Go for JavaScript Developers

From React and Next.js to Go, and building real applications with Grit

Muke JohnBaptist

Written against Grit v3.403.0 and Go 1.25

Contents

Before you start: a repository, and a leaderboard 1
Why the assignments are pushed rather than just done 1
Step one: make the repository 1
Step two: the Committed app 3
Step three: join the Gopher League 3
Step four: the commit convention 3
The loop, for every chapter from here 4
There is more practice on the website 4
Why Go, when you already ship JavaScript 7
What actually pushed you here 7
The shape of Go, in one page 8
What you give up, stated plainly 8
The one example that explains everything 9
What this book does 10
How to read it 11
Installing Go and your first program 13
Installing 13
Your first program 13
Capital letters are the access modifier 14
Modules: go.mod is your package.json 15
The commands you will actually use 15
Compiling a real binary 16
Variables, types, and the compiler that argues with you 19
Declaring things 19
The types you will actually use 20
Zero values: there is no undefined 21
No truthiness 21
Strings are bytes, and that matters 22
Control flow, briefly 23
Functions, multiple returns, and the end of try/catch 27
Functions 27
Two return values, and why 28
`error` is just an interface 29
Wrapping: adding context without losing the original 29
The argument about verbosity 30

Named returns and `defer` 31
Structs and methods, against objects and classes 35
Declaring and creating 35
Methods: functions with a receiver 36
Value receivers and pointer receivers 36
Composition instead of inheritance 37
Struct tags: metadata for encoders 38
Constructors are just functions 38
Comparing structs 39
Slices: the array that is not an array 43
Arrays have a fixed length, in the type 43
Slices are a view onto an array 43
append, and the thing it does not promise 44
Iterating 45
Passing slices to functions 46
Sorting 46
Maps, and why iteration order is random on purpose 49
Declaring and using 49
Missing keys return the zero value 50
Iteration order is deliberately random 50
Maps are reference types 51
Structs as values, and the thing you cannot do 51
Sets 52
Maps are not safe for concurrent use 52
Pointers, without the fear 57
Everything in Go is passed by value 57
Two operators 58
nil, and the panic you will meet 58
When to use a pointer 59
When not to use a pointer 59
The loop variable, and a trap that Go fixed 60
Pointers to struct fields 61
Packages, modules and imports, against npm and ESM 65
A package is a directory 65
Importing 66
The import path is the module path plus the directory 66
internal/ is a real access rule 67
Naming, and the stutter rule 67
Modules, versions and go.mod 68

Major versions are in the path 69
init, and why to avoid it 69
Interfaces, and why Go's point the other way 73
The mechanical part 73
The direction: interfaces belong to the consumer 74
Keep them small 74
The empty interface, and `any` 75
The nil interface trap 76
Interfaces you will meet constantly 77
Why this matters for testing 77
Designing with errors: what a package should return 83
Three kinds of error, and when to use each 83
Wrap, but not at every level 84
Error strings are a contract, lowercase and unpunctuated 85
Do not return both a value and an error 85
Handle an error once 86
Mapping errors at the boundary 86
panic is not an error 87
defer, panic and recover 91
defer: cleanup that cannot be forgotten 91
The three rules of defer 92
defer can change the return value 93
defer and errors from Close 93
panic: stop, this should not have happened 94
recover: only for a boundary 94
Goroutines, against the event loop 99
The one-keyword version 99
What is actually different 100
Waiting properly: sync.WaitGroup 100
Collecting results 101
Races, and the detector that finds them 102
Bounding how many run at once 103
When not to use a goroutine 104
Channels, and the select statement 109
A channel is a typed pipe 109
Unbuffered channels are a handshake 110
Buffered channels have room 110
Closing, and ranging 110
select: waiting on several channels 111

Timeouts, the most common select 112
Signalling with a done channel 112
The nil channel, and why it is useful 113
Worker pools 114
sync: mutexes, and when a channel is the wrong tool 119
Mutex: one goroutine at a time 119
Embedding the mutex 120
RWMutex: many readers, one writer 120
sync.Once: exactly once, no matter who asks 121
atomic: for single values 121
sync.Map, and why you probably do not want it 122
When to use which 122
Context: cancellation, deadlines, and request scope 127
The problem it solves 127
Creating one 127
Using one 128
Cancellation is cooperative 129
The standard library already honours it 129
Timeouts compose 129
Values, and why to be sparing 130
The conventions 130
Generics, briefly and honestly 135
The syntax 135
Constraints are interfaces 136
The constraints you will actually use 136
Generic types 137
What the standard library gives you 138
When to write your own, and when not to 138
Interfaces are still usually the answer 139
Testing: the runner is already installed 143
The rules 143
Error messages are the whole skill 144
Errorf against Fatalf 144
Table-driven tests 144
Comparing things that are not numbers 145
Helpers and t.Helper 146
Setup and cleanup 146
Parallel tests 147

Testing errors 147
Coverage 148
Benchmarks 148
JSON, struct tags, and the capital letter that bites 153
Encoding and decoding 153
The capital letter rule, again 153
Struct tags in full 154
omitempty, and what it actually means 155
Decoding unknown shapes 155
Decoding is case-insensitive, and lenient 155
Streaming with Encoder and Decoder 156
Times, and the format you will meet 156
Custom marshalling 157
Files, streams and the io interfaces 161
Reading a whole file 161
Streaming line by line 161
os.Open against os.Create 162
Buffer your writes 162
io.Copy, and the pattern it enables 163
Writing testable code against interfaces 164
Paths, portably 165
Temporary files and directories 166
net/http: a server with no framework 171
Five lines 171
The two core types 172
Routing with ServeMux 172
Always configure a Server 173
Middleware is a function that wraps a handler 173
Passing values down: context, not globals 174
Graceful shutdown 175
Testing handlers without a server 176
Talking to a database with database/sql 181
Opening a database 181
The pool is the important part 181
Querying one row 182
Querying many rows 182
Writing, and what you get back 183
Placeholders are not optional 184

NULL, and why your struct fields lie 184
Transactions 185
The N+1 query problem 186
Should you use an ORM? 187
Project: a URL shortener 193
What it does 193
The schema and the store 193
Generating codes 195
Validating the URL 196
The server 197
Running it 201
The tests 201
Configuration, environment and secrets 205
Environment variables, the baseline 205
Parse once, into a struct 205
Fail at startup, not on the first request 207
Secrets 208
Flags, for tools rather than services 209
Building, cross compiling and shipping one binary 213
The build 213
Smaller binaries 213
Embedding the version 214
Cross compiling 214
Containers, in two stages 215
Embedding files into the binary 216
Production flags worth knowing 216
Automating releases 216
The twenty six problems, and what a framework is for 221
Two lists, and they belong to different people 221
The developer's twelve 221
The client's fourteen 222
Why these survive a decade of excellent libraries 223
The question a framework has to answer 223
What this means for the rest of the book 224
grit new: what lands, and the shape of a project 227
Installing 227
Creating a project 227
What arrives 228
The internal packages 228

