, `Secure` and `MarshalOptions` fields. The default (all zero) is also the fastest configuration: compact output from the standard library's `encoding/json/v2` encoder (Go 1.27), streamed directly to the writer, with HTML characters left unescaped because `application/json` responses gain nothing from the escaping. Indentation inflates the payload and adds encoding work, and setting `EscapeNonASCII`, `ArrayPrefix` or `Prefix` switches to a buffered encoding path. Keep the defaults on hot endpoints; pretty-printing is for humans reading a terminal, and clients that want it can pipe through `jq`. The `ctx.JSONOrError` helper (Chapter 6) uses the same encoder underneath and adds error handling, so it inherits the same fast path.

Streaming large responses

Building a multi-megabyte export in memory and writing it in one call holds the whole payload alive and delays the first byte until the last row is ready. Stream instead: write a chunk, flush, repeat. `ctx.StreamWriter` runs that loop for you: it calls your function, flushes what was written, and calls it again until you return a non-nil error (`io.EOF` is the conventional "done") or the client disconnects, in which case it returns the request context's cancellation error:

```go
import (
```

```
    "fmt"
    "io"

    "github.com/kataras/iris/v14"
)

func exportCSV(ctx iris.Context) {
    ctx.SetContentType("text/csv")

    next := rowFetcher() // yields one batch of rows per call.
    ctx.StreamWriter(func(w io.Writer) error {
        rows, ok := next()
        if !ok {
            return io.EOF // any non-nil error ends the stream.
        }

        for _, row := range rows {
            if _, err := fmt.Fprintln(w, row); err != nil {
                return err
            }
        }

        return nil // flush this chunk, call again.
    })
}
```

Each callback return is a flush, and flushing has its own cost. Write batches (hundreds of rows, tens of kilobytes) per call rather than one row at a time. For hand-rolled loops, `ctx.Write` plus `ctx.ResponseWriter().Flush()` gives the same behavior with manual control; check `ctx.IsCanceled()` between chunks so a disconnected client does not keep the loop running. Both patterns, including server-sent events, are covered in Chapter 6.

Request Efficiency

The mirror image of response streaming: do not buffer what you do not need, and do not accept what you cannot afford.

Cap request bodies

An unbounded body is a memory-exhaustion vector before it is a performance problem. `ContextOptions.BodyMaxMemory` caps the bytes any body-reading method (`Body`, `ReadJSON`, `ReadJSONStream`, `ReadMultipartRelated`, and everything built on them) will consume; past the limit they fail with `context.ErrRequestEntityTooLarge`, which the `rest/errors` map renders as `413 TOO_LARGE` with the message “request body too large”:

```
app := iris.New(iris.Options{
    Context: iris.ContextOptions{
        BodyMaxMemory: 4 * iris.MB,
```

GO

```
    },
})
```

For per-route limits (a generous cap on an upload endpoint, a tight one everywhere else), use the `bodylimit` middleware: `app.Use(bodylimit.New(2 * iris.MB))` (`github.com/kataras/iris/v14/middleware/bodylimit`). Details in Chapter 5.

Stream large arrays

When a client legitimately sends a large JSON array, decoding it into one slice holds every element in memory at once. `ctx.ReadJSONStream` decodes one object at a time, so memory stays proportional to a single element; process each item (or batch) as it arrives:

```go
batch := make([]Item, 0, 500)
err := ctx.ReadJSONStream(func(decode iris.DecodeFunc) error {
    var item Item
    if err := decode(&item); err != nil {
        return err
    }

    batch = append(batch, item)
    if len(batch) == cap(batch) {
        if err := svc.InsertBatch(ctx, batch); err != nil {
            return err
        }
        batch = batch[:0]
    }

    return nil
})
```

Paginate instead of loading everything

The most common self-inflicted performance wound is `SELECT *` into a slice and `ctx.JSON` of the whole table. `ctx.Paginated` reads `?page=&size=` (plus an optional JSON filter body), runs your collector with `LIMIT` / `OFFSET`-ready options, and responds with a page envelope:

```go
import (
    "context"

    "github.com/kataras/iris/v14"
)

func listBooks(ctx iris.Context) {
    ctx.Paginated(func(reqCtx context.Context,
        opts iris.PageOptions, filter BookFilter) ([]Book, int64, error) {
        return svc.Books(reqCtx, opts, filter)
    })
}
```

The database returns one page, the handler allocates one page, the client downloads one page. Input handling is detailed in Chapter 5, the response shape in Chapter 6.

Caching

The cheapest request is the one you never process. Iris supports the full ladder: `Cache-Control` / `ETag` headers and conditional `304 Not Modified` responses through the `cache` middleware, server-side response caching for expensive endpoints, and application-level caches in your service layer. Which rung to use, and the harder question of how to invalidate it, is the subject of Chapter 14; treat it as the second step of performance work, right after measuring, because no handler optimization competes with not running the handler.

Database Efficiency

In a typical CRUD service the database dominates the latency profile, and the fixes are database-side disciplines rather than Iris APIs.

Opening a database connection is expensive (a TCP handshake, authentication, session setup), so clients keep a **connection pool**: a set of already-open connections that requests borrow and return. That makes the pool's size a real tuning decision rather than an implementation detail.

Size the connection pool deliberately. `database/sql` pools connections for you, but its defaults are not a decision. Cap open connections at what your database can actually serve concurrently, keep a few idle so bursts do not pay connection-setup latency, and recycle connections so load-balancer and server-side timeouts do not surprise you:

```go
db.SetMaxOpenConns(50)
db.SetMaxIdleConns(10)
db.SetConnMaxLifetime(time.Hour)
db.SetConnMaxIdleTime(10 * time.Minute)
```

An oversized pool moves the queue from your application (visible, measurable) into the database (opaque, and it slows every other client too).

Kill N+1 queries. Fetching a page of 50 orders and then issuing 50 `SELECT`s for their customers turns one round trip into fifty-one. The fix is a join or a single `WHERE id IN (...)` batch. N+1 patterns hide well in code and show up loudly in per-request measurement: an endpoint whose duration scales with its item count is the classic signature.

Prepare hot statements. Preparing a statement once and executing it many times saves the database re-parsing and re-planning the query. Some drivers cache prepared statements transparently; check yours before building a caching layer around `db.Prepare` that the driver already provides.

Select what you serve. Query the columns the response needs, let the pagination options above bound the row count, and push filtering into `WHERE` clauses rather than Go loops.

Integration patterns (including `github.com/kataras/pg`, whose error mapping plugs into the Iris error registry) are covered in Chapter 13.

Concurrency in Handlers

A **goroutine** is Go's lightweight unit of concurrency: a function running independently, scheduled by the Go runtime onto operating system threads. Goroutines are cheap enough that a server happily runs thousands of them at once, which is exactly how Go's HTTP server works, one per request, with no configuration on your part.

Every request already runs on its own goroutine, so concurrency inside a handler is for one of two things: finishing the response sooner, or moving work out of the request path entirely.

Background work from a handler

The trap is lifetime, twice over. The `iris.Context` is pooled and reused the moment the handler returns, so a goroutine must never hold it; and the request's standard context is canceled when the response completes, so a goroutine holding *that* gets canceled mid-work. `ctx.WithoutCancel()` solves both: it returns a standard `context.Context`, detached from request cancellation, with the request's context values preserved, safe to keep:

```go
app.Post("/orders", func(ctx iris.Context) {
    order, ok := ctx.BindJSON[Order]()
    if !ok {
        return
    }

    if !ctx.OK(svc.Place(ctx, order), nil) {
        return
    }

    // Must-finish work continues after the response is sent.
    go audit.Record(ctx.WithoutCancel(), order)
})
```

Use `ctx.WithoutCancel()` for must-finish work and `ctx.StdContext()`, the request's own standard context, for work that *should* die with the request. An outbound HTTP call belongs in the second group: the transport keeps its context for cancel propagation after the handler returns, so it gets `ctx.StdContext()`, never the pooled `ctx` itself. Choosing wrongly in either direction produces leaked work or lost work.

Bounded worker pools

Back-pressure is the polite name for saying no early: when a system reaches capacity, it tells the caller to slow down or retry instead of accepting work it cannot finish on time.

Unbounded `go` statements under load become an unbounded number of concurrent database queries, image encodes, or outbound calls. A fixed-size worker pool (plain Go, no framework involvement) turns that into a queue with back-pressure:

```go
type Job struct {
    OrderID string
}

func startWorkers(n int, process func(Job)) chan<- Job {
    jobs := make(chan Job, 64)
    for i := 0; i < n; i++ {
        go func() {
            for job := range jobs {
                process(job)
            }
        }()
    }

    return jobs
}
```

```go
import "github.com/kataras/iris/v14/rest/errors"

jobs := startWorkers(4, processOrder)

app.Post("/orders/{id:uuid}/invoice", func(ctx iris.Context) {
    job := Job{OrderID: ctx.Params().GetString("id")}

    select {
    case jobs <- job:
        ctx.StatusCode(iris.StatusAccepted)
    default: // queue full: shed load instead of piling it up.
        errors.Unavailable.Message(ctx, "the invoice queue is full")
    }
})
```

The `default` branch is the point: when the queue is full, the handler says so immediately instead of letting latency grow without bound.

Nothing here closes `jobs`, so an in-flight invoice dies with the process. When that matters, make the pool an `iris.Worker`: `Run` owns the goroutines and returns when the shutdown context is done, `Stop` closes the channel, and `Services(pool)` ties both to the server's lifecycle (Chapter 3). Size the pool against the bottleneck resource (database connections, CPU cores, third-party rate limits), and measure, because the right number is an experiment, not a guess.

Runtime Tuning

Go's runtime needs little help. `GOMAXPROCS` defaults to the available CPU count and recent Go releases respect container CPU limits on Linux, so overriding it is rarely justified. Two environment knobs are worth knowing: `GOGC` scales how much the heap grows between garbage collections (higher means fewer, longer-spaced collections at the cost of more memory), and `GOMEMLIMIT` sets a soft memory ceiling that makes the collector work harder as the process approaches it, the practical way to keep a containerized service inside its memory allocation instead of meeting the OOM killer. Change one knob at a time, under realistic load, with the pprof and metrics tooling from the start of this chapter open. A GC setting you cannot connect to a measured improvement is a superstition, not a configuration.

Summary

- Measure before optimizing: `pprof.New()` mounted behind authentication for CPU and heap profiles, `go test -bench -benchmem` with the recorder-driven skeleton for before/after comparisons, and `httpcost.New` for per-request cost in production. Compare deltas on the same machine; distrust absolute numbers.
- Iris already pools contexts and response writers, compiles routes into per-method prefix trees at build time, reads parameters through typed non-reflective accessors, and streams JSON. Do not rebuild any of it.
- Spend response bytes deliberately: `compress.Handler` for compressible content, default `ctx.JSON` options on hot paths, and `ctx.StreamWriter` for large or slow payloads.
- Bound requests: `BodyMaxMemory` (413 past the cap), `ctx.ReadJSONStream` for large arrays, `ctx.Paginated` instead of loading whole tables.
- Database wins are disciplines, not APIs: sized pools, no N+1, prepared hot statements, page-bounded queries. Caching (the biggest win of all) is Chapter 14.
- Background goroutines take `ctx.WithoutCancel()`, never the pooled `iris.Context`; bounded worker pools convert overload into fast `503`s instead of unbounded latency.

- Touch `GOGC` / `GOMEMLIMIT` only with a measurement on the screen.

Further Reading

- Profiling Go Programs: the classic walkthrough of finding and fixing a real hotspot with pprof.
- net/http/pprof: the standard library package behind the profiling endpoints and the full list of available profiles.
- A Guide to the Go Garbage Collector: how the GC works, what allocations cost, and what `GOGC` and `GOMEMLIMIT` actually change.
- Go documentation: Managing connections: the official guidance behind the `database/sql` pool settings used in this chapter.
- testing package: Benchmarks: the benchmark harness that `go test -bench` runs, including `b.N`, `ReportAllocs` and timer control.

CHAPTER 21

AI Backends

An AI backend is a web service whose responses are produced by a model rather than by a database, and that one difference reorganises the HTTP layer around it. The answer arrives in fragments over seconds or minutes, it is not all prose, the same endpoint has to serve it as a stream and as a single document, the thing producing it is a third party that fails in five distinct ways, and the client is entitled to leave halfway through. The `ai` package is the vocabulary for that shape. It renders a model's output on the wire, relays somebody else's, and classifies what went wrong upstream. It ships no vendor SDK and no provider schema: what travels over the wire is your business, and the shape of the wire is this package's.

Background

A language model generates its answer one token at a time, and each token depends on every token before it. There is no way to produce the last word faster than the first, so the total time is set by the length of the answer: a few hundred words is tens of seconds of work. Holding all of it back until the end therefore means a client that waits and shows nothing, which is why every AI product streams. The number that matters to a reader is not the duration but the time to the first fragment, because after that the text arrives at reading speed.

What the model emits is not only prose. Four other things travel on the same channel. Many models produce a reasoning trace, a running commentary on the problem that most interfaces render differently from the answer or hide behind a toggle. A model with tools available emits a tool call instead of text, and its arguments arrive in fragments that are not valid JSON until the last one lands. The provider reports usage, the token counts a turn cost, sometimes at the start and sometimes at the end and sometimes in both places. And a generation that started successfully can fail halfway, after the response has already begun with a 200.

That last point is the one that bends HTTP. A status code is written once, before the body, and a streamed answer commits it with the first fragment. Everything that goes wrong after that has to be reported inside the body, in a channel the client is already parsing as content, which means the transport needs a way to say "this ended badly" and a way to say "this ended". Truncation is the failure to worry about: a proxy idle timeout closes a stream cleanly, so a client that assumes a closed connection means a finished answer will silently show half an answer as the whole.

The wire format for all of this is Server-Sent Events, covered in Chapter 6, because a model stream is one-way text and a WebSocket buys nothing for it. The `ai` package sits on the `sse` package and adds the vocabulary: what the fragments are, how a non-streaming client gets the same answer as one document, and what to do when the upstream that produces them refuses.

One Handler, Both Shapes

Every OpenAI-shaped API takes a `stream` field in the request body and serves both answers from one endpoint: a stream of fragments when it is true, one JSON document when it is not. Writing that as two handlers duplicates the generation loop, and writing it as one handler with branches duplicates it inside. `ai.Open` removes the choice from the handler body:

```go
package main

import (
    "strings"

    "github.com/kataras/iris/v14"
    "github.com/kataras/iris/v14/ai"
    "github.com/kataras/iris/v14/rest/errors"
)

// ChatRequest is the request body. Stream mirrors what every
// OpenAI-shaped API puts there.
type ChatRequest struct {
    Prompt string `json:"prompt"`
    Stream bool   `json:"stream"`
}

// Validate runs automatically inside ctx.BindJSON.
func (r ChatRequest) Validate() error {
    if strings.TrimSpace(r.Prompt) == "" {
        return iris.ErrEmptyForm
    }

    return nil
}

func chat(ctx iris.Context) {
    req, ok := ctx.BindJSON[ChatRequest]()
    if !ok {
        return // BindJSON has already written the 400.
```

```
    }

    o := ai.DefaultOptions()
    o.Stream = &req.Stream // honour the request body.

    res, err := ai.Open(ctx, o)
    if errors.HandleError(ctx, err) {
        return
    }

    for delta := range generate(req.Prompt) {
        if err := res.Send(ai.Text("m1", delta)); err != nil {
            res.Close(err)
            return
        }
    }

    res.Send(ai.Meter(ai.Usage{InputTokens: 12, OutputTokens: 24}))
    res.Close(nil)
}
```

When the request asks to stream, each part goes out as it is produced. When it does not, the parts are accumulated and `Close` writes them as one JSON document. Nothing between `Open` and `Close` changes, and `res.Streaming()` reports which shape the turn took for the handful of cases that care.

`ai.DefaultOptions()` carries two other settings worth knowing before the first run. `Options.SSE` is the `sse.Options` from Chapter 6, so the heartbeat and the send deadline are configured there. And `Options.Coalesce` defaults to 100 milliseconds, which means consecutive text and reasoning deltas leave as at most one frame of each per interval rather than one frame per token. It batches parts rather than encoded bytes, because concatenating two rendered frames would destroy the structure the client is parsing. Zero disables it.

`Options.Stream` is a `*bool` rather than a `bool` because the zero value has to stay distinguishable from “not configured”. Left nil, `Open` negotiates on the `Accept` header and streams when the client asked for `text/event-stream`. A plain `bool` in a struct-valued option would mean “do not stream” for every caller who did not set it, which is the opposite of what half of them intend.

Like `sse.Open`, `ai.Open` writes nothing. An authorization check or a policy veto after it still produces an ordinary status code with an ordinary body, and `res.Committed()` reports whether that is still possible.

The Part Vocabulary

A `Part` is one increment of an answer. There are seven kinds, and each has a constructor:

Constructor	Kind	Carries
`ai.Text(id, delta)`	`KindText`	A fragment of the answer
`ai.Reasoning(id, delta)`	`KindReasoning`	A fragment of the reasoning trace
`ai.ToolCall(id, name, args)`	`KindToolCall`	A tool the model wants called, with raw JSON arguments
`ai.ToolResult(id, result)`	`KindToolResult`	What the tool returned
`ai.Meter(usage)`	`KindUsage`	What the turn cost
`ai.Error(err)`	`KindError`	A failure that happened after the response began
`ai.Data(name, raw)`	`KindData`	Anything this vocabulary does not model, passed through under its own name

`Part`'s fields are unexported, so an invalid combination cannot be constructed: there is no way to build a tool call carrying reasoning text or a usage part with a delta, and therefore no encoder has to defend against one. That is as close as Go gets to a sum type, and it turns the grammar into an invariant rather than a doc comment. Read a part back through `Kind()`, `ID()`, `Delta()`, `Name()`, `Args()`, `Result()`, `Usage()`, `Raw()` and `Err()`, each of which answers for the kinds that have it.