Running it 229
grit.json, and the seeding that matters 230
Modules you can switch off 230
It is still just Go 231
The architecture: a modular monolith, layer by layer 233
The short answer 233
The request path 233
The rule the layering exists for 234
Where the rules live, concretely 235
The event bus 235
What it deliberately is not 236
Organised by layer, not by feature 237
Can you refactor to something else? 237
The resource, and why it is the unit of generation 241
The claim 241
One command 241
Why one description is the point 242
A field type is a decision made once 242
It is still your code afterwards 243
What it does not generate 244
Field types: what each one decides 247
The argument 247
The full list 247
Text in three sizes 248
Choices, in three shapes 248
Relationships 248
Identity and files 249
Validated strings 249
Numbers with meaning 249
Time, in three precisions 250
Choosing well, in one table 250
Models, migrations and seeding at scale 253
The model 253
Migrations, and the honest problem 254
The refusal that protects you 254
Seeding, and why it is not optional 255
A million rows, with the heap flat 255
Backups, and the half that counts 256
Services: where the rules live, and why not the handler 259

The rule, and the reason 259
What a generated service contains 260
The two whitelists are the chapter 260
Ownership travels in the context 261
Errors, by design 261
Adding your own method 262
Where a rule does not belong in a service 263
Handlers and the HTTP contract 267
A handler, in full 267
One envelope, everywhere 267
Status codes, and the ones that matter 268
404 as a permissions answer 269
The 409, and a number that must not be a string 269
respond, and the 119 codes 270
Validation, in two places that agree 271
Writing your own handler 271
The generated admin panel 275
What you get without asking 275
How it fits together 275
Customising a resource 276
The list page, honestly 277
Where it stops 277
The hooks underneath 278
Authentication, roles and per-record ownership 281
Three questions, not one 281
What arrives 281
Sessions, not just tokens 282
Four layers on the sign-in form 282
Roles, and adding your own 283
Ownership is the one that bites 283
Multitenancy is the same idea one level up 284
What is not solved for you 284
The shared package: one definition, four outputs 287
What is in it 287
A generated pair, in full 288
Why three, and what each one is for 289
nullable against optional, which is not pedantry 289
grit sync, and exactly what it does 289

The admin resource is updated, not overwritten 291
Where the chain stops, honestly 291
Using it on the frontend 292
Constants, themes and brand 292
Project: a task tracker, end to end 295
What we are building 295
Step one: the project 295
Step two: the resources 296
Step three: the rule, in the only place it can live 297
Step four: the endpoint 298
Step five: data worth looking at 298
Step six: the admin screen, customised 299
Step seven: a page the admin cannot be 300
Step eight: a form for people without accounts 301
Step nine: the things that make it a product 301
What it cost 301
Relationships, and the forms they generate 305
The four shapes, and the question each answers 305
belongs_to, and everything it writes 305
The admin dropdown, and the label it shows 306
one_to_one is belongs_to plus one index 307
A relation pointing at itself is a tree 307
many_to_many, and the table you did not name 308
Line items: a child inside its parent 309
What relationships cost on a list page 309
Choosing well 310
Project: a publication, from draft to public page 313
The model you already have 313
The one that is not optional: sanitising HTML 314
Draft and published, as a service rule 314
The public surface 315
The cache, and the one rule about it 316
Rendering it in the web app 316
The things a publication needs that nothing generates 317
Money, dates, and the types that prevent bugs 321
Why a float is the wrong type for money 321
Money: an integer and a currency 321
The operations, and which ones can fail 322

Not every currency has two decimal places 323
Dates: the type that exists because of one error message 324
date against datetime, and how to choose 324
The same move, in smaller places 325
Project: an invoicing application 329
What makes this harder than CRUD 329
Numbering, which is a concurrency problem in disguise 329
Immutability, and why an edit is the wrong verb 330
The state machine, written as methods 331
Payments, and the status you should not store twice 332
The document itself 333
Reminders, in one line for now 333
File uploads and the image pipeline 337
A file field, and what it stores 337
Where the bytes go 338
What happens to an image on the way in 338
Your own profile, when the default is wrong 339
The decompression bomb, which the size limit cannot see 340
When the pipeline fails 340
Uploading from the frontend 341
Serving them 341
Background jobs, cron and transactional email 345
Why not just a goroutine 345
A job, generated 345
Return an error to retry, return nil to stop 346
The two kinds of “it ran twice” 347
Cron, and the five fields 348
Mail, and the drivers 348
Queued or inline, decided once 349
Seeing it work 349
The event bus, realtime, and webhooks 353
The problem, in the framework’s own history 353
Three deliveries, and choosing between them 353
The outbox, and the choice it removes 354
Realtime: channels, not just users 355
When WebSockets do not arrive 356
Webhooks, outbound 356
Observability: Studio, Pulse and Sentinel 359
Four questions, four tools 359

Studio: the database, as it is 359
Pulse: where the time went 360
Sentinel: the attacks you were not watching 361
Health: the one endpoint a load balancer reads 361
The logs, and the ID that joins everything 362
Making it a habit 362
Testing a Grit application 365
What arrives already written 365
The in-memory database trap 365
What is worth testing, and what is not 366
Service tests, without HTTP 367
Handler tests, for the contract 367
The highest-value test in the whole project 368
Allocation budgets, and a trap in the toolchain 368
The frontend, in two layers 369
What CI should run 369
Upgrading, deploying, and handing it over 373
The upgrade problem, stated honestly 373
What an upgrade actually does 374
The CLI updates itself 374
Deployment, in the order of effort 375
Two deployment bugs worth not repeating 375
The checklist before a first deploy 376
Handing it over 376
Where to go next 377
Build your own CRUD API from scratch 379
What we are building 379
Three layers, decided before any code 379
The type and its rules 380
The store 381
Listing, which is where the work actually is 382
A router, written rather than imported 384
One envelope, one place 386
Handlers, which are now boring 387
Middleware, as a chain 388
Wiring it up 389
Testing the whole thing without a port 389
Now count it 390

Build your own CLI framework 393
What a good CLI does, before any code 393
The command 393
Dispatch 394
Two kinds of failure 395
Help, generated 396
Did you mean 397
Cancellation that is not a kill 397
Output humans read and machines parse 398
Testing a CLI 398
Build your own JSON parser 401
The grammar, in full 401
Tokens 401
The lexer 402
Strings, where the work is 403
Numbers, and the decision JavaScript cannot make 404
The parser 405
Two defences against hostile input 406
Errors people can act on 407
Testing it properly 407
How fast is it 408
Build your own key-value store: a write-ahead log and crash recovery 411
The problem, stated precisely 411
The record format 411
Encoding a record 412
The store 413
Replay, and tolerating a torn tail 414
Writing 415
The one system call that matters 416
Compaction 417
Build your own on-disk index: pages and a B-tree 423
Why a tree and not a hash 423
Pages, the unit of everything 424
Leaves hold data, internal nodes hold directions 424
Searching 425
Inserting, and the split 425
Deleting, honestly 427
The free list 427