Seven kinds is a deliberate ceiling. A richer grammar of start, delta and end triples per content block buys compile-time sequence safety, but only to a framework that owns the generation loop, and this one deliberately does not. The `id` argument gives an interface the same correlation for a fraction of the surface: parts sharing an id belong to the same message.

`KindData` is the escape hatch and it earns its place. Providers add event types continuously, and a closed union would reject a valid-but-newer producer, so a relay passes an unrecognised event through under its own name instead of dropping it. Strict on emit, lenient on parse.

Encoders

An `Encoder` renders parts in a wire dialect. Three ship with the package:

Encoder	Wire	Use it for
`ai.SSEJSON()`	One SSE frame per part, named for its kind, carrying neutral JSON	The default. Browsers and anything that speaks SSE

`ai.NDJSON()`	One JSON object per line, no SSE framing, ending with `{"type":"done"}`	A server-to-server consumer, where the framing buys nothing and parsing is one `json.Decoder`
`ai.PlainText()`	The text deltas and nothing else	`curl`, and a client that wants prose rather than a protocol

`PlainText` drops reasoning, tool calls and usage, because the dialect has nowhere to put them, and it has neither a terminator nor an in-band error: a failure still reaches the logger and the collector, just not the reader. `NDJSON` has both, so its consumer detects truncation the way an SSE client does, by the missing terminal line; a failure after the first byte arrives as `{"type":"error","error":{...}}` before it, carrying the same `rest/errors` envelope the SSE error frame carries. Selecting an encoder is one field, so serving three formats from the handler above is one condition:

```go
if ctx.Query().Get("format") == "text" {
    o.Encoder = ai.PlainText()
}
```

The `Encoder` interface is the one seam through which a vendor shape may enter, and it points outward on purpose. The AI SDK UI Message Stream, AG-UI and the OpenAI `chat.completion.chunk` shapes are vendor response schemas, and a framework that ships one has signed up to track that vendor forever: that format alone changed its framing, its transport, its header name and its semantics inside eighteen months, and its header carries someone else's brand. Those belong in `iris-contrib`, written against this interface and pinned to a version. Implementing one means five methods (`ContentType`, `Header`, `Open`, `Encode`, `Close`) writing into a `Sink`, which offers `Frame` for an event stream and `Write` for a line-oriented dialect.

Reasoning Fenced in a Tag

Some providers return the reasoning trace in its own field, which decodes straight into an `ai.Reasoning` part. Others emit it inline, fenced by a tag such as `<think>`, in the same text channel as the answer. Splitting that with a `strings.Contains` on each delta does not work, because the tag arrives split across deltas as often as not: `"<th"` in one chunk and `"ink>"` in the next. The check misses the tag and the characters leak into the answer.

```go
splitter := ai.TagSplit("m1", "<think>", "</think>")

for delta := range generate(req.Prompt) {
    for _, part := range splitter.Write(delta) {
        if err := res.Send(part); err != nil {
```

```
            res.Close(err)
            return
        }
    }
}

// At the end of the stream, release whatever is still held back.
for _, part := range splitter.Flush() {
    res.Send(part)
}
```

`Write` holds back only what could still turn out to be the start of a tag, so at most `len(tag)-1` characters are ever delayed and a tag split across any number of chunks is recognised. `Flush` matters as much as `Write`: a tag that never completed is not a tag, its characters are part of the answer, and dropping them would truncate it silently. `Inside()` reports where the splitter currently is, and `StartInside()` covers the real and otherwise baffling dialect of a model that emits a closing tag having never emitted an opening one.

Policy and Accounting

Two hooks on `ai.Options` cover what a production turn needs around the generation itself.

`OnPart` runs before each part is encoded, and has three outcomes. Returning the part forwards it, possibly rewritten. Returning `ai.ErrDropPart` drops it and keeps the stream going. Returning any other error fails the turn. That is the whole policy surface, and it is block-only by construction: you cannot revise what has already gone out, so the restriction lives in the type rather than in a warning. A hook that could rewrite history would pass its tests and fail on the wire.

GO

```
o.OnPart = func(ctx iris.Context, p ai.Part) (ai.Part, error) {
    if p.Kind() == ai.KindReasoning && !ctx.Values().GetBool("staff") {
        return p, ai.ErrDropPart // internal readers only.
    }

    return p, nil
}
```

`OnFinish` runs exactly once after the turn ends, under `context.WithoutCancel`, with the assembled answer and the usage. It runs on success, on failure and on client disconnect alike, which is the whole reason it exists: a client that hung up is exactly when recording what the upstream already spent matters most. Persistence, billing and metrics belong here. The stream is an optimisation and never the source of truth, because a design in which the only copy of the answer is on the wire loses answers.

```go
o.OnFinish = func(ctx context.Context, r *ai.Result) {
    if err := store.SaveTurn(ctx, r.Text, r.Usage); err != nil {
        app.Logger().Errorf("saving turn: %v", err)
    }
}
```

The `ai.Result` it receives is also available from `res.Result()` and is complete after `Close`:

Field	What it holds
`Streaming`, `Committed`, `Disconnected`	Which shape the turn took, whether any byte reached the client, and whether the client left
`Abandoned`	The handler returned without calling `Close`. The framework ended the turn before the response was recycled: nothing was written after that, a pending part was dropped, and the client got no terminator
`Text`, `Reasoning`	The assembled answer and trace
`ToolCalls`	The tool-call parts, which already carry the id, the name and the raw arguments
`Usage`	Accumulated across every usage part the turn produced
`Parts`	Every part, when `Options.KeepParts` is set
`Err`	Why the turn ended badly, or nil
`TTFP`, `Duration`	Time to the first part, and the whole turn

`KeepParts` is off by default because a long generation would otherwise be held twice, once as assembled text and once as every delta that built it. `TTFP` is the field to watch: total duration is dominated by how long the answer is, so it measures the model rather than the service, while time to the first part is the latency a reader actually perceives.

`Usage` carries token counts and deliberately no money. Pricing a provider means tracking every provider's price list forever, and it changes more often than a framework ships. Compute cost in `OnFinish`, where you already know which model ran.

`OnDisconnect` decides what a vanished client does to the work in progress. `ai.DisconnectCancel`, the default, ends the turn as soon as the client goes away, because an abandoned generation burns tokens nobody will read. `ai.DisconnectDetach` keeps consuming the upstream so `OnFinish` records what was actually spent. Detaching does not mean the handler returns and work continues in the background: the handler still runs to completion, only the client is gone.

Relaying an Upstream

A service that calls a provider rather than running its own model is relaying: read an upstream event stream, translate each event into parts, forward them. `ai.Relay` is that loop.

```go
resp, err := httpClient.Do(req)
if ue := ai.Classify(resp, err); ue != nil {
    return res.Close(ue)
}

return res.Close(ai.Relay(res, resp, decoder, ai.RelayOptions{
    Read: sse.ReadOptions{Terminators: []string{"[DONE]"}},
}))
```

`Relay` closes the response body, classifies the response before reading it so a non-2xx upstream comes back as a typed error rather than as an empty stream, and checks the content type so a gateway that answers 200 with a JSON error does not silently become a blank answer. That last check is on by default and `RelayOptions.AllowNonEventStream` turns it off, handing the body to the decoder as a single event.

Every failed read of the upstream body comes back the same way: a reset, a timeout, a stream cut mid-event, or one closed cleanly without the sentinel `RelayOptions.Read.Terminators` promised, all pass through `Classify` as a transient upstream error wrapping the cause. The client gets a `BAD_GATEWAY` error frame (`DEADLINE_EXCEEDED` when the cause was a timeout) rather than a terminator claiming the answer was complete, and a gateway's failover logic sees the upstream failure it is. A client that hangs up mid-read stays `ClassCanceled`. `Relay` sets `sse.ReadOptions.RequireTerminator` whenever terminators are configured, with no opt-out.

The `Decoder` is your code, one method turning one upstream event into zero or more parts. The framework has no business knowing what a particular vendor calls a delta this month:

```go
decoder := ai.DecoderFunc(func(ev sse.Event) ([]ai.Part, error) {
    var chunk struct {
        Delta string `json:"delta"`
        Usage *struct {
            In  int64 `json:"in"`
            Out int64 `json:"out"`
        } `json:"usage"`
    }

    if err := json.Unmarshal(ev.Data, &chunk); err != nil {
        return nil, fmt.Errorf("decoding %q: %w", ev.Data, err)
    }

    var parts []ai.Part
    if chunk.Delta != "" {
        parts = append(parts, ai.Text("m1", chunk.Delta))
    }
    if chunk.Usage != nil {
        parts = append(parts, ai.Meter(ai.Usage{
            InputTokens:  chunk.Usage.In,
            OutputTokens: chunk.Usage.Out,
```

```
        }))
    }

    return parts, nil
})
```

Returning zero parts is normal and correct: providers emit keep-alives, role announcements and empty deltas that carry nothing worth forwarding.

Typed Upstream Failures

`ai.Classify` turns an upstream exchange into an `*ai.UpstreamError`, or nil when nothing went wrong. It takes the response and not only the error, and that signature is the whole point: a well-behaved Go HTTP client returns a non-2xx response with a nil error, so a `Classify(err)` form would have to recover the status code from a string and would recreate exactly the type erasure it exists to delete. The alternative is what AI gateways actually end up with, which is a regular expression over `err.Error()`.

The classification answers the only three questions a caller has: retry, try a different credential, or give up.

Class	Meaning	Retryable
`ai.ClassNone`	The response was fine	n/a
`ai.ClassFatal`	The request is wrong and would be just as wrong on another credential	No
`ai.ClassTransient`	A 5xx, a reset, a dial error, or a timeout	Yes
`ai.ClassRateLimited`	A 429	Yes, after `RetryAfter`
`ai.ClassExhausted`	This credential or account cannot serve the request at all	Yes, on another credential
`ai.ClassCanceled`	The caller went away, and only that: `context.Canceled`	No

A timeout is a transient failure, not a cancellation. Only `context.Canceled` is `ClassCanceled`; `context.DeadlineExceeded` and any `net.Error` whose `Timeout()` is true are transient, retryable, and reported as 504, with `ue.Timeout()` to tell them apart. The distinction matters because net/http's own timeout errors, from `http.Client.Timeout` and `Transport.ResponseHeaderTimeout`, satisfy `errors.Is(err, context.DeadlineExceeded)`: read as cancellations, a stalled provider

became an empty 200 that nothing retried and nothing reported. The one exception is `Relay` under `DisconnectCancel`: when the turn's own context ends, whatever ended it, the result is `ClassCanceled` with the cause kept, because nobody is waiting for a retry.

Two of those rows are opinions, and both are load-bearing. Rate limiting is its own class rather than a kind of exhaustion because it is a routing signal and not a health signal: a credential pool that counts a 429 as a fault quarantines a perfectly healthy key for being popular. And 401 and 403 map to exhaustion rather than fatal, because in a gateway they almost always mean this credential is finished rather than this request is malformed. Treating them as fatal is how one rotated key takes a whole service down instead of failing over.

The error carries what a support conversation needs: `Status`, `Class`, the provider's own `Message` and `Code`, the `RequestID` it reported in the body or a header, the `RetryAfter` it asked for, and up to 64 KB of the response `Body`, because the body is where providers put the reason. `ai.AsUpstream(err)` extracts it from an error chain and `Unwrap` exposes the transport error, so `errors.Is` against `context.Canceled` keeps working.

`HTTPStatus()` is the status code to give your own caller, and it is not the upstream's. A 401 from a provider means the gateway's credential is finished, which is a 502 to the caller: telling them they are unauthorised sends them off to check a key they do not have. The one code that passes through is 429, because rate limiting is the caller's problem too and the delay is worth forwarding. For anything more than the number, the next section is the fuller path: it produces the whole error envelope, logged and redacted, and it works after the response has already begun.

Errors After the First Byte

A failure before the first byte is an ordinary HTTP error, and Chapter 9 covers it. A failure after the first byte is the case a streaming API has to answer for, and it needs a different mechanism, because the status code is spent and the only channel left is the one the client is reading as content.

The temptation is to write the error's own text into a frame and move on. That loses twice. Nothing is logged or collected, so the failure is invisible to whatever sink the rest of the application reports to, and whatever the upstream said goes straight to the client, naming a vendor, a model or a host that the caller has no business learning.

`Close` therefore renders its error event through the `ErrorFrame` option, whose default runs the error through `errors.Report`: the route's mappings apply, the logger and the collector fire, an id is generated, and an unmapped or private error is redacted to a generic message. The frame

carries the same `rest/errors` envelope an ordinary failure would have produced. A turn opened with `ai.Open` installs an AI-aware version of that renderer, which adds one thing: a classified upstream failure keeps its canonical code instead of collapsing into a generic internal error.

`ai.Class`	Canonical code	Status
`ClassRateLimited`	`errors.ResourceExhausted`	429
`ClassExhausted`	`errors.Unavailable`	503
`ClassFatal`	`errors.InvalidArgument`, or `errors.TooLarge` for a 413	400, 413
`ClassTransient`, timed out (`ue.Timeout()`)	`errors.DeadlineExceeded`	504
`ClassTransient`, anything else	`errors.BadGateway`	502
`ClassCanceled`	none: nobody is listening	none

`ue.ErrorCode()` returns that code and `ue.Report(ctx)` runs the pipeline under it, returning the client-safe `*errors.Error` or nil when there is nothing to report. The client is told what the failure means and never what the provider said: a rate limit arrives as `RESOURCE_EXHAUSTED` with “the service is rate limited, try again shortly”, while the provider’s own message, code and request id go to the logger and the collector where they are useful. The error is wrapped as private on the way in, so the pipeline would redact it even if that wording were removed.

Because `Report` writes nothing, the same call serves both sides of the committed line. That is what lets a gateway hand its caller one envelope shape whether the upstream failed on the first attempt or halfway through the answer, which is the version the handler below uses.

Two options adjust the result. `Options.Redact` rewrites the client-facing message and detail last, after the pipeline has already made them safe, for the case it cannot cover on its own: an error the application deliberately mapped and described, whose description still names something internal. A `strings.Replacer` is the natural implementation and is safe to share across requests.

```go
o.Redact = strings.NewReplacer(
    "gpt-4o-mini", "the model",
    "api.openai.com", "the provider",
).Replace
```

And `OnPart` sees the terminal error too, as a `KindError` part, so the failure a client is told about passes the same gate everything else does. Returning `ai.ErrDropPart` there writes no error frame at all and the stream still gets its terminator, which is the shape for a client that should learn only that the answer ended.

A Gateway with Failover

Put those pieces together and a gateway is a loop over a chain of backends with one guard in it:

```go
// relayWithFailover walks the chain until one upstream serves the request.
func (g *gateway) relayWithFailover(ctx iris.Context, res *ai.Response) error {
    var lastErr error

    for _, b := range g.chain {
        // Once a delta has reached the client the status code is spent and
        // the answer has begun. Restarting now would splice two different
        // answers into one response, so this is where failover stops.
        if res.Committed() {
            return lastErr
        }

        err := g.relayOnce(ctx, res, b)
        if err == nil {
            return nil
        }
        lastErr = err

        ue, ok := ai.AsUpstream(err)
        if !ok {
            return err // our own bug: do not paper over it.
        }

        switch ue.Class {
        case ai.ClassCanceled:
            // Nobody is waiting for the answer any more.
            return err
        case ai.ClassFatal:
            // The request is wrong; it would be just as wrong on the next one.
            return err
        }

        // Transient, rate limited or exhausted: try the next backend.
    }

    return lastErr
}
```

`Committed()` is the guard that makes the difference between a gateway and a bug generator. Retrying is safe exactly until the first byte reaches the client, and after that a second attempt splices two different answers into one response, which reads as a model losing its mind rather than as an infrastructure failure.

The handler around it has to spend the status code when there is still one to spend, which is the branch a gateway forgets. The symptom is a 200 with an empty body:

```go
res, err := ai.Open(ctx, o)
```

```
if errors.HandleError(ctx, err) {
    return
}

err = res.Close(g.relayWithFailover(ctx, res))
if err == nil || res.Committed() {
    // Either it worked, or the answer had already begun and Close
    // reported the failure in-band. Both are finished responses.
    return
}

// Nothing reached the client, so the status code is still ours to spend.
// Report is the same call the streaming path makes, so the caller gets one
// envelope either way.
if ue, ok := ai.AsUpstream(err); ok {
    reported := ue.Report(ctx)
    if reported == nil {
        return // cancelled: nobody is listening.
    }

    if ue.RetryAfter > 0 {
        ctx.SetHeader("Retry-After", strconv.Itoa(int(ue.RetryAfter.Seconds())))
    }

    ctx.StopWithJSON(reported.ErrorCode.Status, reported)
    return
}

errors.Internal.Err(ctx, err)
```

One detail in the client that sits behind such a gateway is worth stating, because it is invisible until it truncates an answer. `http.Client.Timeout` covers the whole lifecycle including reading the body, so any non-zero value cuts a long generation off mid-sentence. Leave it at zero and let cancellation ride the request context, where it belongs. The full program, with two fake providers so the failover path is visible rather than theoretical, is `_examples/16-advanced/ai-gateway`.

What Breaks in Production

Five failures account for most of what goes wrong with a streaming endpoint, and none of them show up in a unit test.

`Timeout.HandlerTimeout` makes streaming impossible. It wraps the router in `http.TimeoutHandler`, whose response writer implements neither `http.Flusher` nor `http.Hijacker`, so every write buffers and the client gets a 503 page. `sse.Open` and `ai.Open` report `sse.ErrNotStreamable` before writing a byte, and an application that sets it logs a warning at startup. Bound slow handlers individually instead.