Durability, and the page that was half written 427
A buffer pool, because reading the same page is silly 428
Testing a data structure you cannot see 428
Build your own Redis: RESP over raw TCP 431
The protocol, in one page 431
The server loop 432
Reading a command 433
The store, and the lock 434
Commands, as a table 435
Incrementing, which is where atomicity shows up 435
Persistence, which you have already built 436
Testing against the real thing 436
What you are not building 437
Build your own HTTP server, from a TCP socket 439
What a request actually is 439
The accept loop, with the timeouts that matter 440
Parsing the request 441
Where does the body end 442
Keep-alive, and the body you forgot to read 443
Writing the response 444
100-continue, the handshake nobody knows about 445
Routing, which you have already written 445
What you are not building 445
Testing it 445
Build your own Git 449
Everything is an object, named by its hash 449
init, and the directory 450
Writing and reading objects 450
Trees, and the format that catches everybody 451
Commits 452
References, and what a branch is 453
Walking the log 453
The index, and why it exists 454
What you are not building 454
Build your own reverse proxy and load balancer 457
What a reverse proxy does 457
The core, which is shorter than you expect 457
Hop-by-hop headers, and why they must be removed 458

X-Forwarded-For, and the header that lies 458
Choosing a backend 459
Health, actively and passively 460
Retries, and the one that corrupts data 461
Streaming, and the buffering that breaks it 461
Draining a backend 462
Build your own job queue 465
The table 465
Enqueue, inside your transaction 466
Claiming work, which is the whole problem 466
When a worker dies 467
Failure, backoff and the dead letter 468
Polling, and the latency it costs 469
Fairness, and the queue that starves 469
Knowing it works 469
Shutting down without losing work 470
When this is the wrong design 470
Build your own test runner, and what go test really does 473
The trick: it writes a program 473
Step one: discovery 474
Step two: generate and build 475
Step three: running, in parallel, in isolation 476
Step four: output people can use 477
The features that make a runner good 478
What `go test` does that yours will not 478
Appendix: the JavaScript to Go phrasebook 481
Values and types 481
Collections 481
Functions and errors 482
Concurrency 482
Modules and tooling 483
The standard library, by what you would npm install 483
Five habits to unlearn 484
Afterword: what to do on Monday 485
The three things that actually move you 485
The uncomfortable parts, named 485
What to learn next, in order 486
On using a framework at all 486

Thank you, and one request 486

0

Before you start: a repository, and a leaderboard

Every chapter in this book ends with an assignment. You write it, you commit it with an exact message, and you push it. This chapter sets that up, and it takes about ten minutes.

Do it now rather than at chapter four, because the habit is the point and a habit started late is usually not started.

Why the assignments are pushed rather than just done

Three reasons, and the third is the one that matters.

You will finish them. Work that leaves your machine gets finished. Work that stays in a scratch file gets abandoned at the interesting part, which is usually one step before the thing it was teaching.

You will have something to show. By the end of this book your repository holds fifty-odd small Go programs, each one solving a named problem, each with a readable commit history. That is a more convincing portfolio than a single large project, because it shows range and it shows persistence.

You are not doing it alone. The commits are scored on a leaderboard shared with everybody else working through this book, which is what the rest of this chapter is about.

Step one: make the repository

```
mkdir go-journey && cd go-journey
git init
```

One repository for the whole book. Not one per chapter: the history is the record, and a single timeline is what makes the progression visible.

Give it a structure that will still make sense at chapter fifty:

```
go-journey/
├── README.md
├── part1-language/
│   ├── 01-why-go/
│   ├── 02-hello/
│   └── ...
├── part2-ideas/
├── part3-standard-library/
├── part4-grit/
├── part5-building/
└── part6-from-scratch/
```

Each assignment gets a directory with its own `main.go`, and where it needs one, its own `go.mod`:

```
mkdir -p part1-language/02-hello && cd part1-language/02-hello
go mod init go-journey/part1/hello
```

WATCH OUT

A separate `go.mod` per assignment keeps them independent, so an experiment in chapter twenty cannot break chapter three. The alternative, one module at the root with a package per chapter, also works and means one `go.mod`, but every `go build ./...` then compiles everything.

For a learning repository, independent modules are the better trade. Use a Go workspace if you want `go build ./...` to still work across all of them:

```
go work init
go work use ./part1-language/02-hello
```

Write a `README.md` on day one, because you will not write it later:

```
# go-journey

Working through *Go for JavaScript Developers*, one assignment per chapter.

| Part | What it covers |
|---|---|
| 1 | The Go language, against the JavaScript I already knew |
| 2 | Interfaces, errors, goroutines, channels, context |
| 3 | Files, HTTP, databases, shipping a binary |
| 4 | The Grit framework |
| 5 | Building real applications |
| 6 | Building things from scratch: a parser, a database, a server |
```

Then push it:

```
git add .
git commit -m "Setup - Repository"
git branch -M main
git remote add origin git@github.com:YOUR-NAME/go-journey.git
git push -u origin main
```

Make the repository **public**. The leaderboard reads public activity, and a public learning repository has never cost anybody a job.

Step two: the Committed app

Install Committed from the Play Store:

```
play.google.com/store/apps/details?id=com.desishub.committed
```

Sign up **with your GitHub account**, which is how it sees your commits. There is nothing to configure after that.

Step three: join the Gopher League

Open the app and join with the code:

```
5TCLJ27Z
```

Or follow the link, which does the same thing:

```
https://committed-api.gitframework.dev/join/5TCLJ27Z
```

The group's description says what it is:

> *Gopher League is a friendly leaderboard for developers learning Go together. Every commit, merged PR, review and closed issue earns points. Ship Go, climb the table, and get crowned weekly champion.*

Commits score. So do merged pull requests, reviews and closed issues, which is worth knowing: reviewing somebody else's assignment is worth points, and it is also one of the better ways to learn.

Step four: the commit convention

Every assignment in this book carries an exact commit message, printed in its own box:

```
git commit -m "Variables - Assignment One"
```

Use the message exactly as printed. It is not decoration. The leaderboard matches on it, and a commit called "did the variables thing" earns nothing, which is a sad way to lose a week's points.

The pattern is always the same:

```
<Topic> - Assignment <Number>
```

So the first few are `Hello Go - Assignment One`, `Variables - Assignment One`, `Variables - Assignment Two`, and so on.

> **WATCH OUT**
> Commit **per assignment**, not once per chapter and not once per day.
>
> Partly because each one scores. Mostly because a commit that says exactly what it contains is the single most useful habit in this book, and a history of fifty precise commits is worth more to you in six months than fifty files in one commit called "work".

The loop, for every chapter from here

```
# 1. a directory for the assignment
mkdir -p part1-language/03-types && cd part1-language/03-types
go mod init go-journey/part1/types

# 2. write it, and run it until it works
go run .

# 3. commit with the exact message from the book
git add .
git commit -m "Variables - Assignment One"
git push
```

Four commands, maybe fifteen minutes per chapter. The leaderboard updates on its own.

There is more practice on the website

The Grit documentation has a Go guide with forty six interactive exercises, which run in the browser with a solution to check against:

```
gritframework.dev/docs/prerequisites/golang
```

They cover the same ground as Parts I to III and are shorter than this book's assignments, so they work well as a warm-up before a chapter or as revision after it. Where one lines up with a chapter, that chapter's assignment says so.