`Server.WriteTimeout` **is absolute, not idle.** It is a deadline on the whole response, not on one write, so a two-minute generation dies at the limit however busy the connection was. The first `Send` clears it for that response only, which is why streaming through the `sse` package survives a server configuration that would otherwise have to disable the timeout globally.

A buffering middleware de-streams everything. `middleware/compress` writes into a compressor and `middleware/recorder` writes into a `bytes.Buffer`, which `accesslog` installs whenever it reads the response body, and which `iris.NewBuilder().LogRequests(true)` therefore turns on. The first `Send` removes both for that response, so the combination works now; before it did, an access log was enough to turn every event stream into one lump on return.

A disconnected client is invisible for fifteen minutes. Only a real FIN or RST, or a failing write, cancels a request context. A closed lid or a dropped link produces none of those, and TCP retransmits for roughly 924 seconds on Linux before giving up, all of it billed upstream. The heartbeat is what turns that into a failed write within seconds, and `OnDisconnect` decides what happens next.

Shutdown does not cancel a request context. `http.Server.Shutdown` waits for handlers to return, and a handler blocked in a stream loop has no way to learn the process is stopping. Select on `ctx.ShuttingDown()` alongside the stream's own context, as the loop in Chapter 6 does, or graceful shutdown waits for its full timeout on every open stream.

One more is about the client rather than the server. `EventSource` cannot send a request body, which is why every LLM client is built on `fetch` instead, and why `Last-Event-ID` arrives as a query parameter as often as a header. `sse.Stream.LastEventID` reads both.

Summary

- An AI backend streams because a model's answer takes as long as the answer is long; the number a reader feels is the time to the first fragment, which `ai.Result.TTFP` measures.
- `ai.Open` serves the streaming and the non-streaming shape from one handler. `Options.Stream` is a `*bool` so that nil can mean "negotiate on `Accept`", and nothing between `Open` and `Close` changes between shapes.
- Seven `Part` kinds cover text, reasoning, tool calls, tool results, usage, errors and a `KindData` passthrough. `Part`'s fields are unexported, so an invalid part cannot be built.
- `ai.SSEJSON` is the default encoder, `ai.NDJSON` suits a server-to-server consumer and `ai.PlainText` suits `curl`. Vendor dialects belong behind the `Encoder` interface, not inside the package.

- `ai.TagSplit` separates reasoning fenced in a tag such as `<think>` across chunk boundaries, and `Flush` releases a tag that never completed rather than truncating the answer.
- `OnPart` is block-only policy; `OnFinish` runs exactly once under `context.WithoutCancel`, on success, failure and disconnect alike, and is where persistence, billing and metrics belong.
- `ai.Relay` forwards an upstream stream through a `Decoder` you write. `ai.Classify` takes the response as well as the error and returns a typed `*ai.UpstreamError` whose `Class` says whether to retry, switch credential or stop, and whose `HTTPStatus` is what to tell your caller.
- `Committed()` is where failover has to stop, because after the first delta a retry splices two answers into one response.
- A failure after the first byte goes out as an error frame rendered through the `rest/errors` pipeline, so it is mapped, logged, collected and redacted. A classified upstream failure keeps its canonical code: `ClassRateLimited` reaches the client as `RESOURCE_EXHAUSTED`, not as a generic internal error. `Options.Redact` rewrites the client-facing text last, and `OnPart` can drop the error frame entirely.

Further Reading

- WHATWG HTML: Server-sent events: the wire format underneath every part this chapter sends.
- MDN: Using the Fetch API for streams: reading a stream in the browser, which is what a client does when `EventSource` cannot POST the request body.
- pkg.go.dev: iris/v14/ai: the full API, with the reasoning for each default in its doc comment.
- pkg.go.dev: iris/v14/sse: the transport this package is built on.
- RFC 6585: Additional HTTP Status Codes: 429 and the `Retry-After` semantics a gateway forwards.

CHAPTER 22

Real-World Examples

Every previous chapter isolated one topic. This chapter does the opposite: it presents two complete applications, listed file by file, with nothing elided. The first (a URL shortener) uses `iris.New()` and two source files, the lightweight shape for small programs. The second (an inventory service for warehouses and stock) uses the full recommended stack: `iris.NewBuilder`, dependency injection, controllers, a central error map, a protected admin surface, startup and shutdown hooks, and an `httptest` suite. Both programs compile and their tests pass exactly as printed.

Background

How to read a complete program listing

Each code block below belongs to one real file, named in the text right before it. Copy the blocks of an application into files with those names, run the setup commands shown at its start, and you have the working program. The same files were built and tested with Go 1.27 against Iris v14 before being pasted here, and the test output in this chapter is their real output.

Read a listing the way you would review a colleague's pull request, not the way you read a novel. Start at the file tree to take in the shape, then read the innermost package first (the one that imports the least) and finish with the file that wires everything together. For Stockroom that order is `inventory`, then `api`, then `main.go`, which is exactly the order the chapter prints them in.

What separates a toy from a production-shaped service

A toy program answers requests. A production-shaped service also answers the questions that arrive later: how do settings differ between your laptop and the server, who may call the dangerous endpoints, do all failures come out in one consistent shape, and how do you prove

any of it still works after a change. Four ingredients cover those questions, and all four appear in Stockroom:

- **Configuration**: settings live outside the code, in a file or in the process environment, so the same compiled binary runs in development and in production; secrets never sit in committed files.
- **Guards**: small checks that run before handlers do, so a rule such as "sealed warehouses are read-only" is enforced in one place instead of being repeated inside every handler.
- **One error map**: every failure the business logic can produce is translated to an HTTP status code exactly once, so clients see the same response shape for the same problem everywhere.
- **Tests**: an in-memory suite proves the routes, guards and error map behave, without a running server or a real database.

Why two shapes

The two applications deliberately share nothing, because program size honestly changes what good structure looks like. Shortl shows how little ceremony a small Iris program needs; any layering there would be ceremony for its own sake. Stockroom shows where every piece of the recommended architecture goes when the program is meant to grow. Neither is a lesser version of the other; the closing section lists the concrete signals for choosing, and growing from the first shape into the second is mechanical rather than a rewrite.

Shortl: A URL Shortener

Shortl stores a target URL under a random six-letter code and redirects visitors of `/{code}` to it, counting hits. Three routes, two files:

Method	Path	Purpose
POST	/api/links	create a short link
GET	/api/links/{code}	link statistics
GET	/{code}	redirect to the target

```
shortl/
├── go.mod
├── main.go        # routes and validation
├── store.go       # in-memory store behind a mutex
└── main_test.go   # httptest suite
```

Setup:

```sh
mkdir shortl && cd shortl
go mod init shortl
go get github.com/kataras/iris/v14@latest
```

Run `go mod tidy` after pasting the files (the test file uses one more module, the `httpexpect` library that `httptest` builds on).

The Store

A Go server handles many requests at once, each on its own goroutine, a lightweight thread managed by the Go runtime. Two goroutines writing to the same map at the same moment corrupt it, so shared data needs a lock. A mutex (mutual-exclusion lock) admits one goroutine at a time; the read-write variant used here lets any number of readers in simultaneously while a writer waits for exclusive access.

The store is ordinary Go: a map guarded by a read-write mutex, a random code generator, and a retry loop for the rare collision. Codes are built from letters only: that is not an aesthetic choice, it is what lets the route macro type-check them, as you will see in `main.go`.

`store.go`:

```go
// store.go
package main

import (
    "errors"
    "math/rand/v2"
    "sync"
    "time"
)

// Link is a stored short link.
type Link struct {
    Code      string    `json:"code"`
    URL       string    `json:"url"`
    Hits      int64     `json:"hits"`
    CreatedAt time.Time `json:"created_at"`
}

// ErrCodeSpaceExhausted is returned when no free code can be found.
var ErrCodeSpaceExhausted = errors.New("shortl: code space exhausted")

// codeAlphabet holds letters only, so every generated code matches the
// {code:alphabetical} route macro. Ambiguous glyphs (l, I, O) are left out.
const (
    codeAlphabet = "abcdefghijkmnpqrstuvwxyzABCDEFGHJKMNPQRSTUVWXYZ"
    codeLength   = 6
)

// Store is an in-memory link store guarded by a read-write mutex.
```

```
type Store struct {
    mu    sync.RWMutex
    links map[string]*Link
}

func NewStore() *Store {
    return &Store{links: make(map[string]*Link)}
}

// Save stores the target URL under a fresh random code.
func (s *Store) Save(url string) (Link, error) {
    s.mu.Lock()
    defer s.mu.Unlock()

    for range 100 {
        code := randomCode()
        if _, taken := s.links[code]; taken {
            continue
        }

        link := &Link{
            Code:      code,
            URL:       url,
            CreatedAt: time.Now().UTC(),
        }
        s.links[code] = link
        return *link, nil
    }
    return Link{}, ErrCodeSpaceExhausted
}

// Resolve returns the target URL of a code and counts the hit.
func (s *Store) Resolve(code string) (string, bool) {
    s.mu.Lock()
    defer s.mu.Unlock()

    link, ok := s.links[code]
    if !ok {
        return "", false
    }
    link.Hits++
    return link.URL, true
}

// Stats returns a snapshot of one link.
func (s *Store) Stats(code string) (Link, bool) {
    s.mu.RLock()
    defer s.mu.RUnlock()

    link, ok := s.links[code]
    if !ok {
        return Link{}, false
    }
    return *link, true
}

func randomCode() string {
    b := make([]byte, codeLength)
    for i := range b {
        b[i] = codeAlphabet[rand.IntN(len(codeAlphabet))]
    }
    return string(b)
```

```
}
```

The Application

The whole HTTP surface fits in one file. Three details carry it.

First, the input type implements `Validate() error`, which makes `ctx.BindJSON` run it immediately after decoding; the handler never calls it. The URL check is a custom `Func` chained onto the `validation.String` builder: return an empty string for a valid value, a reason for an invalid one, and the reason lands in the structured 400 payload with its field name.

Second, the catch-all route is typed: `/{code:alphabetical}` matches letters (and spaces) only, per the macro's `^[a-zA-Z ]+$` rule. Requests like `/favicon.ico` or `/robots.txt` never reach the handler: the router answers 404 on its own. The static `/api` prefix always wins over the parameterized root route, because static segments sort before parameters (Chapter 4).

Third, `iris.New(iris.Options{Name: "shortl"})` passes a partial options literal: it merges with the defaults, configuring only what it sets.

`main.go`:

GO

```go
// Shortl is a URL shortener: the smallest complete Iris program that is
// still honest about input validation and error responses.
package main

import (
    "net/url"

    "github.com/kataras/iris/v14"
    "github.com/kataras/iris/v14/rest/errors"
    "github.com/kataras/iris/v14/rest/errors/validation"
)

// ShortenInput is the create-link payload. Implementing Validate makes
// ctx.BindJSON run it right after decoding: handlers never call it.
type ShortenInput struct {
    URL string `json:"url"`
}

func (in ShortenInput) Validate() error {
    return validation.Join(
        validation.String("url", in.URL).NotEmpty().
            Func(absoluteHTTPURL),
    )
}

// absoluteHTTPURL is a custom validation.Func: empty string means valid.
func absoluteHTTPURL(s string) string {
    u, err := url.Parse(s)
    if err != nil || u.Host == "" ||
        (u.Scheme != "http" && u.Scheme != "https") {
        return "must be an absolute http(s) URL"
    }
```

```
    return ""
}

// newApp wires the routes around a store. Tests call it too.
func newApp(store *Store) *iris.Application {
    app := iris.New(iris.Options{Name: "shortl"})

    api := app.Router("/api")

    api.Post("/links", func(ctx iris.Context) {
        input, ok := ctx.BindJSON[ShortenInput]()
        if !ok {
            return // 400 already written: bad JSON or failed Validate.
        }

        link, err := store.Save(input.URL)
        ctx.Created(link, err) // 201 with the full Link as JSON.
    })

    api.Get("/links/{code:alphabetical}", func(ctx iris.Context) {
        link, ok := store.Stats(ctx.Params().GetString("code"))
        if !ok {
            errors.NotFound.Message(ctx, "unknown short code")
            return
        }
        ctx.OK(link, nil)
    })

    // The typed macro keeps this catch-all narrow: only letter codes
    // reach the handler, everything else is a plain 404.
    app.Get("/{code:alphabetical}", func(ctx iris.Context) {
        target, ok := store.Resolve(ctx.Params().GetString("code"))
        if !ok {
            errors.NotFound.Message(ctx, "unknown short code")
            return
        }
        ctx.Redirect(target, iris.StatusMovedPermanently)
    })

    return app
}

func main() {
    app := newApp(NewStore())
    app.Listen(":8080")
}
```

Even without the builder, the `rest` and `rest/errors` helpers work at full strength: `ctx.BindJSON` renders its own field-level 400, and the `errors.NotFound.Message` sender writes the canonical JSON error shape (Chapter 9).

The Tests

The suite drives the real application in memory through `httptest.New`. One technique is worth pausing on: the test client follows redirects by default, which would turn the redirect assertion into a request against the target. `WithRedirectPolicy(httpexpect.DontFollowRedirects)` (from the

`httpexpect` library that `httptest` re-exposes) freezes the response so the test can inspect the 301 and its `Location` header.

`main_test.go`:

```go
// main_test.go
package main

import (
    "testing"

    "github.com/iris-contrib/httpexpect/v2"

    "github.com/kataras/iris/v14/httptest"
)

func TestShortenResolveStats(t *testing.T) {
    e := httptest.New(t, newApp(NewStore()))

    // Create a link; the response is the stored Link.
    obj := e.POST("/api/links").
        WithJSON(ShortenInput{URL: "https://iris-go.com/docs"}).
        Expect().Status(httptest.StatusCreated).JSON().Object()
    obj.HasValue("url", "https://iris-go.com/docs")
    code := obj.Value("code").String().NotEmpty().Raw()

    // The redirect: do not follow it, inspect it.
    e.GET("/"+code).
        WithRedirectPolicy(httpexpect.DontFollowRedirects).
        Expect().Status(httptest.StatusMovedPermanently).
        Header("Location").IsEqual("https://iris-go.com/docs")

    // The hit above was counted.
    e.GET("/api/links/"+code).
        Expect().Status(httptest.StatusOK).
        JSON().Object().HasValue("hits", 1)
}

func TestValidationAndNotFound(t *testing.T) {
    e := httptest.New(t, newApp(NewStore()))

    // Validate rejects a relative URL with a field-level payload.
    e.POST("/api/links").
        WithJSON(ShortenInput{URL: "not-a-url"}).
        Expect().Status(httptest.StatusBadRequest).
        JSON().Object().Value("validation").Array().NotEmpty()

    // Unknown code: matched by the macro, missing in the store.
    e.GET("/nosuch").
        WithRedirectPolicy(httpexpect.DontFollowRedirects).
        Expect().Status(httptest.StatusNotFound)
}
```

Run It

```sh
go mod tidy && go test ./...   # ok  shortl
go run .
```

```sh
curl -X POST localhost:8080/api/links \
  -H 'Content-Type: application/json' \
  -d '{"url":"https://iris-go.com"}'


curl -i localhost:8080/XkfPam


curl -X POST localhost:8080/api/links \
  -H 'Content-Type: application/json' -d '{"url":"not-a-url"}'
```

Stockroom: An Inventory API

Stockroom manages warehouses and the stock items inside them. One business rule shapes the whole service: a *sealed* warehouse is read-only. Nothing can be added, withdrawn or deleted until an administrator unseals it. Unsealing is deliberately absent from the public API. A second rule guards quantities: a withdrawal that would drive an item's balance below zero is refused.

Method	Path	Purpose	Protection
GET	/api/warehouses	list warehouses	—
POST	/api/warehouses	create a warehouse	validation
GET	/api/warehouses/{id}	one warehouse	—
DELETE	/api/warehouses/{id}	delete (refused when sealed)	—
POST	/api/warehouses/{id}/seal	seal a warehouse	—
GET	/api/warehouses/{wid}/items	list items, paginated	warehouse exists
POST	/api/warehouses/{wid}/items	add an item	warehouse unsealed
GET	/api/warehouses/{wid}/items/{id}	one item	warehouse exists
DELETE	/api/warehouses/{wid}/items/{id}	remove an item	warehouse unsealed
POST	/api/warehouses/{wid}/items/{id}/withdraw	remove units	warehouse unsealed
GET	/api/admin/stats	totals	basic auth
POST	/api/admin/warehouses/{id}/unseal	force-unseal	basic auth

GET	/health	health check	—

The layout keeps one direction of dependency: `api` depends on `inventory`; `inventory` knows nothing about HTTP.

```
stockroom/
├── go.mod
├── config.yml            # committed settings; secrets stay in the env
├── main.go               # configuration, wiring, startup
├── main_test.go          # HTTP tests with fake repositories
├── api/
│   ├── warehouses.go     # WarehousesAPI: CRUD, seal, nested router
│   ├── items.go          # ItemsAPI: guards, CRUD, withdraw, pagination
│   └── admin.go          # AdminAPI: stats, force-unseal
└── inventory/
    ├── inventory.go      # entities, inputs, sentinel errors
    ├── repository.go     # the storage interface (the test seam)
    ├── memstore.go       # in-memory implementation
    └── service.go        # business rules and warmup
```

Setup:

```sh
mkdir stockroom && cd stockroom
go mod init stockroom
go get github.com/kataras/iris/v14@latest
go get gopkg.in/yaml.v3@latest
```

Run `go mod tidy` after pasting the files below.

The Domain

Two terms carry this section. The *domain* is the part of a program that speaks the business language (warehouses, items, quantities) and knows nothing about transport concerns such as HTTP. A *sentinel error* is a named, package-level error value created once with `errors.New`; failing code returns one of the fixed set instead of inventing a new message on the spot, so callers recognize each failure by identity.

The `inventory` package defines the entities, the request inputs and three sentinel errors. The sentinels are the domain's whole vocabulary of failure, with no status codes here. The HTTP layer maps them exactly once, in `main.go`. Each input type implements `Validate() error` with the `rest/errors/validation` builders, so `ctx.BindJSON` rejects bad payloads with field-level detail before any handler logic runs. (UUIDs come from the standard library's `uuid` package, part of Go since 1.27, so nothing beyond Iris and the YAML reader needs fetching.)

`inventory/inventory.go`:

```go
// Package inventory is the domain of the Stockroom service: entities,
```

```
// request inputs and business errors. It imports no HTTP types.
package inventory

import (
    "errors"
    "time"
    "uuid"

    "github.com/kataras/iris/v14/rest/errors/validation"
)

// Sentinel errors: the service and the store return these, the HTTP
// layer maps them to status codes exactly once, at startup.
var (
    ErrNotFound          = errors.New("inventory: not found")
    ErrSealed            = errors.New("inventory: warehouse is sealed")
    ErrInsufficientStock = errors.New("inventory: insufficient stock")
)

// Warehouse holds stock items. A sealed warehouse is read-only: no new
// items, no withdrawals, no deletion — until an admin unseals it.
type Warehouse struct {
    ID        uuid.UUID `json:"id"`
    Name      string    `json:"name"`
    Region    string    `json:"region"`
    Sealed    bool      `json:"sealed"`
    CreatedAt time.Time `json:"created_at"`
}

// StockItem is one SKU inside one warehouse.
type StockItem struct {
    ID          uuid.UUID `json:"id"`
    WarehouseID uuid.UUID `json:"warehouse_id"`
    SKU         string    `json:"sku"`
    Name        string    `json:"name"`
    Quantity    int       `json:"quantity"`
    CreatedAt   time.Time `json:"created_at"`
}

// WarehouseInput is the create payload. Implementing Validate makes
// ctx.BindJSON run it after decoding.
type WarehouseInput struct {
    Name   string `json:"name"`
    Region string `json:"region"`
}

func (in WarehouseInput) Validate() error {
    return validation.Join(
        validation.String("name", in.Name).NotEmpty().Length(1, 100),
        validation.String("region", in.Region).NotEmpty(),
    )
}

// ItemInput is the create payload for a stock item.
type ItemInput struct {
    SKU      string `json:"sku"`
    Name     string `json:"name"`
    Quantity int    `json:"quantity"`
}

func (in ItemInput) Validate() error {
    return validation.Join(
```

```
        validation.String("sku", in.SKU).NotEmpty().Length(3, 32),
        validation.String("name", in.Name).NotEmpty(),
        validation.Number("quantity", in.Quantity).Positive(),
    )
}

// WithdrawInput asks to remove units from a stock item.
type WithdrawInput struct {
    Quantity int `json:"quantity"`
}

func (in WithdrawInput) Validate() error {
    return validation.Join(
        validation.Number("quantity", in.Quantity).Positive(),
    )
}

// ItemFilter narrows item listings; it arrives as an optional JSON body
// next to the ?page=&size= query (see ctx.Paginated).
type ItemFilter struct {
    SKU string `json:"sku"`
}

// Stats is the admin dashboard summary. Units is the sum of all
// item quantities.
type Stats struct {
    Warehouses int64 `json:"warehouses"`
    Items      int64 `json:"items"`
    Units      int64 `json:"units"`
}
```

The Storage Seam

A Go *interface* names a set of methods without naming who implements them; any type with those methods satisfies it automatically. That indirection creates a seam: a line along which one implementation swaps for another (a real database in production, an in-memory fake in tests) without the calling code changing. Two more words below deserve definitions: a *race* is two concurrent requests interleaving in an unlucky order (both read a balance of 5, both withdraw 5), leaving a state neither intended; *atomic* means a check and its write happen as one indivisible step, so no other request can slip between them.

The service talks to storage through an interface, the seam that makes the application testable. Every method takes a `context.Context` first; `iris.Context` implements it, so request cancellation reaches the store with no extra code. One method deserves attention: `AdjustItemQuantity` is atomic and owns the non-negative balance invariant, because a check-then-write across two calls would race.

`inventory/repository.go`:

```
// inventory/repository.go
package inventory
```
GO

```
import (
    "context"
    "uuid"

    "github.com/kataras/iris/v14"
)

// Repository is the storage seam: the service depends on this interface,
// production wires a database implementation, tests wire MemStore or a
// stub. Every method takes a context.Context first — iris.Context
// implements it, so request cancellation reaches the store for free.
type Repository interface {
    CreateWarehouse(ctx context.Context, w Warehouse) error
    Warehouses(ctx context.Context) ([]Warehouse, error)
    // Warehouse returns ErrNotFound for an unknown id.
    Warehouse(ctx context.Context, id uuid.UUID) (Warehouse, error)
    // SetWarehouseSealed reports false when the flag did not change.
    SetWarehouseSealed(ctx context.Context, id uuid.UUID,
        sealed bool) (bool, error)
    DeleteWarehouse(ctx context.Context, id uuid.UUID) (bool, error)

    CreateItem(ctx context.Context, item StockItem) error
    Items(ctx context.Context, warehouseID uuid.UUID,
        opts iris.PageOptions, filter ItemFilter) ([]StockItem, int64, error)
    // Item returns ErrNotFound for an unknown pair.
    Item(ctx context.Context, warehouseID, id uuid.UUID) (StockItem, error)
    // AdjustItemQuantity atomically adds delta to the quantity. It
    // returns ErrNotFound for an unknown item and ErrInsufficientStock
    // when the result would go below zero.
    AdjustItemQuantity(ctx context.Context, warehouseID, id uuid.UUID,
        delta int) (StockItem, error)
    DeleteItem(ctx context.Context, warehouseID, id uuid.UUID) (bool, error)

    Counts(ctx context.Context) (warehouses, items, units int64, err error)
}
```

The in-memory implementation is a mutex and two maps: development storage today, the test suite's honest fake forever. A database implementation replaces it later by changing one `Services` entry.

`inventory/memstore.go`:

```
// inventory/memstore.go
package inventory

import (
    "context"
    "sort"
    "strings"
    "sync"
    "uuid"

    "github.com/kataras/iris/v14"
)

// MemStore is an in-memory Repository: development storage and the test
// suite's fake. Swapping it for a database changes one Services entry.
type MemStore struct {
```

GO

```
    mu         sync.RWMutex
    warehouses map[uuid.UUID]Warehouse
    items      map[uuid.UUID]StockItem
}

var _ Repository = (*MemStore)(nil)

func NewMemStore() *MemStore {
    return &MemStore{
        warehouses: make(map[uuid.UUID]Warehouse),
        items:      make(map[uuid.UUID]StockItem),
    }
}

func (s *MemStore) CreateWarehouse(_ context.Context, w Warehouse) error {
    s.mu.Lock()
    defer s.mu.Unlock()

    s.warehouses[w.ID] = w
    return nil
}

func (s *MemStore) Warehouses(_ context.Context) ([]Warehouse, error) {
    s.mu.RLock()
    defer s.mu.RUnlock()

    list := make([]Warehouse, 0, len(s.warehouses))
    for _, w := range s.warehouses {
        list = append(list, w)
    }
    sort.Slice(list, func(i, j int) bool {
        return list[i].CreatedAt.Before(list[j].CreatedAt)
    })
    return list, nil
}

func (s *MemStore) Warehouse(_ context.Context,
    id uuid.UUID) (Warehouse, error) {
    s.mu.RLock()
    defer s.mu.RUnlock()

    w, ok := s.warehouses[id]
    if !ok {
        return Warehouse{}, ErrNotFound
    }
    return w, nil
}

func (s *MemStore) SetWarehouseSealed(_ context.Context, id uuid.UUID,
    sealed bool) (bool, error) {
    s.mu.Lock()
    defer s.mu.Unlock()

    w, ok := s.warehouses[id]
    if !ok {
        return false, ErrNotFound
    }
    if w.Sealed == sealed {
        return false, nil
    }
    w.Sealed = sealed
    s.warehouses[id] = w
```

```
    return true, nil
}

func (s *MemStore) DeleteWarehouse(_ context.Context,
    id uuid.UUID) (bool, error) {
    s.mu.Lock()
    defer s.mu.Unlock()

    if _, ok := s.warehouses[id]; !ok {
        return false, nil
    }

    delete(s.warehouses, id)
    for itemID, item := range s.items {
        if item.WarehouseID == id {
            delete(s.items, itemID)
        }
    }
    return true, nil
}

func (s *MemStore) CreateItem(_ context.Context, item StockItem) error {
    s.mu.Lock()
    defer s.mu.Unlock()

    s.items[item.ID] = item
    return nil
}

func (s *MemStore) Items(_ context.Context, warehouseID uuid.UUID,
    opts iris.PageOptions, filter ItemFilter) ([]StockItem, int64, error) {
    s.mu.RLock()
    defer s.mu.RUnlock()

    matched := make([]StockItem, 0)
    for _, item := range s.items {
        if item.WarehouseID != warehouseID {
            continue
        }
        if filter.SKU != "" &&
            !strings.EqualFold(item.SKU, filter.SKU) {
            continue
        }
        matched = append(matched, item)
    }
    sort.Slice(matched, func(i, j int) bool {
        return matched[i].CreatedAt.Before(matched[j].CreatedAt)
    })

    total := int64(len(matched))

    // Apply the page window (a SQL implementation uses LIMIT/OFFSET).
    offset, limit := opts.GetOffset(), opts.GetLimit()
    if offset >= len(matched) {
        return []StockItem{}, total, nil
    }
    end := offset + limit
    if limit == 0 || end > len(matched) {
        end = len(matched)
    }
    return matched[offset:end], total, nil
}
```

```
func (s *MemStore) Item(_ context.Context,
    warehouseID, id uuid.UUID) (StockItem, error) {
    s.mu.RLock()
    defer s.mu.RUnlock()

    item, ok := s.items[id]
    if !ok || item.WarehouseID != warehouseID {
        return StockItem{}, ErrNotFound
    }
    return item, nil
}

func (s *MemStore) AdjustItemQuantity(_ context.Context,
    warehouseID, id uuid.UUID, delta int) (StockItem, error) {
    s.mu.Lock()
    defer s.mu.Unlock()

    item, ok := s.items[id]
    if !ok || item.WarehouseID != warehouseID {
        return StockItem{}, ErrNotFound
    }
    if item.Quantity+delta < 0 {
        return StockItem{}, ErrInsufficientStock
    }
    item.Quantity += delta
    s.items[id] = item
    return item, nil
}

func (s *MemStore) DeleteItem(_ context.Context,
    warehouseID, id uuid.UUID) (bool, error) {
    s.mu.Lock()
    defer s.mu.Unlock()

    item, ok := s.items[id]
    if !ok || item.WarehouseID != warehouseID {
        return false, nil
    }
    delete(s.items, id)
    return true, nil
}

func (s *MemStore) Counts(_ context.Context) (int64, int64, int64, error) {
    s.mu.RLock()
    defer s.mu.RUnlock()

    var units int64
    for _, item := range s.items {
        units += int64(item.Quantity)
    }
    return int64(len(s.warehouses)), int64(len(s.items)), units, nil
}
```

Rules and Warmup

The service owns the business rules. `DeleteWarehouse` refuses a sealed warehouse; handlers never check the flag for deletion. They forward whatever the service returns. The service also implements `iris.Initializer`: because it is registered through the builder's `Services` step, its

`Init(ctx)` runs exactly once inside `Build()`, after dependency injection and before the server accepts traffic. Here it proves storage is reachable and logs the counters, the startup mirror of the `Deferrables` shutdown hooks. A failing `Init` stops the application from building, which is exactly what you want from a broken database at boot.

`inventory/service.go`:

GO

```go
// inventory/service.go
package inventory

import (
    "context"
    "fmt"
    "log"
    "time"
    "uuid"

    "github.com/kataras/iris/v14"
)

// Service holds the business rules. Its constructor declares what it
// needs; the iris dependency container calls it with the registered
// Repository.
type Service struct {
    repo Repository
}

func NewService(repo Repository) *Service {
    return &Service{repo: repo}
}

// Init implements iris.Initializer: it runs once inside Build, after
// dependency injection and before the server accepts traffic. Here it
// proves the storage is reachable and warms the counters.
func (s *Service) Init(ctx context.Context) error {
    warehouses, items, units, err := s.repo.Counts(ctx)
    if err != nil {
        return fmt.Errorf("inventory: warmup: %w", err)
    }

    log.Printf("inventory: ready (%d warehouses, %d items, %d units)",
        warehouses, items, units)
    return nil
}

func (s *Service) CreateWarehouse(ctx context.Context,
    in WarehouseInput) (uuid.UUID, error) {
    w := Warehouse{
        ID:        uuid.New(),
        Name:      in.Name,
        Region:    in.Region,
        CreatedAt: time.Now().UTC(),
    }
    return w.ID, s.repo.CreateWarehouse(ctx, w)
}

func (s *Service) Warehouses(ctx context.Context) ([]Warehouse, error) {
    return s.repo.Warehouses(ctx)
}
```

```
func (s *Service) Warehouse(ctx context.Context,
    id uuid.UUID) (Warehouse, error) {
    return s.repo.Warehouse(ctx, id)
}

// Seal freezes a warehouse. Sealing twice is not an error: the false
// result renders as 304 Not Modified.
func (s *Service) Seal(ctx context.Context, id uuid.UUID) (bool, error) {
    return s.repo.SetWarehouseSealed(ctx, id, true)
}

// Unseal reverses Seal; only the admin API exposes it.
func (s *Service) Unseal(ctx context.Context, id uuid.UUID) (bool, error) {
    return s.repo.SetWarehouseSealed(ctx, id, false)
}

// DeleteWarehouse refuses to delete a sealed warehouse; that rule lives
// here, not in a handler.
func (s *Service) DeleteWarehouse(ctx context.Context,
    id uuid.UUID) (bool, error) {
    w, err := s.repo.Warehouse(ctx, id)
    if err != nil {
        return false, err
    }
    if w.Sealed {
        return false, ErrSealed
    }
    return s.repo.DeleteWarehouse(ctx, id)
}

func (s *Service) CreateItem(ctx context.Context, warehouseID uuid.UUID,
    in ItemInput) (uuid.UUID, error) {
    item := StockItem{
        ID:          uuid.New(),
        WarehouseID: warehouseID,
        SKU:         in.SKU,
        Name:        in.Name,
        Quantity:    in.Quantity,
        CreatedAt:   time.Now().UTC(),
    }
    return item.ID, s.repo.CreateItem(ctx, item)
}

func (s *Service) Items(ctx context.Context, warehouseID uuid.UUID,
    opts iris.PageOptions, filter ItemFilter) ([]StockItem, int64, error) {
    return s.repo.Items(ctx, warehouseID, opts, filter)
}

func (s *Service) Item(ctx context.Context,
    warehouseID, id uuid.UUID) (StockItem, error) {
    return s.repo.Item(ctx, warehouseID, id)
}

// Withdraw removes units from a stock item. The repository adjusts
// atomically and returns ErrInsufficientStock when the balance would go
// negative — mapped to 409 Conflict by the central error map.
func (s *Service) Withdraw(ctx context.Context, warehouseID, id uuid.UUID,
    in WithdrawInput) (StockItem, error) {
    return s.repo.AdjustItemQuantity(ctx, warehouseID, id, -in.Quantity)
}
```

```
func (s *Service) DeleteItem(ctx context.Context,
    warehouseID, id uuid.UUID) (bool, error) {
    return s.repo.DeleteItem(ctx, warehouseID, id)
}

func (s *Service) Stats(ctx context.Context) (Stats, error) {
    warehouses, items, units, err := s.repo.Counts(ctx)
    return Stats{Warehouses: warehouses, Items: items, Units: units}, err
}
```

The Warehouses Controller

A controller here is a plain struct with a constructor (its dependency declaration) and a `Routing(r iris.Router)` method (its route table). The container calls `NewWarehousesAPI` with the registered `*inventory.Service`; the router calls `Routing` once at startup. The `{id:uuid}` macros reject malformed ids before any handler runs; in builder-built applications the macro error handler renders them as 400 `INVALID_ARGUMENT` (“invalid path parameter”).

`api/warehouses.go`:

GO

```
// Package api is the HTTP layer of the Stockroom service: thin handlers
// that read input, call the service and hand the result to rest helpers.
package api

import (
    "github.com/kataras/iris/v14"

    "stockroom/inventory"
)

// WarehousesAPI groups the /warehouses handlers around their
// dependencies. NewWarehousesAPI is registered in the builder's API
// step; the container calls it with the registered *inventory.Service.
type WarehousesAPI struct {
    svc *inventory.Service
}

func NewWarehousesAPI(svc *inventory.Service) *WarehousesAPI {
    return &WarehousesAPI{svc: svc}
}

// Routing implements the iris controller convention: the router calls
// it once at startup.
func (api *WarehousesAPI) Routing(r iris.Router) {
    r.Get("/", api.list)
    r.Post("/", api.create)
    r.Get("/{id:uuid}", api.get)
    r.Delete("/{id:uuid}", api.delete)
    r.Post("/{id:uuid}/seal", api.seal)

    // Everything about a warehouse's items lives in its own nested
    // router; the container resolves NewItemsAPI like any controller.
    r.Router("/{warehouse_id:uuid}/items", NewItemsAPI)
}

func (api *WarehousesAPI) list(ctx iris.Context) {
```

```
    warehouses, err := api.svc.Warehouses(ctx)
    ctx.List(warehouses, err)
}

func (api *WarehousesAPI) create(ctx iris.Context) {
    input, ok := ctx.BindJSON[inventory.WarehouseInput]()
    if !ok {
        return // 400 already written: bad JSON or failed Validate.
    }

    id, err := api.svc.CreateWarehouse(ctx, input)
    ctx.Created(id, err) // 201 with {"id": "..."}.
}

func (api *WarehousesAPI) get(ctx iris.Context) {
    id := ctx.Params().GetUUID("id")

    w, err := api.svc.Warehouse(ctx, id)
    ctx.OK(w, err) // ErrNotFound is mapped to 404 centrally.
}

func (api *WarehousesAPI) delete(ctx iris.Context) {
    id := ctx.Params().GetUUID("id")

    deleted, err := api.svc.DeleteWarehouse(ctx, id)
    ctx.NoContent(deleted, err) // 204, or 403 when sealed.
}

func (api *WarehousesAPI) seal(ctx iris.Context) {
    id := ctx.Params().GetUUID("id")

    sealed, err := api.svc.Seal(ctx, id)
    ctx.NoContent(sealed, err) // 204 first time, 304 after.
}
```

Guards and Nesting

Items live under a warehouse, so they get their own router mounted at `/{warehouse_id:uuid}/items`. Two middleware idioms do the protective work. A group guard (`r.Use`) runs before every item route: it loads the warehouse once, replies with the mapped 404 when it does not exist, and caches it on the request so no handler loads it twice. A positional guard sits between path and handler on the write routes only, rejecting sealed warehouses with 403. Guards are methods on the API struct, so they see the same injected service as the handlers.