The division worth knowing: **the website exercises are for practising a concept, and this book's assignments are for building something and pushing it**. Do both if you have time, and if you only do one, do the assignment, because that is the one that leaves your machine.

ASSIGNMENT

Set up everything above, and prove it works.

One. Create the `go-journey` repository, public, with the directory structure and the README.

Two. Install Committed, sign up with GitHub, and join Gopher League with the code `5TCLJ27Z`.

Three. Add a file `part1-language/00-setup/notes.md` answering the question from the end of chapter one in two sentences: what is the specific thing in your current stack that made you pick this book up? You will want that answer again at chapter twenty six.

Four. Commit and push, then open the app and confirm your commit appears on the leaderboard. If it does not, the usual cause is that the repository is private or the email on the commit does not match your GitHub account:

```
git config user.email "the-email-on-your-github-account"
```

COMMIT AS

```
git commit -m "Setup - Repository"
```

YOUR TURN

Open the leaderboard and find your position. It will be last, which is the correct place to start and the only time the number matters.

Then look at who is near the top and how often they commit. Almost always the answer is "a little, most days" rather than "a lot, once". That pattern is the actual lesson of the leaderboard, and it applies to learning Go whether or not anybody is counting.

1

Why Go, when you already ship JavaScript

You already build and ship software. You know React, you know Next.js, and you can take an idea to production without asking anybody how. So the first question this book owes you is not "what is Go", it is "why would you spend a month learning a second language when the one you have works".

Here is the honest answer, and then the honest list of what you give up.

What actually pushed you here

Most JavaScript developers do not come to Go out of curiosity. They come because something in their stack got painful in a specific way. Usually one of these four.

The deployment got heavy. Your Next.js app needs Node on the server, a `node_modules` directory with a few hundred megabytes in it, and a build step that has to run somewhere. You want to copy one file onto a server and run it. A compiled Go program is one file. It has no runtime to install, no package directory, and no build step on the target machine. You build it on your laptop and the output is the thing that runs.

The work stopped being request-and-respond. An API that answers a question in forty milliseconds is a perfect job for Node. A process that holds ten thousand open connections, or runs a job queue, or chews through a large file while staying responsive, is where the single-threaded event loop stops being an elegance and starts being a constraint you work around.

You wanted the errors earlier. In JavaScript, a renamed field is found by a user. TypeScript helps enormously and you should keep using it, but it is a layer on top of a language that does not have types, and it disappears at runtime. In Go the types are the language. A program that compiles has already had a whole class of mistake removed from it.

Somebody said the word "backend" and meant it. Payment handling, a scheduler, an audit trail, something that must still be correct at 3am when nobody is watching. Go was built by people who operate systems like that for a living, and it shows in what the language refuses to let you do.

> **IF YOU KNOW JAVASCRIPT**
> None of this makes Go better than JavaScript. It makes it different in ways that matter for one kind of work. You are not replacing React. In the second half of this book you will have a Go backend and a React frontend in the same project, talking to each other with types generated from one definition. That is the shape most people actually want.

The shape of Go, in one page

Before any syntax, here is the character of the language, because it explains every decision you are about to meet.

Go is **small on purpose**. The entire specification is about one hundred pages. There are twenty five keywords. JavaScript has roughly double that and a standard library that has grown for thirty years. You can hold all of Go in your head, which is a design goal rather than an accident, and it is why Go teams onboard people in days rather than weeks.

Go is **explicit on purpose**. There is no implicit type coercion, no `this` that changes meaning depending on how a function was called, no prototype chain to walk. When you read a line of Go, what happens is what is written. This is sometimes tedious and it is never surprising, and after a few weeks you will notice you have stopped debugging a category of problem entirely.

Go is **opinionated about formatting**, which sounds trivial and is not. There is one correct way to format Go code, a tool called `gofmt` produces it, and every Go project on earth uses it. No Prettier config, no debate about semicolons, no `.editorconfig`. The argument is over because the language ended it.

Go **compiles fast**. Not "fast for a compiled language", fast in absolute terms: a medium project builds in a couple of seconds. This matters more than it sounds, because it means the compile step never becomes a reason to avoid checking your work.

What you give up, stated plainly

A book that only lists advantages is marketing. Here is the other side, and you should weigh it before chapter two.

You will write more lines. Go has no `map`, `filter` or `reduce` built into the language the way JavaScript does. Where you write one chained expression, you will often write a four-line loop. Go's position is that the loop is easier to read six months later. You may disagree, and you will still write the loop.

Error handling is verbose. You are about to meet `if err != nil` and you will type it hundreds of times. There is no `try` / `catch` wrapping half a function. Every single oper-

ation that can fail says so, at the point it can fail. This is the single biggest adjustment coming from JavaScript, and chapter four is entirely about making peace with it.

The ecosystem is smaller. npm has over two million packages. Go has far fewer. For web application work this matters less than you would think, because Go's standard library covers an enormous amount of what you would reach for a package to do in Node. But if you need an obscure SDK, check before you commit.

No null safety. Go has `nil`, and a nil pointer dereference will crash your program at runtime exactly like reading a property of `undefined` crashes yours. Go's type system is simpler than TypeScript's, and in this one specific area TypeScript is genuinely ahead.

> **WATCH OUT**
> If someone tells you Go is simply better than JavaScript, stop listening to them. The two languages were built for different jobs by people with different problems. Go's concurrency is better than the event loop for sustained work, and the event loop is a beautiful fit for a UI. Use both.

The one example that explains everything

Here is the same program in both languages: read three URLs, and print how long each took. It is worth studying before you understand any Go syntax, because the difference in shape is the whole argument.

JAVASCRIPT

```
async function timeAll(urls) {
  const results = await Promise.all(
    urls.map(async (url) => {
      const start = Date.now();
      await fetch(url);
      return { url, ms: Date.now() - start };
    })
  );
  for (const r of results) {
    console.log(r.url, r.ms);
  }
}
```

GO

```
func timeAll(urls []string) {
    var wg sync.WaitGroup
    results := make([]result, len(urls))

    for i, url := range urls {
        wg.Add(1)
```

```
        go func() {
            defer wg.Done()
            start := time.Now()
            http.Get(url)
            results[i] = result{url, time.Since(start)}
        }()
    }

    wg.Wait()
    for _, r := range results {
        fmt.Println(r.url, r.ms)
    }
}
```

The JavaScript is shorter and, if you know JavaScript, clearer. Do not let anybody tell you otherwise.

But look at what the Go version is doing. The word `go` in front of that function call starts a **goroutine**: an independent line of execution that the Go runtime schedules onto a real operating system thread. Three URLs here, but the same code with fifty thousand URLs would start fifty thousand goroutines, and Go would handle it, because a goroutine costs about two kilobytes of memory rather than the megabyte an OS thread costs.

The JavaScript version is also concurrent, and for three HTTP requests it is just as fast, because all three are waiting on the network and the event loop is excellent at waiting. The difference appears when the work is not waiting. If each of those tasks had to parse a large document or resize an image, the JavaScript version would do them one at a time on one core, and the Go version would use every core you have.