Pagination returns a large collection in numbered slices (page 2, twenty items per page) instead of all at once, so responses stay small and fast no matter how much data accumulates.

The `list` handler is one `ctx.Paginated` call: it reads `?page=` and `?size=` from the query and an optional JSON filter body, invokes the collector, and renders the standard page envelope (`current_page`, `total_items`, `has_next_page`, `items`, ...).

Before the code, the shape of the whole tree, because where each guard attaches is the design:



A group guard on the items child router runs for every route below it, while the seal check is positional and sits only on the writes, which is why a sealed warehouse still answers reads.

`api/items.go`:

```go
// api/items.go
package api

import (
    stdContext "context"

    "github.com/kataras/iris/v14"
    "github.com/kataras/iris/v14/rest/errors"

    "stockroom/inventory"
)
```

```
// warehouseContextKey caches the guard-loaded warehouse on the request
// so the handlers below never load it twice.
const warehouseContextKey = "stockroom.warehouse"

// ItemsAPI serves /warehouses/{warehouse_id:uuid}/items.
type ItemsAPI struct {
    svc *inventory.Service
}

func NewItemsAPI(svc *inventory.Service) *ItemsAPI {
    return &ItemsAPI{svc: svc}
}

func (api *ItemsAPI) Routing(r iris.Router) {
    // Group guard: every item route needs an existing warehouse.
    r.Use(api.requireWarehouse)

    r.Get("/", api.list)
    // Positional guards: writes additionally need an unsealed warehouse.
    r.Post("/", api.requireUnsealed, api.create)
    r.Get("/{id:uuid}", api.get)
    r.Delete("/{id:uuid}", api.requireUnsealed, api.delete)
    r.Post("/{id:uuid}/withdraw", api.requireUnsealed, api.withdraw)
}

// requireWarehouse loads the warehouse of the path parameter, replies
// 404 when it does not exist and caches it for later handlers.
func (api *ItemsAPI) requireWarehouse(ctx iris.Context) {
    id := ctx.Params().GetUUID("warehouse_id")

    w, err := api.svc.Warehouse(ctx, id)
    if errors.HandleError(ctx, err) {
        return // mapped 404, request stopped.
    }

    ctx.Values().Set(warehouseContextKey, w)
    ctx.Next()
}

// requireUnsealed rejects writes against a sealed warehouse.
func (api *ItemsAPI) requireUnsealed(ctx iris.Context) {
    if currentWarehouse(ctx).Sealed {
        errors.PermissionDenied.Detail(ctx, "warehouse is sealed",
            "ask an administrator to unseal it first")
        return
    }
    ctx.Next()
}

func currentWarehouse(ctx iris.Context) inventory.Warehouse {
    w, _ := ctx.ValueAs[inventory.Warehouse](warehouseContextKey)
    return w
}

func (api *ItemsAPI) list(ctx iris.Context) {
    warehouse := currentWarehouse(ctx)

    ctx.Paginated(func(reqCtx stdContext.Context,
        opts iris.PageOptions,
        filter inventory.ItemFilter) ([]inventory.StockItem, int64, error) {
        return api.svc.Items(reqCtx, warehouse.ID, opts, filter)
    })
```

```
}

func (api *ItemsAPI) create(ctx iris.Context) {
    input, ok := ctx.BindJSON[inventory.ItemInput]()
    if !ok {
        return
    }

    id, err := api.svc.CreateItem(ctx, currentWarehouse(ctx).ID, input)
    ctx.Created(id, err)
}

func (api *ItemsAPI) get(ctx iris.Context) {
    id := ctx.Params().GetUUID("id")

    item, err := api.svc.Item(ctx, currentWarehouse(ctx).ID, id)
    ctx.OK(item, err)
}

func (api *ItemsAPI) withdraw(ctx iris.Context) {
    id := ctx.Params().GetUUID("id")

    input, ok := ctx.BindJSON[inventory.WithdrawInput]()
    if !ok {
        return
    }

    item, err := api.svc.Withdraw(ctx, currentWarehouse(ctx).ID, id, input)
    ctx.OK(item, err) // ErrInsufficientStock renders as 409.
}

func (api *ItemsAPI) delete(ctx iris.Context) {
    id := ctx.Params().GetUUID("id")

    deleted, err := api.svc.DeleteItem(ctx, currentWarehouse(ctx).ID, id)
    ctx.NoContent(deleted, err)
}
```

The Admin Party

HTTP Basic Authentication is the simplest login scheme the protocol offers: the client repeats a username and password in an `Authorization` header on every request, and the server verifies them before serving. The credentials are only encoded, not encrypted, so the scheme is safe strictly over HTTPS, a good fit for a small internal admin surface, the wrong tool for public user login (Chapter 10 covers the alternatives).

The admin controller knows nothing about authentication: protection is applied where it is mounted, in the builder's `API` step. It exposes the one capability the public API refuses to offer: reversing a seal.

`api/admin.go`:

```
// api/admin.go
package api
```
GO

```
import (
    "github.com/kataras/iris/v14"

    "stockroom/inventory"
)

// AdminAPI is mounted behind basic auth (see the builder's API step).
// Nothing in it knows about authentication.
type AdminAPI struct {
    svc *inventory.Service
}

func NewAdminAPI(svc *inventory.Service) *AdminAPI {
    return &AdminAPI{svc: svc}
}

func (api *AdminAPI) Routing(r iris.Router) {
    r.Get("/stats", api.stats)
    r.Post("/warehouses/{id:uuid}/unseal", api.unseal)
}

func (api *AdminAPI) stats(ctx iris.Context) {
    stats, err := api.svc.Stats(ctx)
    ctx.OK(stats, err)
}

// unseal reverses a seal: an admin-only capability, the force action
// that the public API deliberately does not expose.
func (api *AdminAPI) unseal(ctx iris.Context) {
    id := ctx.Params().GetUUID("id")

    unsealed, err := api.svc.Unseal(ctx, id)
    ctx.NoContent(unsealed, err)
}
```

Configuration and Wiring

An *environment variable* is a named string the operating system hands a process at start. It is the standard channel for per-deployment settings and secrets precisely because it lives outside the source tree: nothing to commit, nothing to leak through a repository, and every container platform knows how to set one.

One struct holds the whole configuration. It embeds `iris.BaseConfiguration`, which carries `Host`, `Port`, `AllowOrigin`, `EnableCompression`, `RequestLog` and the framework `Iris` options, so a single YAML document configures the server and the application alike (Chapter 11). Loading is one call: `iris.LoadConfiguration[Config]` reads the `STOCKROOM_CONFIG` variable when it is set (the value may be the document itself or a path to it, so a container can mount or inline it) and `config.yml` otherwise, then runs the `BindEnv` that `BaseConfiguration` provides, letting `HOST`, `PORT`, `HTTP_PLATFORM_PORT` and `ALLOW_ORIGIN` override the document. The admin password comes only from the environment; its field is `yaml:"-"` because the file is committed and

secrets never are. One difference from the hand-written loader this replaces: the document is now required, so a deployment with neither the variable nor the file fails at startup instead of running on defaults nobody chose.

`newApp` assembles everything in one builder expression (Chapter 3). The chain order is compiler-enforced. Note where each concern lands: the three sentinel errors map to their status codes once in `Errors`; the repository and the service constructor register in `Services` (the container calls `NewService(repo)` because the types match, then runs its `Init`); `Deferrables` registers the shutdown hook; the admin controller mounts behind `basicauth.Default`. `Health(true, ...)` serves `GET /health` at the root, outside the `/api` prefix. The repository arrives as a parameter of `newApp`: main passes the real store, tests pass fakes.

`main.go`:

```go
// Stockroom is a warehouse inventory service on the full recommended
// Iris stack: builder, dependency injection, controllers, central error
// map, admin party, warmup and shutdown hooks.
package main

import (
    "fmt"
    "log"
    "os"

    "github.com/kataras/iris/v14"
    "github.com/kataras/iris/v14/middleware/basicauth"
    "github.com/kataras/iris/v14/rest/errors"

    "stockroom/api"
    "stockroom/inventory"
)

// Config is the whole application configuration: the fields every
// service shares come from iris.BaseConfiguration, inlined, and the
// stockroom's own settings sit next to them in the same YAML document.
// AdminPassword is deliberately yaml:"-": secrets come from the
// environment, never from a committed file.
type Config struct {
    iris.BaseConfiguration `yaml:",inline"`

    Env           string `yaml:"Env"`
    Developer     string `yaml:"Developer"`
    AdminUser     string `yaml:"AdminUser"`
    AdminPassword string `yaml:"-"`
}

// LoadConfig reads the document from the STOCKROOM_CONFIG environment
// variable or the config.yml file, lets HOST, PORT and ALLOW_ORIGIN
// override it, then reads the secrets from the environment.
func LoadConfig() (Config, error) {
    cfg, err := iris.LoadConfiguration[Config](
        "STOCKROOM_CONFIG", "config.yml")
    if err != nil {
        return cfg, err
    }
```

```
    cfg.AdminPassword = os.Getenv("ADMIN_PASSWORD")
    if cfg.AdminPassword == "" {
        return cfg, fmt.Errorf("ADMIN_PASSWORD is required")
    }
    return cfg, nil
}

// newApp assembles the whole application. The repository arrives as a
// parameter: main passes the real store, tests pass fakes.
func newApp(cfg Config, repo inventory.Repository) *iris.Application {
    return iris.NewBuilder(cfg.Iris).
        Prefix("/api").
        AllowOrigin(cfg.AllowOrigin).
        Compression(cfg.EnableCompression).
        LogRequests(cfg.RequestLog).
        Health(true, cfg.Env, cfg.Developer).
        Errors(errors.NewOptions().
            MapErrors(errors.NotFound, inventory.ErrNotFound).
            MapErrors(errors.PermissionDenied, inventory.ErrSealed).
            MapErrors(errors.Aborted, inventory.ErrInsufficientStock)).
        Deferrables(func() { log.Println("stockroom: goodbye") }).
        Services(repo, inventory.NewService).
        RouterMiddlewares().
        Middlewares().
        API("/warehouses", api.NewWarehousesAPI).
        API("/admin", basicauth.Default(map[string]string{
            cfg.AdminUser: cfg.AdminPassword,
        }), api.NewAdminAPI).
        Build()
}

func main() {
    cfg, err := LoadConfig()
    if err != nil {
        log.Fatal(err)
    }

    app := newApp(cfg, inventory.NewMemStore())
    app.Listen(cfg.Addr())
}
```

The committed `config.yml`:

YAML

```
Port: 8080
AllowOrigin: none
EnableCompression: true
RequestLog: false
Env: development
Developer: stockroom
AdminUser: admin
Iris:
  Name: stockroom
  LogLevel: info
```

The mapping of `errors.Aborted` deserves a note: its canonical status is 409 Conflict, the honest answer for a withdrawal that lost the race for the last units. The full code table lives in Chapter 9.

The Test Suite

Because `newApp(cfg, repo)` takes its storage as a parameter, every test builds the real application (real router, real guards, real error map, real basic auth) around any repository it likes, and `httptest.New` drives it in memory. The final test embeds the `Repository` interface in a struct and overrides a single method to fail with a fake connection string, proving a storage outage renders as a generic 500 that leaks nothing. Note that the fake breaks `Warehouses`, not `Counts`, so the service's `Init` warmup still succeeds and the application builds.

`main_test.go`:

```go
// main_test.go
package main

import (
    stdContext "context"
    "fmt"
    "testing"
    "uuid"

    "github.com/kataras/iris/v14"
    "github.com/kataras/iris/v14/httptest"

    "stockroom/inventory"
)

func testConfig() Config {
    return Config{
        BaseConfiguration: iris.BaseConfiguration{
            Iris:        iris.DefaultOptions,
            AllowOrigin: "none",
        },
        Env:           "test",
        Developer:     "tests",
        AdminUser:     "admin",
        AdminPassword: "test-password",
    }
}

func TestWarehouseLifecycleAndSealRule(t *testing.T) {
    app := newApp(testConfig(), inventory.NewMemStore())
    e := httptest.New(t, app)

    // Create.
    id := e.POST("/api/warehouses").
        WithJSON(inventory.WarehouseInput{
            Name: "central", Region: "eu-west",
        }).
        Expect().Status(httptest.StatusCreated).
        JSON().Object().Value("id").String().NotEmpty().Raw()
```

```
    // Read back.
    e.GET("/api/warehouses/"+id).Expect().Status(httptest.StatusOK).
        JSON().Object().HasValue("name", "central")

    // Validation: ctx.BindJSON runs WarehouseInput.Validate -> 400.
    e.POST("/api/warehouses").
        WithJSON(inventory.WarehouseInput{Name: "", Region: ""}).
        Expect().Status(httptest.StatusBadRequest)

    // Seal: 204 the first time, 304 when nothing changes.
    e.POST("/api/warehouses/" + id + "/seal").
        Expect().Status(httptest.StatusNoContent)
    e.POST("/api/warehouses/" + id + "/seal").
        Expect().Status(httptest.StatusNotModified)

    // A sealed warehouse refuses deletion: ErrSealed -> 403.
    e.DELETE("/api/warehouses/" + id).
        Expect().Status(httptest.StatusForbidden)

    // The admin force action reverses the seal; then deletion works.
    e.POST("/api/admin/warehouses/"+id+"/unseal").
        WithBasicAuth("admin", "test-password").
        Expect().Status(httptest.StatusNoContent)
    e.DELETE("/api/warehouses/" + id).
        Expect().Status(httptest.StatusNoContent)

    // The central error map turns ErrNotFound into a 404.
    e.GET("/api/warehouses/" + uuid.New().String()).
        Expect().Status(httptest.StatusNotFound)
}

func TestItemsPaginationGuardsAndWithdraw(t *testing.T) {
    app := newApp(testConfig(), inventory.NewMemStore())
    e := httptest.New(t, app)

    warehouseID := e.POST("/api/warehouses").
        WithJSON(inventory.WarehouseInput{
            Name: "north", Region: "eu-north",
        }).
        Expect().Status(httptest.StatusCreated).
        JSON().Object().Value("id").String().NotEmpty().Raw()

    itemsPath := "/api/warehouses/" + warehouseID + "/items"

    var firstItemID string
    for i := 1; i <= 5; i++ {
        itemID := e.POST(itemsPath).
            WithJSON(inventory.ItemInput{
                SKU:      fmt.Sprintf("CRT-%03d", i),
                Name:     fmt.Sprintf("Crate %d", i),
                Quantity: 10 * i,
            }).
            Expect().Status(httptest.StatusCreated).
            JSON().Object().Value("id").String().NotEmpty().Raw()
        if i == 1 {
            firstItemID = itemID
        }
    }

    // Page 2 with 2 items per page: items 3 and 4 of 5.
    page := e.GET(itemsPath).
        WithQuery("page", 2).WithQuery("size", 2).
```

```
        Expect().Status(httptest.StatusOK).JSON().Object()
    page.HasValue("current_page", 2)
    page.HasValue("total_items", 5)
    page.Value("items").Array().Length().IsEqual(2)

    // The JSON filter body narrows the listing to one SKU.
    e.GET(itemsPath).
        WithJSON(inventory.ItemFilter{SKU: "CRT-003"}).
        Expect().Status(httptest.StatusOK).
        JSON().Object().HasValue("total_items", 1)

    // Withdraw 4 of 10: the response carries the new balance.
    e.POST(itemsPath+"/"+firstItemID+"/withdraw").
        WithJSON(inventory.WithdrawInput{Quantity: 4}).
        Expect().Status(httptest.StatusOK).
        JSON().Object().HasValue("quantity", 6)

    // Withdrawing more than the balance: ErrInsufficientStock -> 409.
    e.POST(itemsPath+"/"+firstItemID+"/withdraw").
        WithJSON(inventory.WithdrawInput{Quantity: 100}).
        Expect().Status(httptest.StatusConflict)

    // Seal the warehouse: writes turn 403, reads keep working.
    e.POST("/api/warehouses/" + warehouseID + "/seal").
        Expect().Status(httptest.StatusNoContent)
    e.POST(itemsPath).
        WithJSON(inventory.ItemInput{
            SKU: "CRT-999", Name: "Late crate", Quantity: 1,
        }).
        Expect().Status(httptest.StatusForbidden)
    e.GET(itemsPath).Expect().Status(httptest.StatusOK)

    // Unknown warehouse: the group guard replies 404 first.
    e.GET("/api/warehouses/" + uuid.New().String() + "/items").
        Expect().Status(httptest.StatusNotFound)
}

func TestAdminRequiresCredentials(t *testing.T) {
    app := newApp(testConfig(), inventory.NewMemStore())
    e := httptest.New(t, app)

    e.GET("/api/admin/stats").
        Expect().Status(httptest.StatusUnauthorized)

    e.GET("/api/admin/stats").
        WithBasicAuth("admin", "test-password").
        Expect().Status(httptest.StatusOK).
        JSON().Object().HasValue("warehouses", 0)
}

// failingRepo overrides one method of a working repository; everything
// else is inherited. Interface embedding makes one-method fakes
// one-liners. Counts still works, so Service.Init succeeds and the
// application builds.
type failingRepo struct {
    inventory.Repository
}

func (failingRepo) Warehouses(
    stdContext.Context) ([]inventory.Warehouse, error) {
    return nil, fmt.Errorf("pg: SSL is off on 192.168.7.4:5432")
}
```

```
// Internal error details must never reach the client.
func TestInternalErrorsDoNotLeak(t *testing.T) {
    app := newApp(testConfig(), failingRepo{inventory.NewMemStore()})
    e := httptest.New(t, app)

    body := e.GET("/api/warehouses").
        Expect().Status(httptest.StatusInternalServerError).
        JSON().Object()
    body.HasValue("message", "server error")
    body.NotContainsValue("192.168.7.4") // stays server-side.
}
```

Run It

```sh
go mod tidy && go test ./...   # ok  stockroom
ADMIN_PASSWORD=change-me go run .
```

Walk the rules with curl (keep the created id in `$W`):

```sh
curl -X POST localhost:8080/api/warehouses \
  -H 'Content-Type: application/json' \
  -d '{"name":"central","region":"eu-west"}'


curl -X POST localhost:8080/api/warehouses/$W/items \
  -H 'Content-Type: application/json' \
  -d '{"sku":"CRT-001","name":"Crates","quantity":40}'


curl "localhost:8080/api/warehouses/$W/items?page=1&size=10"


curl -X POST localhost:8080/api/warehouses/$W/seal
curl -X DELETE localhost:8080/api/warehouses/$W


curl -X POST localhost:8080/api/admin/warehouses/$W/unseal
curl -u admin:change-me -X POST \
  localhost:8080/api/admin/warehouses/$W/unseal


curl localhost:8080/health
```

Choosing Between the Two Shapes

Shortl and Stockroom are not two quality levels; they are two honest answers to differently sized problems. `iris.New()` with hand-registered routes is the right call when the whole program fits in a couple of files: there is no indirection to justify, and the response helpers and typed macros

work identically in both shapes.

Reach for `iris.NewBuilder` when any of these appear: more than one group of endpoints, a dependency you will want to fake in tests, errors that repeat across handlers, configuration that differs between environments, or work that must happen at startup or shutdown. Each of those concerns has exactly one home in the builder chain, which is the point: Stockroom's tests could replace the entire storage layer in one line only because the wiring lives in `newApp`, not scattered across handlers.

Growing from the first shape to the second is mechanical, not a rewrite: extract the domain into its own package, put the storage behind an interface, turn route groups into controllers, and move the wiring into the builder. The chapters this one leans on cover each move in depth: routing and controllers in Chapter 4, dependency injection in Chapter 8, the error pipeline in Chapter 9, configuration in Chapter 11 and the test techniques in Chapter 16.

Summary

- Both applications in this chapter are complete: paste the labeled blocks into files, run `go mod tidy`, and `go test ./...` passes.
- Shortl shows the lightweight path: `iris.New()` with a partial options literal, a typed `{code:alphabetical}` catch-all that filters junk requests before the handler, `ctx.BindJSON` with a custom `validation.Func`, a mutex-guarded store, and a redirect test that inspects the 301 instead of following it.
- Stockroom shows the production shape: env-first configuration with secrets outside YAML, three sentinel errors mapped once, controllers with constructors and `Routing`, a nested router protected by a group guard plus positional guards, `ctx.Paginated` listings, an admin party behind basic auth exposing the force action, `iris.Initializer` warmup inside `Build()` and `Deferrables` at shutdown.
- The architecture is what makes the tests cheap: because storage is an interface handed to `newApp`, an interface-embedding fake proved in five lines that internal failures never leak to clients.

Further Reading

- MDN: Redirections in HTTP: how 3xx status codes and the `Location` header drive Shortl's 301.

- MDN: HTTP authentication: the Basic scheme that protects Stockroom's admin party.
- Go: sync.RWMutex: the read-write lock guarding Shortl's in-memory store.
- Go: the errors package: sentinel error values and `errors.Is`, the standard mechanics an error map builds on.
- Go: Add a test: the official primer on `go test` and the `_test.go` convention both suites rely on.
- uuid: the standard library's UUID package (Go 1.27) behind Stockroom's `{id:uuid}` identifiers.

CHAPTER 23

Upgrading to v14

This book documents Iris v14 (`github.com/kataras/iris/v14`) exclusively. Every chapter describes the current API with no comparisons to earlier releases, so there is nothing in these pages to translate or un-learn. This chapter exists for one audience: readers bringing an existing v12 codebase forward. It tells you where the migration instructions live and how to approach the upgrade. It deliberately does not duplicate the instructions themselves.

Background

In semantic versioning (the `MAJOR.MINOR.PATCH` numbering Go modules follow), a **major version upgrade** is the one release type allowed to break existing code: names may be removed, defaults may change, and your program may need edits before it compiles again. Breaking changes are not carelessness; they are how a framework sheds accumulated mistakes and unsafe defaults without carrying them forever: v14's secure-by-default behavior exists precisely because a major release was the one moment it could change. Go makes the break explicit by putting the major version in the import path (`/v12`, `/v14`), so the two versions cannot be confused inside one build.

The Canonical Migration Guide

The step-by-step migration guide ships with the framework repository and is maintained alongside the code:

https://github.com/kataras/iris/blob/main/MIGRATION.md

Because it lives next to the source, it (not this book) is the authority on what changed between v12 and v14. Read it end to end before touching your code. The guide covers these categories, each with the exact replacement and a code sample:

Category	What it covers
Import path	The module major version moves from `/v12` to `/v14`; a one-shot `sed` rewrite for `go.mod` and every source file.
Package relocations	`sessions`, `cache`, `websocket`, `i18n`, `view` and `versioning` move under `middleware/`; `middleware/recover` becomes `middleware/recovery`, `mvc` becomes `controller/mvc`, `macro` becomes `core/router/macro`, `x/client` becomes `client`, `x/jsonx` becomes `rest/types`, `x/errors` becomes `rest/errors`. `middleware/logger`, `hero` and `x/sqlx` are removed outright. Four moved packages also changed their entry point (`cache`, `recovery`, `websocket`, `view`).
Removed v12 APIs	Each removed name with its single v14 replacement: `Party` → `Router`, `PartyConfigure` / `Configure` → `Router` / `Routing`, `Done` / `DoneGlobal` / `UseGlobal` → the `Use` family, `OnErrorCode` → `HandleNoRoute` / `HandleNoMethod` plus `rest/errors`, `iris.Configuration` → `iris.Options`, `RegisterView` / `ctx.View` → `middleware/view`, `iris.Addr` / `iris.TLS` / `iris.AutoTLS` / `Runner` → `app.Run(iris.ServerOptions{...})`, `SetContextErrorHandler` → the `ContextOptions.ErrorHandler` option, `HandleDir` / `Favicon` → `controller/fileserver`, `WWW` / `SubdomainRedirect` → `middleware/rewrite`, `app.Validator` → `ContextOptions.RequestValidator`, `hero` → the built-in DI container, `mvc.GRPC` → `controller/mvc.GRPC`, `Dependencies()` → `DI()`.
Context method changes	The largest mechanical part: all eighteen `URLParam*` methods become `ctx.Query().Get*`; `ctx.Params().Get` becomes `GetString`; request and response accessors split by name, so `ctx.GetHeader(k)` becomes `ctx.Header(k)` and the v12 two-argument `ctx.Header(k, v)` setter becomes `ctx.SetHeader(k, v)` (with `ResponseHeader` new for reading back what a handler set); the same split applies to content type. Plus the rename table (`GetCookie` → `CookieValue`, `GetBody` → `Body`, ...), the removed methods (`CallFunc`, `ReflectValue`, ...) and the new `SetLogoutFunc` / `Logout` signatures.
Corrected v12 behavior	Six v12 bugs whose fixes change runtime behavior without needing a code change: `ReadBody` with a `*string` / `*[]byte` target no longer re-reads a drained body, `KitchenTime.Scan` no longer drops the minutes of a database `TIME` value, a `UserMap` with capitalized keys resolves, `ParseContentType` keeps the case of parameter values such as a multipart `boundary`, `UniqueHandlers` removes every occurrence and leaves its input alone, and `Params().GetTrimHTML` / `GetTrimJSON` / `GetTrimXML` remove the extension only (`math.html` no longer comes back as `ma`).
Secure-by-default changes	Session cookies ship `Secure` with `SameSite=Lax` (and the `sessions.Database` interface is redesigned around generics); JWT verification reads the `Authorization` header only, dropping the v12 `?token=` extractor; CORS no longer advertises credentials alongside a wildcard origin; `AddCookie` defaults to `SameSite=Lax`.
Opt-in hardening	New v14 capabilities that are off by default: `ContextOptions.BodyMaxMemory` request-body caps, `session.RegenerateID` after sign-in, `fileserver.BlockHidden` for static servers, `requestid.ServerOnlyGenerator`, a raised `password.Cost`.

One path-parameter change is easy to miss because most code keeps compiling. In v12 a `{id:uuid}` parameter stored the text it validated; in v14 it stores a parsed `uuid.UUID`. Read it with `ctx.Params().GetUUID("id")` or `ctx.BindParam[uuid.UUID]("id")`.

`ctx.Params().GetString("id")` still works and returns the canonical lowercase form, while `ctx.BindParam[string]("id")` on such a route now answers 400.

Tips for a Smooth Upgrade

- **Read the guide first, code second.** Skim all of MIGRATION.md before editing anything, and note which categories your application actually touches. Most codebases hit the import path, one or two removed APIs and one behavior change. Knowing which ones up front turns the upgrade from exploration into execution.
- **Upgrade in one commit on a dedicated branch.** The import-path change touches every Go file, so there is no useful halfway state. Do the mechanical rewrite, apply the API replacements, and land the whole move as a single reviewable change.
- **Let the compiler drive.** The removed v12 names fail loudly: the renames were designed so that every stale call site becomes a compile error rather than code that silently misbehaves. Work through the error list with the guide's replacement table beside you; when the build is green, the mechanical part of the migration is done.
- **Run your test suite before shipping.** The secure-by-default changes (session cookie attributes, JWT extraction, CORS credentials) and the corrected v12 bugs compile without complaint but behave differently at runtime. Your integration tests are where those differences surface; see Chapter 16 if you need to add coverage first.

Starting Fresh

If you are not carrying v12 code, skip the migration guide entirely. New projects should start from Chapter 1 or the framework's TUTORIAL.md, both of which build directly on the recommended v14 stack: `iris.NewBuilder`, dependency-injected services, controllers with `Routing(r iris.Router)` methods and the response helpers. There is no v12 heritage in that path and nothing to migrate away from.

Summary

- This book covers v14 only; upgrades from v12 are documented in the maintained guide at https://github.com/kataras/iris/blob/main/MIGRATION.md.

- The guide covers the import path move, package relocations, removed v12 APIs with their replacements, the `Context` method changes, corrected v12 behavior, secure-by-default changes and recommended opt-in hardening.
- Upgrade deliberately: read the guide first, land the change in one commit, let compile errors guide the API replacements, and rely on your test suite to catch the behavior changes that compile cleanly.
- Fresh projects have nothing to migrate. Start at Chapter 1.

Further Reading

- MIGRATION.md: the maintained, authoritative v12-to-v14 migration guide this chapter points at.
- Semantic Versioning: the `MAJOR.MINOR.PATCH` contract and what each part promises about compatibility.
- Go Modules: Developing a major version update: why the major version appears in the import path and how `/v14` releases work.

CHAPTER 24

Advanced Topics

Iris has no plugin system, and it does not need one: its extension model is plain Go. Every advanced capability in this chapter is an ordinary, verifiable API: generic controllers that generate REST routes from your types, typed contexts that replace `iris.Context` in your handler signatures, global patches that swap framework internals such as the JSON encoder, custom path parameter types, raw router wrappers, a multi-application registry, and interop with `net/http`. This final chapter walks through each extension point and closes with the built-in controllers that ship with the framework.

Background

What an extension point is

An extension point is a place where a framework lets you replace or extend its own machinery rather than merely use it: swap the JSON encoder behind `ctx.JSON`, teach the router a new path parameter type, run code before routing even begins. Reaching for one is justified in two situations: you keep repeating the same workaround in many handlers because a framework default does not fit your case, or you are integrating tooling that predates or lives outside Iris. Be clear about the flip side: most applications never need this chapter. The first twenty-two chapters cover everyday application work; if you are reading the book front to back, it is enough to remember what exists here and return when a concrete need appears.

What generics buy the MVC controllers

Go generics (*type parameters*, added to the language in Go 1.18) let one piece of code be written against placeholder types and specialized by the compiler for the concrete types you supply. The first section uses them as a code generator: from five typed method signatures the framework derives routes, request decoding, response encoding and error handling. The alternative in older frameworks is reflection (inspecting types while the program runs), which

costs performance and turns a wrong method shape into a runtime surprise. With generics the same wiring is checked when you compile: a controller whose signatures do not match the contract fails the build, not a production request.

What process-global means

A *process* is one running instance of your compiled program, the unit the operating system starts and stops. State that is process-global is shared by everything inside that binary: every Iris application in it, every library it imports, every test in the same `go test` run. Two features below are process-global (`iris.Patches()` and the application registry behind the `apps` package), and such state demands caution for the reason globals always do: whoever writes last wins, and every reader everywhere is affected. The rules that keep it safe appear in their sections; the short version is mutate once, at startup, from `main`, never from a library.

Generic MVC Controllers

The routing controllers of Chapter 4 give you full control: a `Routing(r iris.Router)` method and any routes you like. The `controller/mvc` package sits one level higher. You implement five typed methods on a struct and the framework generates the REST routes, decodes requests, encodes responses, and wires the error pipeline of Chapter 9, all without reflection at serve time.

The Controller Interface

```go
// github.com/kataras/iris/v14/controller/mvc
type Controller[Context handler.ContextType, Request any, Response any,
    ID iris.IDType, Filter any, Total iris.CountType] interface {
    Create(Context, Request) (Response, error)
    Get(Context, ID) (Response, error)
    Update(Context, ID, Request, ...string) (*Response, bool, error)
    Delete(Context, ID) (bool, error)
    List(ctx Context, filter Filter, limit, offset int) (
        []Response, Total, error)
}
```

The six type parameters, honestly:

Parameter	Constraint	Meaning
`Context`	`handler.ContextType`	`iris.Context` or a standard `context.Context`; your methods may accept either
`Request`	`any`	The input payload, decoded from the request
`Response`	`any`	The output model; also names the generated path parameter

`ID`	`iris.IDType`	`string | int | int64 | uint64 | uuid.UUID`
`Filter`	`any`	The `List` filter payload
`Total`	`iris.CountType`	Any integer or float type for the list total

At the call site you write only the controller type; the compiler infers the rest from your method signatures. `iris.CountType` is defined in `core/pkg/pagination` as `constraints.Number`, and it and the other numeric bounds the framework's generics use come from `core/pkg/constraints`, a six-interface package that mirrors the subset of `golang.org/x/exp/constraints` Iris needed, so the module carries no dependency on the experimental repository.

A Complete Controller

```go
package main

import (
    "github.com/kataras/iris/v14"
    "github.com/kataras/iris/v14/controller/mvc"
)

type Store struct {
    ID       string `json:"id"`
    Name     string `json:"name"`
    Location string `json:"location"`
}

type StoreFilter struct {
    Location string `json:"location"`
}

type StoreService struct{ /* database handle */ }

func NewStoreService() *StoreService { return &StoreService{} }

type StoreController struct {
    Service *StoreService // injected by the DI container.
}

func (c *StoreController) Create(ctx iris.Context, in Store) (Store, error) {
    in.ID = "generated"
    return in, nil
}

func (c *StoreController) Get(ctx iris.Context, id string) (Store, error) {
    return Store{ID: id, Name: "Bookstore", Location: "New York"}, nil
}

func (c *StoreController) Update(ctx iris.Context, id string, in Store,
    columns ...string) (*Store, bool, error) {
    return &in, true, nil
}

func (c *StoreController) Delete(ctx iris.Context, id string) (bool, error) {
    return true, nil
}
```

```
func (c *StoreController) List(ctx iris.Context, filter StoreFilter,
    limit, offset int) ([]Store, int, error) {
    stores := []Store{{ID: "1", Name: "Bookstore", Location: "New York"}}
    return stores, len(stores), nil
}

func main() {
    app := iris.New()
    app.RegisterDependency(NewStoreService)
    app.Router("/stores", mvc.UseController[*StoreController]())
    app.Listen(":8080")
}
```

`UseController` resolves the controller through the dependency injection container of Chapter 8, or constructs it and binds its exported fields, then registers these routes:

Method, path	Controller method	Success response
`POST /stores`	`Create`	201, JSON body
`GET /stores/{store_id}`	`Get`	200, JSON body
`PUT /stores/{store_id}?columns=a&columns=b`	`Update`	200 if `*Response` non-nil, else 204 (true) or 304 (false)
`DELETE /stores/{store_id}`	`Delete`	204 (true) or 304 (false)
`POST /stores/list`, `QUERY /stores/list`	`List`	200, paginated JSON

The path parameter name derives from the response model (`Store` becomes `store_id`); its macro type derives from the `ID` type parameter: `int`, `int64` and `uint64` map to the matching macro, `uuid.UUID` maps to `{...:uuid}`, and a `string` ID stays a plain `{...:string}`. Declare the controller with `uuid.UUID` when you want the uuid macro's validation. (`iris.IDType` permits only those five, so there is no `uint` case to reach.) Errors returned by any method flow through the `rest/errors` pipeline automatically: the controller installs `errors.Handler` on its child router for you.