> **THE MENTAL MODEL SHIFT**
> In JavaScript, concurrency is something you **await**, and parallelism is something you get by starting more processes. In Go, concurrency is something you **start**, with one keyword, and parallelism comes free because the runtime spreads goroutines across cores. Chapter eleven is where this clicks properly. For now, notice only that `go doSomething()` is the entire syntax.

What this book does

This book has two halves and they have different jobs.

Part one through three teach you Go, assuming you know JavaScript and nothing else. Every concept is introduced by naming the JavaScript thing it resembles, then showing exactly where the resemblance breaks. You will build small things constantly:

a CLI tool, a JSON API with no framework at all, a URL shortener backed by a real database. By the end you can read and write Go without this book open.

Part four and five teach you Grit, which is a framework that generates a Go API, a React admin panel and the typed client between them from one description of your data. The reason it comes second is that a framework you cannot read is a liability. Once you know Go, Grit's generated code is just Go you did not have to type, and you can change any of it.

> **WHAT YOU WILL HAVE BUILT BY THE END**
> A command line tool that formats JSON. An HTTP API written with nothing but the standard library. A URL shortener with a SQLite database. A task tracker, a blog with a working admin panel, and an invoicing application with customers, line items and money handled correctly. All of them small, all of them finished, none of them a toy that stops at the interesting part.

How to read it

Type the code. Do not copy and paste it. This is advice everybody gives and almost nobody follows, so here is the specific reason it matters for Go: the compiler's error messages are a large part of how you learn the language, and you only see them if you make mistakes. Typing produces mistakes. Pasting does not.

Run everything. Every code block that is a complete program is marked, and you should run it, break it deliberately, and run it again. Go builds in seconds, so the cost of an experiment is almost nothing.

Do the "Your turn" boxes. They are short, and they are the difference between recognising Go and writing it.

> **YOUR TURN**
> Before the next chapter, answer this for your own situation in two sentences: what is the specific thing in your current stack that made you pick this book up? Write it down. In chapter twenty three there is a list of twenty six problems that every project runs into, and you will want to know which of them is yours.

> **ASSIGNMENT**
> Before any Go, write down why you are here, because chapter twenty six will ask you to check the answer.
>
> Create `part1-language/01-why-go/NOTES.md` and answer three things in a paragraph each.

One. Which of the four pressures in this chapter is yours: the deployment, the long-running work, the late errors, or the word “backend”? Name the actual system it happened on.

Two. What did you try first, and why did it not settle the matter? There is almost always a library or a rewrite that nearly worked.

Three. What would have to be true in six months for this book to have been worth the time? Be specific enough that you could be wrong.

COMMIT AS

```
git commit -m "Why Go - Assignment One"
```

2

Installing Go and your first program

This chapter gets Go on your machine, gets a program running, and introduces the toolchain. If you have ever set up a Node project you will find this shorter than you expect, because most of the steps you are used to do not exist.

Installing

Go to go.dev/dl and download the installer for your platform. There is no version manager to install first, no equivalent of `nvm`, and for most people there never needs to be. Go's compatibility promise means a program written for Go 1.0 in 2012 still compiles today, so "which version" is rarely a question that costs you an afternoon.

Check it worked:

```
go version
```

You should see something like `go version go1.25.0 windows/amd64`. If the command is not found, the installer did not put Go on your `PATH`; restart your terminal first, because that fixes it more often than anything else.

> **IF YOU KNOW JAVASCRIPT**
> There is no `nvm`, no `.nvmrc`, and no per-project runtime. One Go installation builds every project on your machine. When a project needs a specific version, it says so in one line of `go.mod` and the toolchain downloads that compiler itself.

Your first program

Make a directory and put one file in it.

```
// main.go
package main

import "fmt"

func main() {
    fmt.Println("Hello from Go")
}
```

Run it:

```
go run main.go
```

Four things in that tiny file deserve explanation, because each is a rule you will meet constantly.

`package main` declares which package this file belongs to. Every Go file starts with a package declaration. The name `main` is special: it means this package builds into a runnable program rather than a library.

`import "fmt"` pulls in the formatting package from the standard library. Note there is no `npm install`. `fmt` ships with Go, as does an HTTP server, a JSON encoder, a test runner, a crypto suite and a templating engine.

`func main()` is the entry point. A program starts here. There is exactly one `main` function in a program.

`fmt.Println` prints a line. Capital `P` matters enormously, and it is the next section.

Capital letters are the access modifier

This is the first genuinely surprising rule in Go, and it catches everybody.

An identifier starting with a capital letter is exported. One starting with a lowercase letter is private to its package. That is the entire visibility system. There is no `public`, no `private`, no `export` keyword.

```
package money

// Convert is callable from other packages, because it is capitalised.
func Convert(cents int) string { ... }

// round is not. Only code inside package money can call it.
func round(f float64) int { ... }
```

> **IF YOU KNOW JAVASCRIPT**
> In JavaScript you choose what leaves a module with `export`. In Go the name itself decides. `fmt.Println` works because `Println` is capitalised inside package `fmt`; if it were `println` you could not reach it. This applies to functions, types, struct fields, constants and variables alike.

> **WATCH OUT**
> This bites hardest with struct fields and JSON. A lowercase field is invisible to the JSON encoder, because the encoder lives in another package and genuinely cannot see it.

A struct whose fields are all lowercase encodes to `{}`, with no error and no warning. Chapter sixteen covers this properly, but remember the symptom now: empty JSON usually means lowercase fields.

Modules: go.mod is your package.json

A single file is fine for a demo. A real project is a **module**. Initialise one:

```
mkdir hello && cd hello
go mod init github.com/yourname/hello
```

That writes `go.mod`:

```
module github.com/yourname/hello

go 1.25
```

That is the whole file. Compare it to a `package.json` for an equivalent project and notice what is missing: no `scripts`, no `devDependencies`, no `main`, no `type: module`. The module path doubles as the import path, which is why it looks like a URL: if you push this to GitHub, anybody can import it at exactly that address with no registry publish step.

Add a dependency and `go.mod` grows a line:

```
go get github.com/google/uuid
```

```
module github.com/yourname/hello

go 1.25

require github.com/google/uuid v1.6.0
```

IF YOU KNOW JAVASCRIPT

`go.sum` is your lockfile, equivalent to `package-lock.json`. Commit it. There is no `node_modules`: downloaded packages go to a shared module cache outside your project, and the compiler links what it needs directly into the binary. Deleting a project directory does not take 300MB of dependencies with it, because they were never there.

The commands you will actually use

Six commands cover almost everything.

`go run .`	Compile and run the current package, without keeping the binary
`go build`	Compile to an executable in the current directory
`go test ./...`	Run every test in this module
`go mod tidy`	Add what is imported, remove what is not. Run it often
`gofmt -w .`	Format every file. Your editor probably does this on save
`go vet ./...`	Report suspicious code the compiler allows

The `./...` pattern means "this directory and everything under it", and you will type it constantly.

Compiling a real binary

This is the part that sells Go to people who have deployed Node.

```
go build -o hello .
./hello
```

You now have one executable file. It contains your code, every dependency, and the Go runtime. The target machine needs nothing installed. You can also build for a platform you are not on:

```
GOOS=linux GOARCH=amd64 go build -o hello-linux .
```

That produces a Linux binary from a Mac or Windows laptop, with no container, no cross-compilation toolchain and no configuration. Copy it to a Linux server and it runs.