The two type parameters of a controller decide one shared path parameter, and each of the five interface methods becomes a route wrapped in the decode, call and respond pipeline.

Resource Options

`UseController` takes a variadic `...mvc.Option`, and the five options below cover what a generated resource usually needs to say about itself. Existing `UseController()` calls compile unchanged.

```go
app.Router("/stores", mvc.UseController[*StoreController](
    mvc.Guard(auth.RequireAdmin),
    mvc.Without(mvc.Delete),
    mvc.IDParam("id"),
    mvc.Pagination(iris.AllowSort("name", "created_at")),
))
```

Option	What it does
`mvc.Guard(handlers...)`	Registers middleware on the write routes only: `Create`, `Update`, `Delete`
`mvc.Without(verbs...)`	Drops generated routes. The verbs are `mvc.Create`, `mvc.Get`, `mvc.Update`, `mvc.Delete` and `mvc.List`
`mvc.IDParam(name)`	Overrides the derived parameter name; the macro still comes from the ID type
`mvc.BeforeWrite(fn)`	Runs on `Create` and `Update`, after the body is decoded and before the controller sees it
`mvc.Pagination(opts...)`	The list route's sort policy: `iris.AllowSort`, `iris.DenySort`, or `errors.<Code>.DenySort` to answer with a chosen code

`Guard` exists because "everyone may read, only a signed-in caller may write" is the common shape and a router-level `Use` cannot express it without splitting the resource into two routers. A dropped route under `Without` is not registered at all, so it answers 404, or 405 when the application enables `FireMethodNotAllowed` and another method of the same path survives.

`BeforeWrite` is how a value the client must not choose gets stamped on every body in one place, which is the other half of the request scope from Chapter 10:

```go
mvc.BeforeWrite(func(ctx iris.Context, v *Store) error {
    tenantID, ok := auth.ScopeID[string](ctx)
    if !ok {
        return errors.Unauthenticated
    }

    v.TenantID = tenantID
    return nil
})
```

The model type is a type parameter, so the hook receives `*T` directly and no reflection happens at serve time. Three of the options fail loudly rather than quietly: a `BeforeWrite` declared for a type the resource does not write panics when the routes are registered, naming both types, because what it was silently skipping is typically "stamp the tenant id on this body" and the symptom would otherwise be a cross-tenant write. `Without` panics on an unknown verb and `IDParam` on an empty name, both at option-construction time.

Routes Straight Over a Repository

When a resource's HTTP half adds nothing to its store, there is no controller type left to write. `mvc.UseRepository` registers the same six routes directly over the repository:

```go
app.Router("/invoices", mvc.UseRepository(invoiceRepository))
```

The repository is the five methods a service already has, over the *standard* context: a repository never sees Iris.

```go
type Repository[T any, ID iris.IDType, Filter any] interface {
    Create(ctx context.Context, v T) (ID, error)
    Get(ctx context.Context, id ID) (T, error)
    Update(ctx context.Context, id ID, v T, onlyColumns ...string) (
        updated bool, err error)
    Delete(ctx context.Context, id ID) (deleted bool, err error)
    List(ctx context.Context, filter Filter, limit, offset int) (
        []T, int64, error)
}
```

`POST /` renders 201 `{"id": ...}`, `GET /{model_id}` the model, `PUT` and `PATCH /{model_id}` a 204 or a 304 through `ctx.NoContent`, `DELETE /{model_id}` the same, and `POST` or `QUERY /list` the page the `List` route already renders. The parameter name and its macro derive from the model and its ID type exactly as they do for `UseController`, so a `uuid.UUID` id registers `{invoice_id:uuid}` and validates before any handler runs, and every option from the table above applies here too.

One deliberate difference: the update routes read their body and columns through `ctx.BindUpdate` (Chapter 5), so a column that is not a member of the model answers 400 naming it and the members it could have been. The controller routes cannot do that, because their request type is decoded through a pluggable decoder. The names that reach the repository are still the client's, so a repository that builds SQL from them maps them to its own columns and never interpolates them.

Reach for a controller as soon as a route needs anything the repository does not do: a mail on create, a computed response, a second store to keep in sync.

Custom Routes and the Generated Parameter

A controller may additionally implement `Routing(r iris.Router)` to add routes beside the generated ones. `mvc.GetPathParam` returns the generated parameter's name, macro type and route path:

```go
func (c *StoreController) Routing(r iris.Router) {
    param := mvc.GetPathParam(c) // {Name: "store_id", Type: "uuid", ...}
    r.Get("/featured", func(ctx iris.Context) {
        ctx.Writef("featured stores (id param: %s)", param.Name)
    })
}
```

`Routing` runs before the five generated routes are registered, so an `r.Use` here guards those too:

```go
func (c *StoreController) Routing(r iris.Router) {
    r.Use(auth.RequireAdmin) // protects Create, Get, Update, Delete, List
}
```

The ordering is what makes that work. A route captures the router's middleware at the moment it is created, so middleware added after a route exists never reaches it.

One Controller, Every Request

A controller is a singleton. It is resolved once, its exported fields are bound once, and its five methods are captured into handlers at registration. There is no per-request instance, so every field is shared across all concurrent requests:

```go
type StoreController struct {
    Service *StoreService // fine: written once, read everywhere
    current *Store        // wrong: one request's value, visible to all
}
```

Keep fields for dependencies. Anything that belongs to a single request belongs on the context or in a method parameter.

Nested Controllers

`UseController` returns a `*DeployedController`. Its `With(relPathWithoutParam, controllers...)` method mounts children under the parent's id segment:

```go
app.Router("/stores", mvc.UseController[*StoreController]().
    With("/books", mvc.UseController[*BookController]()))
// BookController serves under /stores/{store_id}/books
```

A child reads the parent's path parameter through an auto-injected `mvc.PathParam[T, V]` field, where `T` is the parent response type and `V` its ID type:

```go
type BookController struct {
    Service    *BookService
    StoreParam *mvc.PathParam[Store, string]
}

func (c *BookController) Create(ctx iris.Context, book Book) (Book, error) {
    book.StoreID = c.StoreParam.Get(ctx) // or GetString(ctx)
    return book, nil
}
```

`DeployedController.WithViews` adds HTML endpoints beside the JSON ones for controllers that also implement the view methods (`IndexView`, `CreateView`, `DetailsView`, `UpdateView`, `DeleteView`, `ListView`) against a registered view engine. It is a generic method: the four type arguments name the data types those methods return, and a controller that does not implement `mvc.ViewController` for them panics at registration time rather than at first request:

```go
app.Router("/books", mvc.UseController[*BookController]().
    WithViews[IndexData, CreateData, ModelData, ListData]())
```

The generated routes are `GET /`, `/create`, `/{id}/details`, `/{id}/update`, `/{id}/delete` and `/list`, and the template names derive from the controller's model name plus `mvc.DefaultTemplateExtension` (`books/index.html` and so on).

The handler Subpackage

The machinery behind `UseController` is public in `controller/mvc/handler` and usable route-by-route. `handler.Handler` wraps a typed function into an `iris.Handler` with the same decode/encode/error pipeline; options tune it:

```go
import "github.com/kataras/iris/v14/controller/mvc/handler"

// POST /orders: decode Order from the body, respond 201.
app.Post("/orders", handler.Handler(createOrder,
    handler.WithStatusCode(iris.StatusCreated)))

// GET /orders/{order_id:int64}: decode the named path parameter
// into the function's second argument.
app.Get("/orders/{order_id:int64}", handler.Handler(getOrder,
    handler.WithPathParameter[int64]("order_id")))

func createOrder(ctx iris.Context, in Order) (Order, error) { /* ... */ }
func getOrder(ctx iris.Context, id int64) (Order, error)    { /* ... */ }
```

For single-value types (`string`, `int`, `int64`, `uint64`, `uuid.UUID`) the default decoder already reads the last path parameter; `WithPathParameter[T](name)` makes the binding explicit by name. `handler.IndexHandler` wraps a `func(Context) (Response, error)` action with no request payload.

WebSocket and gRPC Controllers

Both protocols in this section depart from HTTP's request-response rhythm. A *WebSocket* is a long-lived, two-way connection: after an initial HTTP handshake, client and server can each push messages at any moment, the transport behind chats, live dashboards and games. *gRPC* is a remote procedure call protocol: instead of paths and JSON, a client calls typed methods defined in a shared `.proto` schema, with binary messages carried over HTTP/2, a common choice for backend-to-backend APIs.

Two more controller forms ship in the same package. `mvc.Websocket` registers WebSocket controllers backed by the third-party `github.com/kataras/neffos` library; struct pointers or constructor functions, resolved through DI:

```go
app.Router("/websocket", mvc.Websocket(newChatController))
```

A WebSocket controller may implement `Namespace() string` and the standard neffos events (`OnNamespaceConnect`, `OnNamespaceConnected`, `OnNamespaceDisconnect`, `OnRoomJoin`, `OnRoomJoined`, `OnRoomLeave`, `OnRoomLeft`); any other exported method with the `func(*neffos.NSConn, neffos.Message) error` shape becomes a custom event.

`mvc.GRPC` serves a gRPC server (third-party `google.golang.org/grpc`) through the Iris router, POST-only, at `/{ServiceName}/{Method}` paths:

```go
grpcServer := grpc.NewServer()
pb.RegisterGreeterServer(grpcServer, ctrl)

app.Router("/", mvc.GRPC(ctrl).
    Server(grpcServer).
    ServiceName("helloworld.Greeter").
    Strict(false))
```

With `Strict(true)` only gRPC clients are served and plain HTTP requests receive 404; set it to true for streaming.

When Generic Controllers Pay Off

Reach for `mvc.UseController` when you have several resources that genuinely fit the Create/Get/Update/Delete/List shape and you want identical REST semantics, error handling and pagination across all of them: the five signatures are a contract, and uniformity is the reward. When a resource is not CRUD-shaped, or its routes need individual middleware and status conventions, a plain routing controller from Chapter 4 is the better tool. The two styles compose freely: a generic controller's `Routing` method and `With` children are ordinary routers.

Custom Typed Contexts

Handlers that keep reading the same values out of `ctx.Values()` and casting them accumulate noise. `iris.NewContextWrapper` lets you define your own context type (embedding `iris.Context`) and write handlers against it directly:

```go
package main
```

```
import (
    "github.com/kataras/iris/v14"
)

// appContext extends iris.Context with application-specific helpers.
type appContext struct {
    iris.Context
}

// SetContext makes *appContext usable by the wrapper's default pool.
func (c *appContext) SetContext(ctx iris.Context) {
    c.Context = ctx
}

func (c *appContext) TenantID() string {
    return c.Values().GetString("tenant_id")
}

func main() {
    app := iris.New()

    // The pointer type must implement SetContext(iris.Context).
    w := iris.NewContextWrapper[appContext]()

    app.Get("/dashboard", w.Handler(func(ctx *appContext) {
        ctx.JSON(iris.Map{"tenant": ctx.TenantID()})
    }))

    app.Listen(":8080")
}
```

`w.Handler` converts a `func(*appContext)` into a standard `iris.Handler`: per request it acquires an `*appContext`, calls `SetContext` with the live Iris context, runs your function, and releases the value. Because the custom type embeds `iris.Context`, every context method remains available, and you can override methods or add helpers as needed.

The wrapper offers more converters for the places Iris expects specific function shapes: `Handlers` (a slice at once), `HandlerReturnError`, `HandlerReturnDuration` and `Filter`.

A `sync.Pool`, from Go's standard library, is a recycling bin for short-lived objects: instead of allocating a fresh value for every request and leaving it to the garbage collector, code takes an instance from the pool and puts it back when done, reducing allocation pressure under load.

The default pool allocates a zero value per request. To recycle instances, build the wrapper from the `context` subpackage with a `sync.Pool`-backed pool:

GO

```
import "github.com/kataras/iris/v14/context"

w := context.NewContextWrapperWithPool(
    context.NewContextPool[appContext]())
```

With a recycling pool, `SetContext` is your reset hook: it runs on every acquire, so reset any custom fields there: instances are reused across requests.

Global Patches

`iris.Patches()` returns a process-wide singleton for overriding framework defaults. These are the patch points:

Patch	Effect
`Patches().Context().Writers().JSON(fn)`	Replaces the writer behind `ctx.JSON`
`Patches().Context().Writers().JSONP/XML/Markdown/YAML(fn)`	Same for the other body writers
`Patches().Context().GetDomain(fn)`	Changes how `Context.Domain()` derives the domain from host:port
`Patches().Context().SetDefaultCookieExpiration(d)`	Changes the lifetime `Context.AddCookie` gives a cookie with no `CookieExpires` option (365 days by default)
`Patches().Context().ResolveHTTPFS(fn)`	Changes filesystem resolution for the file server
`Patches().Context().ResolveFS(fn)`	Changes filesystem resolution for the view engine
`Patches().Context().SetMaxPageSize(n)`	Sets `pagination.MaxPageSize`, the upper bound of `PageOptions.GetLimit`
`Patches().Context().SetDefaultPageSize(n)`	Sets `pagination.DefaultPageSize`, used when a list request sends no size

The classic use of the JSON patch point was swapping in a faster third-party encoder (jsoniter, easyjson). That trade no longer exists: since Go 1.27 `ctx.JSON` streams through the standard library's `encoding/json/v2`, redesigned around the same streaming approach those libraries were built for, so the stdlib encoder is already the fast path. What the seam is still for is policy: a binary that must pretty-print every response, or enforce one marshaling option set everywhere, patches once instead of threading options through every call:

```
    // Process-global: applies to every ctx.JSON call
    // of every Iris application in this binary.
    iris.Patches().Context().Writers().JSON(
        func(ctx iris.Context, v any, options *iris.JSON) error {
            return json.MarshalWrite(ctx.ResponseWriter(), v,
                jsontext.WithIndent("  ")) // house style: always indented.
        })

    app := iris.New()
    app.Get("/", func(ctx iris.Context) {
        ctx.JSON(iris.Map{"encoder": "stdlib, patched"})
    })
    app.Listen(":8080")
}
```

(For a per-call version of the same control, pass options in the `iris.JSON` struct's `MarshalOptions` field instead; see Chapter 6. A type that implements `encoding/json/v2`'s `json.MarshalerTo` interface streams itself through plain `ctx.JSON` with no patch at all, which replaces the old `ctx.EasyJSON` special case.)

Two caveats keep patches honest. First, they are process-global: a patch affects every application in the binary, including test servers, so libraries must never patch; only the `main` package should. Second, apply them once, at startup, before serving; patches are package-level variables underneath, not synchronized for concurrent replacement.

Custom Path Macros

Chapter 4 covers the built-in path parameter types (`{id:int}`, `{id:uuid}`, `{p:path}` and friends). `app.Macros()` exposes the registry behind them, so you can extend an existing type with a new function or register an entirely new parameter type.

Unlike a patch, the registry belongs to one application. Each application gets a copy of the package-level `macro.Defaults` when it is created, and what you register through `app.Macros()` stays in it: a second application in the same process, a test server included, never sees it. Register before the routes that use it, because a route reads its macros at the moment it is registered. A parameter type the application does not know panics there. A function it does not know is skipped silently, and the route then accepts what the function would have refused.

Add a function to an existing type with `Get` + `RegisterFunc`. The outer function receives the compile-time arguments from the route pattern; the returned function validates each request's value:

```
app.Macros().Get("string").RegisterFunc("equals",
    func(expected string) func(string) bool {
        return func(paramValue string) bool {
```

GO

```
            return paramValue == expected
        }
    })

// Usage:
app.Get("/status/{state:string equals(active)}", handler)
```

Register a new parameter type with `Register`. The evaluator converts the raw segment into a typed value, or reports failure:

```go
import "time"

app.Macros().Register("datetime", "", time.Time{}, false, false,
    func(paramValue string) (any, bool) {
        t, err := time.Parse("2006-01-02", paramValue)
        if err != nil {
            return nil, false
        }
        return t, true
    })

app.Get("/events/{when:datetime}", func(ctx iris.Context) {
    when := ctx.Params().GetTime("when") // the stored time.Time, as is
    ctx.JSON(iris.Map{"weekday": when.Weekday().String()})
})
```

The `Register` arguments after the alias are the sample value (its type becomes the parameter's Go type), whether the type is the master default, and whether it is trailing (consumes the rest of the path, like `path`).

When a typed parameter fails to evaluate, builder-built applications respond with 400 `INVALID_ARGUMENT` ("invalid path parameter") by default. That handler lives in the builder's own application: an `iris.New()` application in the same process keeps answering 404, or trying the next route, until it sets one itself. `SetErrorHandler` (on the whole registry or a single macro) replaces that behavior, for the routes registered after the call:

```go
import "github.com/kataras/iris/v14/rest/errors"

app.Macros().SetErrorHandler(
    func(ctx iris.Context, paramIndex int, err error) {
        errors.InvalidArgument.Detailf(ctx, "bad path parameter",
            "parameter %d: %v", paramIndex, err)
    })
```

Wrapping the Router

`net/http` is the HTTP server in Go's standard library, the foundation every Go web framework, Iris included, builds on. Its two base types are `http.ResponseWriter`, which writes the response, and `*http.Request`, which describes the request; a router wrapper works directly at that level, before any `iris.Context` exists.

Middleware runs after Iris has acquired a context and, for `Use`, matched a route. Router wrappers run earlier: raw `net/http` code executed before routing, for every request. The wrapper signature is:

```go
func(w http.ResponseWriter, r *http.Request, next http.HandlerFunc)
```

Call `next` to continue into the router (or the next wrapper); skip it to short-circuit. Because the wrapper sees the request before any matching happens, it is the right place for cross-cutting URL rewrites:

```go
package main

import (
    "net/http"
    "strings"

    "github.com/kataras/iris/v14"
)

func main() {
    app := iris.New()

    // Serve the old /legacy/* URLs from the /v2 routes,
    // without redirects and without touching the routes themselves.
    app.WrapRouter(func(w http.ResponseWriter, r *http.Request,
        next http.HandlerFunc) {
        if p, ok := strings.CutPrefix(r.URL.Path, "/legacy/"); ok {
            r.URL.Path = "/v2/" + p
        }
        next(w, r)
    })

    app.Get("/v2/users", func(ctx iris.Context) {
        ctx.JSON(iris.Map{"version": 2})
    })

    app.Listen(":8080")
}
```

Three registration methods exist on the root router, differing only in ordering:

- `WrapRouter(fn)`: wraps immediately; with multiple calls the last registered wrapper runs first (each new wrapper wraps the previous chain).
- `AddRouterWrapper(fn)`: defers wrapping until the router builds; the first registered runs first. Use it for redirect-style wrappers that must observe the request before others rewrite it.

- `PrependRouterWrapper(fn)`: like `AddRouterWrapper` but always first; reserved for path-normalization concerns.

Choose `UseRouter` (Iris middleware, has a `Context`, still before route matching) when you need context features; choose a router wrapper when you need to mutate the raw request or integrate `net/http`-level tooling.

At the extreme end, `app.Downgrade(handlerFunc)` replaces the entire Iris router with your own `http.HandlerFunc`. The application still builds, hosts and shuts down normally, but routing (and every wrapper) is bypassed; you are responsible for everything.

Multi-Application Composition

Every `iris.New` call registers the application in a process-wide registry. Give applications names through their options and any package can retrieve them without import cycles:

```go
app := iris.New(iris.Options{Name: "api"})

// Elsewhere, without importing the package that built it:
import "github.com/kataras/iris/v14/apps"

api := apps.Get("api") // nil if no app has that exact name.
all := apps.GetAll()   // every registered *iris.Application.
apps.OnApplicationRegistered(func(app *iris.Application) {
    // fires for each future registration.
})
```

The registry itself lives in the `context` package: `context.GetApplications()` returns the same set as read-only `context.Application` interface values; that is what framework internals use, while `apps.GetAll` casts to the concrete type for you.

The Switch Application

`apps.Switch` composes several applications behind one listener. It returns a new, ordinary application whose job is to route each request to the first matching case:

```go
package main

import (
    "github.com/kataras/iris/v14"
    "github.com/kataras/iris/v14/apps"
)

func newWebsite() *iris.Application {
    app := iris.New(iris.Options{Name: "website"})
    app.Get("/", func(ctx iris.Context) { ctx.Writef("website") })
```

```
    return app
}

func newAPI() *iris.Application {
    app := iris.New(iris.Options{Name: "api"})
    app.Get("/status", func(ctx iris.Context) {
        ctx.JSON(iris.Map{"ok": true})
    })
    return app
}

func main() {
    website := newWebsite()
    api := newAPI()

    switcher := apps.Switch(apps.Hosts{
        {Pattern: "api.example.com", Target: api},
        {Pattern: "example.com", Target: website},
    })

    switcher.Listen(":80")
}
```

What `Switch` actually does: it builds each case application on its own `Build` (via `OnBuild`), then installs a `UseRouter` handler that evaluates the case filters in registration order. On the first match it forwards the request to that application's `ServeHTTP`: the matched app serves with its own context pool and configuration. If nothing matches, the request falls through to the switcher's own router, so you can register fallback routes or a custom 404 on the switcher itself. It is deliberately not a load balancer; it is a filter chain.

The `Hosts` provider matches the request host against each `Pattern`, which may be a literal or a regular expression (the `apps.AnyDomain` constant matches any domain). `Target` accepts an `*iris.Application`, the name of a registered application, or a plain host string; the latter produces a permanent redirect to that host. Providers combine with `apps.Join`, and the `apps.SetHost("canonical.example.com")` option rewrites the request host before the matched application sees it, useful for serving `www.` and the root domain from one application without redirects.

Standard Library Interop

Iris converts in both directions: standard handlers into Iris handlers, and the application itself into a standard `http.Handler`.

`http.Handler` is a one-method interface, `ServeHTTP(http.ResponseWriter, *http.Request)`, and it is the lingua franca of Go's HTTP ecosystem: metrics exporters, profilers, proxies and entire frameworks all either implement it or accept it. That shared contract is what makes both

conversions below mechanical.

Calls in the other direction, out to somebody else's service, have their own package: `github.com/kataras/iris/v14/client`, whose `client.New` takes a base URL, persistent request options, timeouts, rate limiters and a retry policy, and whose request helpers marshal and unmarshal JSON, form and multipart bodies, so a service does not hand-roll a fourth `*http.Client` wrapper. `c.BindJSON[Order](ctx, http.MethodGet, "/orders/42", nil)` names the response type at the call site and returns it, the way `ctx.BindJSON` does on the server side; a status of 400 or above comes back as an `APIError` that `rest/errors.HandleAPIError` forwards to your own client. A paginated Iris list reads back through `client.List[T]`, one page, or `client.Items[T]`, which walks every page as an `iter.Seq2[T, error]`; both send QUERY and decode into the same `Page[T]` the server built. Retries with backoff, a rate limit per client or per endpoint, and redaction of API keys from error messages and debug dumps are options, not code you write. The package is a thin layer over the author's standalone `github.com/kataras/httpclient`, with the framework's JSON policy as its default, so an application never imports that module directly.

Mounting http.Handler Values

`iris.FromStd` accepts an `http.Handler`, an `http.HandlerFunc`-shaped function, or a middleware-shaped `func(w, r, next http.HandlerFunc)` and returns an `iris.Handler`:

```go
package main

import (
    // Third-party: the Prometheus client.
    "github.com/prometheus/client_golang/prometheus/promhttp"

    "github.com/kataras/iris/v14"
)

func main() {
    app := iris.New()
    app.Get("/metrics", iris.FromStd(promhttp.Handler()))
    app.Listen(":8080")
}
```

A mounted handler reads route parameters with `r.PathValue(name)` and the matched route template, such as `/users/{id:uuid}`, from `r.Pattern`, in all three forms. `r.Pattern` stays empty where no route has matched yet or none will: in the `UseRouter` slot and in the built-in 404 and 405 handlers. Anything written against `net/http` mounts this way; see Chapter 17 for the full monitoring setup around that `/metrics` route.

The Application as an http.Handler

After `Build`, the application satisfies `http.Handler`, so it drops into any standard server, test harness or third-party listener:

```go
package main

import (
    "log"
    "net/http"

    "github.com/kataras/iris/v14"
)

func main() {
    app := iris.New()
    app.Get("/", func(ctx iris.Context) {
        ctx.JSON(iris.Map{"served_by": "net/http"})
    })

    if err := app.Build(); err != nil {
        log.Fatal(err)
    }

    srv := &http.Server{
        Addr:    ":8080",
        Handler: app, // *iris.Application implements http.Handler.
    }
    log.Fatal(srv.ListenAndServe())
}
```

`app.Listen` and `app.Run` call `Build` for you; call it yourself only when handing the application to infrastructure you control, as above. This is also how one Iris application embeds inside another: build the inner app and forward requests to its `ServeHTTP` from a route of the outer one.

Built-in Controllers

The `controller` package tree ships ready-made controllers that implement the same `Routing(r iris.Router)` interface as your own and register the same two ways:

```go
// Directly on a router:
app.Router("/static", fileserver.New("./assets", fileserver.DefaultOptions))

// Or through the builder's API step (see Chapter 3):
iris.NewBuilder().
    // ...builder steps...
    API("/static", fileserver.New("./assets", fileserver.DefaultOptions)).
    Build().
    Listen(":8080")
```

File Server

`fileserver.New(fileSystemOrDirectory, options)` serves static files from a directory path, an `http.FileSystem`, an `embed.FS` or an `fs.FS`. A directory path follows symbolic links that point outside it; pass `root.FS()` of an `os.OpenRoot` instead when that matters, and accept that a link you placed there on purpose stops being served (Chapter 18):

```go
package main

import (
    "embed"

    "github.com/kataras/iris/v14"
    "github.com/kataras/iris/v14/controller/fileserver"
)

//go:embed assets
var embeddedAssets embed.FS

func main() {
    app := iris.New()

    app.Router("/static", fileserver.New(embeddedAssets,
        fileserver.Options{
            IndexName:      "index.html",
            ShowList:       true,
            AssetValidator: fileserver.BlockHidden,
        }))

    app.Listen(":8080")
}
```

The options worth knowing: `IndexName` (directory index file), `ShowList` and `ShowHidden` (directory listings, with `DirList` for a custom renderer), `SPA` (serve the index instead of 404 for unknown files, as single-page applications), `Attachments` (download mode with rate limiting), `Cache` (in-memory caching and pre-compression), `PushTargets` (HTTP/2 push), and `AssetValidator`, a per-request `func(ctx iris.Context, name string) bool` gate. `fileserver.BlockHidden` is a ready validator that denies dotfiles such as `.git`, `.env` and `.htpasswd`. `fileserver.DefaultOptions` is a sensible starting value.

One more option is for sites made of plain HTML pages. The router removes the trailing slash of a directory URL, so the index page of `doc/` is served at `/static/doc`, and from there a browser resolves a relative link such as `css/app.css` against the parent path, `/static/`. Set `RedirectToIndex` and `/static/doc` redirects to `/static/doc/index.html` instead, where relative links resolve inside the directory. It is off by default and does nothing in `SPA` mode. Root-relative links never needed it.

The package also exports `Handler` (the same serving logic as a plain `iris.Handler`, no controller), `Favicon(path)` (a cached `GET|HEAD /favicon.ico` controller) and `WebDAV(handler)` (wraps a `golang.org/x/net/webdav` handler, third-party, registering all WebDAV methods on

`/{p:path}`).

API Graph

`apigraph.New` serves an interactive D3.js visualization of every registered route: paths, methods, handlers, with editor links to each handler's source line. The graph is built after the application builds, so registration order does not matter:

```go
import "github.com/kataras/iris/v14/controller/apigraph"

app.Router("/apigraph", apigraph.New(apigraph.Options{Theme: "dark"}))
```

`Options.Theme` is `"light"` (default) or `"dark"`; register custom themes with `apigraph.RegisterTheme`. Mount it in development builds: it is a map of your entire API surface.

Sitemap

A sitemap is an XML file that lists a site's pages for search-engine crawlers, optionally hinting how often each page changes and how important it is relative to the rest.

`sitemap.New(domain)` generates `/sitemap.xml` from the registered static routes at build time:

```go
import "github.com/kataras/iris/v14/controller/sitemap"

app.Router("/", sitemap.New("https://www.example.com"))

app.Get("/", index)
app.Get("/about", about).
    SetLastMod(time.Now()).
    SetChangeFreq("monthly").
    SetPriority(0.8)
app.Get("/internal", internal).ExcludeSitemap()
// GET /sitemap.xml now lists / and /about.
```

The route methods `SetLastMod`, `SetChangeFreq` and `SetPriority` fill the matching sitemap elements; `ExcludeSitemap` removes a route. Dynamic, subdomain and offline routes are excluded automatically, and above 50,000 entries the output splits into an indexed set. With the i18n middleware in place, `WithLocalization` emits translated alternate links per language.

Summary

- `mvc.UseController[*C]()` generates the five REST routes from a typed controller, with DI-resolved construction, nested resources via `With` and parent parameters via `mvc.PathParam[T, V]`; the `controller/mvc/handler` package exposes the same typed pipeline

(`Handler` , `WithStatusCode` , `WithPathParameter`) route-by-route, and `mvc.Websocket` / `mvc.GRPC` cover the non-REST protocols.

- `iris.NewContextWrapper[T]()` turns a struct embedding `iris.Context` into the argument type of your handlers; pool it for reuse with `context.NewContextWrapperWithPool` .
- `iris.Patches()` swaps framework internals (JSON and other body writers, domain resolution, cookie defaults, filesystem resolution) once, at startup, for the whole process.
- `app.Macros()` extends the path parameter language of one application: `RegisterFunc` adds functions to existing types, `Register` adds new types, and `SetErrorHandler` shapes the 400 response.
- `WrapRouter` , `AddRouterWrapper` and `PrependRouterWrapper` run raw `net/http` code before routing; `Downgrade` replaces the router entirely.
- The `apps` package is a process-wide application registry, and `apps.Switch` with `apps.Hosts` routes one listener across many applications by host pattern.
- `iris.FromStd` mounts any `net/http` handler, and after `Build` the application itself is an `http.Handler` .
- `controller/fileserver` , `controller/apigraph` and `controller/sitemap` are ready-made controllers for static assets, route visualization and sitemap generation.

Further Reading

- Go: Getting started with generics: the official hands-on introduction to type parameters.
- Go blog: An Introduction to Generics: why generics exist and how constraints work, from the Go team.
- Go: http.Handler: the one-method interface behind router wrappers and both interop directions.
- Go: sync.Pool: the recycling mechanism behind the pooled context wrapper.
- MDN: The WebSocket API: what the protocol behind `mvc.Websocket` provides.
- google.golang.org/grpc: the gRPC library that `mvc.GRPC` serves through the Iris router.

You have followed Iris from a first `iris.New()` to generic controllers, typed contexts and framework patches; there is no larger machinery left hidden.

Epilogue

Twenty-three chapters ago this book promised a path from `go mod init` to production. Here at the end, it is worth naming what you actually carry out of it, because it is more than a list of APIs.

What You Can Do Now

You can take an empty directory and, within the hour, stand up a service with typed routes, validated input, one coherent error format, dependency injection that makes testing trivial, and an HTTP test suite that runs in milliseconds. You can read a request from any angle (path, query, body, form, upload) and answer it with the right status, the right headers and the right shape. You can put authentication in front of anything, from a one-line Basic Auth guard to a token SDK with database-backed users. You know where every cross-cutting concern belongs, what the builder installs so you never install it twice, and how to find out (with a profiler, not a hunch) where the time really goes when something is slow.

Just as importantly, you know what the framework does *for* you: the secure defaults it ships, the panics it contains, the internals it keeps out of error responses, the connections it drains on shutdown. Good infrastructure is measured by the incidents that never happen; a lot of this book has quietly been about those.

The Habits Worth Keeping

If the details fade, keep the habits. They were the real curriculum:

- **Thin handlers.** Read, call, respond. Business rules live in services; plumbing lives in middleware; neither lives in a handler.
- **One place for errors.** Domain code returns meaningful errors; exactly one map decides what clients see. The moment you write a status code in a handler, ask why.

- **Constructors declare, containers provide.** Every dependency that arrives through a constructor is a dependency you can fake in a test.
- **Tests in milliseconds.** If the suite is fast, you run it constantly; if you run it constantly, you refactor without fear. In-memory HTTP testing is what makes that cheap.
- **Measure before optimizing, verify before trusting.** The profiler over folklore; the compiler and the test suite over hope.

None of these are Iris-specific. They will serve you in every service you ever write. Iris simply removes the friction between the habit and the code.

What to Build Next

Reading builds understanding; building makes it stick. Three suggestions, in rising order of ambition:

1. **Rebuild the Bookshelf from memory.** The framework's `TUTORIAL.md` walks it step by step; doing it without looking is the fastest way to find the parts that have not settled yet.
2. **Extend one of Chapter 22's applications.** Give the Stockroom API real persistence with `kataras/pg`, or give Shortl an admin party with usage statistics. Both programs were built to be grown.
3. **Port something real.** Take a service you already run (or the next one you are asked to write) and give it the shape this book teaches: the seams, the error map, the test suite. That is the moment the material becomes yours.

Staying Current

- **The live AI wiki** at codewiki.google/github.com/kataras/iris is the fastest way to get unstuck. It is a free, Google-generated wiki built from the Iris source itself and regenerated as the code evolves. Every page states the commit it reflects, so it stays current across releases without waiting for anyone to update it. Browse it like documentation (architecture diagrams included), or use its chat to ask questions in plain language and get answers grounded in the actual codebase. As with any AI, double-check what it tells you against the source, but as a first responder for "how does Iris do X?", it is hard to beat.
- **Plexon AI** at plexon.ai is a desktop assistant for Windows, macOS and Linux, built by the author of Iris. It ships an Iris skill built in, covering the same ground as this book in a form written for an agent to read, and its Software Developer persona turns it on.

- The framework lives at github.com/kataras/iris: releases, issues and discussions all happen there, and `HISTORY.md` records what each release changes.
- API reference documentation is on pkg.go.dev/github.com/kataras/iris/v14, generated from the same source you have been reading about.
- News and guides live at iris-go.com.
- If you ever bring an older codebase forward, the maintained migration guide is at github.com/kataras/iris/blob/main/MIGRATION.md.

Contributing

Iris has been built in the open since 2016, by hundreds of people who started exactly where you are now. Bug reports with a failing test, documentation fixes, middleware improvements, honest questions in the discussions. All of it moves the framework forward. The same goes for this book: when you spot something that could be clearer or wrong, say so in the discussions. Corrections from readers are how a book stays trustworthy.

Thank You

Software frameworks are wagers: that the hours one group of people pours into the plumbing will save many more hours for everyone who builds on top of it. By reading this far, and by whatever you build next, you are the other side of that wager paying off.

Thank you for reading. Now go build something good.

- Gerasimos Maropoulos