PROJECT 1: A GREETING WITH ARGUMENTS

Write a program that takes a name as a command line argument and greets it, falling back to "world" when none is given. You need `os.Args`, which is a slice of strings where `os.Args[0]` is the program name itself.

```
package main

import (
    "fmt"
    "os"
)

func main() {
    name := "world"
    if len(os.Args) > 1 {
        name = os.Args[1]
```

```
    }
    fmt.Printf("Hello, %s\n", name)
}
```

Build it, run `./hello Ada`, then run it with no argument. Then break it deliberately: change `len(os.Args) > 1` to `len(os.Args) > 0` and run it with no argument to see what a real Go panic looks like. Reading panics early is worth more than avoiding them.

YOUR TURN

Extend the greeting to accept several names and greet each on its own line. You will need a `for` loop over a slice, which looks like `for _, n := range os.Args[1:] { ... }`. The next chapter explains every piece of that line, so guessing now is fine and the compiler will tell you when you are wrong.

ASSIGNMENT

A module, a program, and a binary you can hand to somebody.

One. In `part1-language/02-hello`, run `go mod init`, write a program that prints a greeting, and run it with `go run .`.

Two. Add a second file in the same package with one exported and one unexported function. Call both from `main`. Then try to call the unexported one from a different package and keep the compiler error in a comment.

Three. Build a real binary and commit its size, not the binary itself:

```
go build -o hello .
ls -lh hello > BUILD.txt
```

Four. Cross compile the same program for another operating system and add that line to `BUILD.txt`. You now have the one-file deployment this chapter claimed, and you did it in two commands.

The website's **Your First Program** and **Print/Println and Raw Strings** exercises cover the same ground in the browser if you want the warm-up.

COMMIT AS

```
git commit -m "Hello Go - Assignment One"
```

3

Variables, types, and the compiler that argues with you

This is the chapter where Go feels most foreign, and it is worth knowing why in advance. In JavaScript, a variable is a box that holds anything and can hold something else tomorrow. In Go, a variable has one type forever, decided when it is created, and the compiler refuses to build your program if you break that promise.

That refusal is the feature. Work with it for a week and you will notice a whole category of bug has stopped happening to you.

Declaring things

Go has two ways to make a variable, and you will use one of them ninety percent of the time.

```
var name string = "Ada"   // the long form, type stated
var age = 36              // type inferred, still a var
// short form: declare and infer, inside functions
name := "Ada"
```

The `:=` operator is the one you want. It declares a new variable and infers its type from the value on the right. It only works inside a function, which is why package level variables use `var`.

> **IF YOU KNOW JAVASCRIPT**
> `:=` is closest to `let` in that it creates a new binding. There is no equivalent of `const` for variables created with `:=`. Go's `const` exists, but it is only for values known at compile time: numbers, strings, booleans. You cannot declare a constant struct or slice. If you want "do not reassign this", Go's answer is to write a small enough function that it is obvious.

An unused variable is a **compile error**, not a warning.

```
func main() {
    count := 10
```

```
    fmt.Println("hello")
}
// ./main.go:4:5: declared and not used: count
```

This feels hostile on day one and reasonable by day ten. Dead variables are usually the residue of a half-finished edit, and Go would rather stop you than let the residue accumulate. The same rule applies to unused imports.

> **WATCH OUT**
> The escape hatch is `_`, the blank identifier, which means "I know there is a value here and I am deliberately ignoring it". You will use it constantly in loops and multiple returns. Do not reach for it to silence a real warning: `_ = count` compiles and tells the next reader nothing.

The types you will actually use

Go has a lot of numeric types. You will use about four of them.

Type	JavaScript	Notes
`string`	`string`	Immutable, UTF-8 bytes. Indexing gives a byte, not a character
`int`	`number`	Whole numbers. 64 bit on any machine you own
`float64`	`number`	Decimals. The default for a fractional literal
`bool`	`boolean`	`true` or `false`. No truthiness, ever
`byte`	-	Alias for `uint8`. A byte of data
`rune`	-	Alias for `int32`. One Unicode code point

The one to notice is that JavaScript's single `number` splits into `int` and `float64`, and **Go will not convert between them for you**.

```
count := 10          // int
price := 2.5         // float64
total := count * price
// invalid operation: count * price (mismatched types int and float64)
```

You must say what you mean:

```
total := float64(count) * price   // 25
```

> **IF YOU KNOW JAVASCRIPT**
> JavaScript coerces almost anything: `"3" * 2` is `6`, `[] + {}` is a string, and `0.1 + 0.2` is famously not `0.3`. Go does none of this. There is no implicit conversion between any two types, not even `int` and `int64`. Every conversion is written out, which is more typing and removes an entire family of surprise.

Zero values: there is no undefined

Every type in Go has a **zero value**, and a declared variable always has it. There is no `undefined`, and no uninitialised memory.

`string`	`""`
`int`, `float64`	`0`
`bool`	`false`
pointers, slices, maps, interfaces, functions	`nil`

```
var name string
var count int
var ready bool
fmt.Println(name == "", count == 0, ready == false) // true true true
```

This is genuinely useful. A struct you create is immediately valid, with every field at a sensible default, and you never check whether a field "exists".

> **WATCH OUT**
> It is also the one place zero values hurt: you cannot tell "the caller set this to zero" apart from "the caller did not set it". If that distinction matters, for example a form field where empty and absent mean different things, you need a pointer (`*int`, where `nil` means absent) or a dedicated type. Chapter eight covers pointers properly. This problem is real, and it is the usual reason you see `*string` in a struct.

No truthiness

An `if` in Go takes a `bool`. Not a value that is bool-ish. A `bool`.

JAVASCRIPT

```
if (name) {
  greet(name);
}
if (items.length) {
  render(items);
}
```

GO

```go
if name != "" {
    greet(name)
}
if len(items) > 0 {
    render(items)
}
```

More characters, and no memorising which values are falsy. There is no `==` and `===` distinction either, because there is only one `==` and it does not coerce.

Strings are bytes, and that matters

A Go string is a read-only slice of bytes holding UTF-8. Indexing it gives you a byte.

```go
s := "héllo"
fmt.Println(len(s))     // 6, not 5: é takes two bytes
fmt.Println(s[1])       // 195, a byte value, not "é"
```

To work with characters, range over the string, which decodes runes:

```go
for i, r := range "héllo" {
    fmt.Printf("%d: %c\n", i, r)
}
// 0: h
// 1: é      <- index jumps to 3 next, because é was two bytes
// 3: l
// 4: l
// 5: o
```

> **IF YOU KNOW JAVASCRIPT**
> JavaScript strings are UTF-16 and have the same class of problem: `"👍".length` is `2`. Both languages make you think about encoding the moment you leave ASCII. Go is at least explicit that you are holding bytes.

Strings are immutable, so building one in a loop with `+=` allocates a new string every time. For anything beyond a few concatenations use a builder:

```go
var b strings.Builder
for _, word := range words {
    b.WriteString(word)
    b.WriteString(" ")
}
result := b.String()
```

Control flow, briefly

`if` and `for` are almost familiar, with two differences worth stating.

There are no parentheses around the condition, and braces are mandatory even for one line.

```
if count > 10 {
    fmt.Println("plenty")
} else if count > 0 {
    fmt.Println("some")
} else {
    fmt.Println("none")
}
```

`for` is the only loop keyword. It covers every case JavaScript spreads across `for`, `while` and `for...of`.

```
for i := 0; i < 10; i++ { }      // classic
for count < 10 { }               // while
for { }                          // infinite
for i, v := range items { }      // for...of, with the index
```

An `if` can carry a statement before the condition, scoped to the `if`. You will see this constantly in real code:

```
if err := doSomething(); err != nil {
    return err
}
// err does not exist here, which is the point
```

IF YOU KNOW JAVASCRIPT

`switch` in Go needs no `break`: cases do not fall through. You can switch on no value at all, which makes it a tidy replacement for a long `else if` chain:

```
switch {
case score > 90:
    grade = "A"
case score > 80:
    grade = "B"
default:
    grade = "C"
}
```

PROJECT 2: A TEMPERATURE TABLE

Write a program that prints a table converting Celsius to Fahrenheit from minus 40 to 100 in steps of 20. The formula is `f = c*9/5 + 32`.

```
package main

import "fmt"

func main() {
    fmt.Printf("%8s %8s\n", "C", "F")
    for c := -40; c <= 100; c += 20 {
        f := float64(c)*9/5 + 32
        fmt.Printf("%8d %8.1f\n", c, f)
    }
}
```

Two things to notice. `float64(c)` is required: without it you get integer division and wrong answers for anything not divisible by five, which is the bug this chapter exists to prevent. And `%8.1f` means "right align in eight columns, one decimal place", which is how Go does string formatting.

Now break it on purpose: remove the `float64(...)` conversion and read the error. Then change it to `c*9/5` with `c` still an `int` and see that it compiles but produces wrong numbers. The compiler catches the type error. It cannot catch the maths error, and knowing which is which is the skill.

YOUR TURN

Add a third column showing Kelvin (`k = c + 273.15`), and right align the header over it. Then make the step size a variable at the top of `main` and confirm the table still lines up when you change it to 10.

ASSIGNMENT

Make the zero values concrete, because `undefined` is the habit that takes longest to drop.

In `part1-language/03-types` , write a program that declares a variable of every type in this chapter's table, with no initial value, and prints each one with its type:

```
fmt.Printf("%-10T %q\n", s, s)
```

One. Print the zero value of a string, an int, a float64, a bool, a slice, a map, a pointer and a struct. Note in a comment which of them you can use immediately and which panic if you write to them.

Two. Write three lines that each fail to compile, and keep them commented out with the exact error: an int assigned to a float64, a string compared to a number, and an `if` on a non-boolean.

Three. In `NOTES.md`, answer in two sentences: which of these would JavaScript have accepted, and what would it have done instead?

COMMIT AS

```
git commit -m "Variables - Assignment One"
```

ASSIGNMENT

Strings are bytes, and the day that matters is the day somebody types their own name into your form.

In `part1-language/03-types/strings`:

One. Take a string with characters outside ASCII in it, your own name or a word with an accent on it, and print `len(s)` next to `utf8.RuneCountInString(s)`. Explain the difference in a comment.

Two. Index it with `s[0]` and print what comes back. It is a number, and the type is the explanation.

Three. Loop over it twice, once with an index loop over bytes and once with `range`, printing each item. Only one of them produces characters.

Four. Reverse the string, correctly. The byte-wise reversal that works for `"hello"` will produce nonsense for your test string, and both versions belong in the commit.

Five. In `NOTES.md`, write the one sentence you would put in a code review when somebody slices a string to truncate it for display.

COMMIT AS

```
git commit -m "Variables - Assignment Two"
```

4

Functions, multiple returns, and the end of try/catch

This is the chapter that decides whether you get on with Go. Error handling is where the language is most unlike JavaScript, it is the thing newcomers complain about loudest, and the complaint is half right. Let us do the mechanics first and the argument second.

> **THINGS YOU HAVE NOT MET YET**
> Code here uses a few things this book has not covered yet: structs in chapter 5, slices in chapter 6, interfaces in chapter 10, errors.Is in chapter 11 and defer in chapter 12. You are not missing anything. Read them as the shapes they look like, and each one gets its own chapter.

Functions

```
func add(a int, b int) int {
    return a + b
}

func add(a, b int) int {   // same thing, shared type
    return a + b
}
```

The type comes **after** the name, which reads backwards for about a day and then stops mattering. The reason is that it makes complicated declarations read left to right instead of spiralling outwards, which anyone who has read a C function pointer declaration will appreciate.

Functions are values, so the things you do in JavaScript still work:

```
double := func(n int) int { return n * 2 }

func apply(nums []int, f func(int) int) []int {
    out := make([]int, len(nums))
    for i, n := range nums {
        out[i] = f(n)
```

```
    }
    return out
}

apply([]int{1, 2, 3}, double)  // [2 4 6]
```

> **IF YOU KNOW JAVASCRIPT**
> That `apply` is `Array.prototype.map`, written by hand. Go had no generics until 1.18 and still has no `map`, `filter` or `reduce` in the standard library for slices. You write the loop. Go's argument is that a three-line loop needs no explanation and a chain of five callbacks sometimes does. You are allowed to find this annoying.

Two return values, and why

A Go function can return several values, and this is not a curiosity: it is the foundation of the error system.

```
func divide(a, b float64) (float64, error) {
    if b == 0 {
        return 0, errors.New("divide by zero")
    }
    return a / b, nil
}
```

The caller gets both:

```
result, err := divide(10, 2)
if err != nil {
    fmt.Println("failed:", err)
    return
}
fmt.Println(result)
```

The convention is absolute, across the standard library and every package you will use: **the error is the last return value, and you check it immediately**.

> **IF YOU KNOW JAVASCRIPT**
> JavaScript has one return value, so a function that can fail either throws or returns something like `{ data, error }`. Go makes the second shape the language default. If you have written Go-style error handling in TypeScript with a `Result` type, or used Rust, you already have the mental model.

`error` is just an interface

There is nothing magic about errors. `error` is an interface with one method:

```
type error interface {
    Error() string
}
```

Anything with an `Error() string` method is an error. That means you can make your own, carrying whatever fields you want:

```
type ValidationError struct {
    Field   string
    Message string
}

func (e ValidationError) Error() string {
    return e.Field + ": " + e.Message
}

func validate(age int) error {
    if age < 0 {
        return ValidationError{
            Field: "age", Message: "must not be negative"}
    }
    return nil
}
```

Wrapping: adding context without losing the original

A bare error from deep in a call stack is useless on its own. `fmt.Errorf` with the `%w` verb wraps one error inside another, keeping both.

```
func loadConfig(path string) (*Config, error) {
    data, err := os.ReadFile(path)
    if err != nil {
        return nil, fmt.Errorf("loading config from %s: %w", path, err)
    }
    ...
}
```

The message becomes a trail that reads from the outside in:

```
loading config from /etc/app.json: open /etc/app.json:
  no such file or directory
```

Use `%w`, not `%v`. `%v` formats the error as text and throws the original away, and then both of the checks below stop working.

Two functions read a wrapped chain. `errors.Is` asks "is this error, anywhere in the chain, that specific one":

```
if errors.Is(err, os.ErrNotExist) {
    // the file was missing, however deeply that was wrapped
}
```

`errors.As` asks "is there an error of this type in the chain", and gives it to you with its fields intact:

```
var ve ValidationError
if errors.As(err, &ve) {
    fmt.Println("bad field:", ve.Field)
}
```

WATCH OUT

`err != nil` is the only check that matters, and comparing an error to `nil` has one notorious trap. A function returning a **concrete** error type rather than the `error` interface can produce an error value that is not `nil` even when it holds nothing:

```
func bad() error {
    var e *ValidationError   // nil pointer
    // Not nil: a non-nil interface holding a nil pointer.
    return e
}
```

The fix is a rule: **always declare the return type as `error`, and return a literal `nil`, never a typed nil variable.** This catches experienced people and it is worth reading twice.

The argument about verbosity

Here is the complaint, stated fairly. This:

JAVASCRIPT

```
async function setup() {
  try {
    const cfg = await loadConfig();
    const db = await connect(cfg.dsn);
    await migrate(db);
    return db;
  } catch (e) {
    throw new Error(`setup failed: ${e.message}`);
  }
}
```

GO

```go
func setup() (*DB, error) {
    cfg, err := loadConfig()
    if err != nil {
        return nil, fmt.Errorf("setup: %w", err)
    }
    db, err := connect(cfg.DSN)
    if err != nil {
        return nil, fmt.Errorf("setup: %w", err)
    }
    if err := migrate(db); err != nil {
        return nil, fmt.Errorf("setup: %w", err)
    }
    return db, nil
}
```

The Go is twice as long. That is not a matter of opinion.

What you get for it is that every failure point is visible at the point it can fail. In the JavaScript, which of the three awaited calls threw? You cannot tell from the code, and the catch block treats all three identically whether or not that is what you wanted. In the Go, each failure is a separate decision, and you could handle the migration failure differently from the connection failure by changing one block.

The honest summary: `try` / `catch` is better when you want to treat a whole region of code as one unit that either works or does not. Explicit returns are better when failures deserve individual handling. Most real backend code is the second kind more often than people expect, and Go bet the language on it.

> **WHAT YOU LOSE, AND SHOULD KNOW ABOUT**
> In JavaScript an uncaught exception unwinds to the nearest handler, so a forgotten `catch` still surfaces somewhere. In Go a forgotten error check silently continues with a zero value. Nothing warns you.
>
> This is a real hazard and the tooling answers it: `errcheck`, bundled into `golangci-lint`, fails the build on an unchecked error. Install it early. It is the one linter that is not optional.

Named returns and `defer`

A function can name its return values, which is occasionally useful and frequently overused:

```go
func split(sum int) (x, y int) {
    x = sum * 4 / 9
    y = sum - x
```

```
    return          // a "naked" return, sending back x and y
}
```

Use naked returns only in short functions. In a long one, a bare `return` forces the reader to scroll up to learn what is being returned.

Named returns do earn their place with `defer`, which schedules a call to run when the surrounding function exits, no matter how it exits:

```
func process(path string) (err error) {
    f, err := os.Open(path)
    if err != nil {
        return err
    }
    defer f.Close()     // runs on every return path below

    // ... many returns, all of which close the file
    return nil
}
```

`defer f.Close()` immediately after a successful open is one of the most common patterns in Go, and it is how the language avoids the cleanup bugs that `try` / `finally` exists to solve.

WATCH OUT

Deferred calls run in last-in, first-out order, and their **arguments** are evaluated immediately, not at exit:

```
for i := 0; i < 3; i++ {
    defer fmt.Println(i)   // prints 2, 1, 0
}
```

And a `defer` inside a loop does not run at the end of each iteration, it runs when the **function** returns. Opening a thousand files in a loop with `defer f.Close()` keeps all thousand open until the function ends. Put the body in its own function if you need per-iteration cleanup.

PROJECT 3: A WORD COUNTER THAT HANDLES FAILURE

Write a program that takes a filename, counts the words in it, and reports errors properly rather than crashing.

```
package main

import (
```

```
    "fmt"
    "os"
    "strings"
)

func countWords(path string) (int, error) {
    data, err := os.ReadFile(path)
    if err != nil {
        return 0, fmt.Errorf("counting words in %s: %w", path, err)
    }
    return len(strings.Fields(string(data))), nil
}

func main() {
    if len(os.Args) < 2 {
        fmt.Fprintln(os.Stderr, "usage: wc <file>")
        os.Exit(1)
    }
    n, err := countWords(os.Args[1])
    if err != nil {
        fmt.Fprintln(os.Stderr, "error:", err)
        os.Exit(1)
    }
    fmt.Printf("%d words\n", n)
}
```

Run it on a real file, then on one that does not exist, and read the wrapped message. Notice `os.Exit(1)` and writing to `os.Stderr`: a command line tool that reports failure on stdout with exit code zero is a tool that breaks every script that uses it.

`strings.Fields` splits on any run of whitespace, which is the right behaviour here and is worth remembering: it is not the same as `strings.Split(s, " ")`, which gives empty strings between consecutive spaces.

YOUR TURN

Add a check that reports a clearer message when the path is a directory rather than a file. `os.ReadFile` returns an error you can inspect; try `errors.Is(err, syscall.EISDIR)` on Unix, or simply call `os.Stat` first and check `info.IsDir()`, which is portable and is the version you should probably ship.

ASSIGNMENT

Write a small parser that fails well, which is most of what error handling is for.

In `part1-language/04-functions`, write `ParseRange(s string) (int, int, error)` that accepts `"10-20"` and returns the two numbers.

One. Return a sentinel for the shape being wrong:

```
var ErrNotARange = errors.New("not a range")
```

Two. When a side does not parse as a number, wrap the error from `strconv.Atoi` with `%w` and enough context to say which side failed.

Three. In `main`, call it with `"10-20"`, `"ten-20"`, `"10"` and `""`, and use `errors.Is` to tell the sentinel apart from a number failure. Print a different message for each.

Four. Make one function return a named result and set it in a `defer`, then write one sentence in `NOTES.md` on whether it read better or worse.

The website's **Functions & Errors** and **Sentinel Errors** exercises are the short version of this.

COMMIT AS

```
git commit -m "Functions - Assignment One"
```

5

Structs and methods, against objects and classes

A JavaScript object is a bag of keys you can add to at any time. A Go struct is a fixed set of named fields with fixed types, decided when you declare the type and never changing. Everything else in this chapter follows from that one difference.

> **THINGS YOU HAVE NOT MET YET**
> Two things appear before their chapters: the & in front of a struct is a pointer, which chapter 8 is about, and the slices in these examples come in chapter 6. Neither matters for understanding a struct.

Declaring and creating

```
type User struct {
    ID    int
    Name  string
    Email string
    Admin bool
}
```

There are four ways to make one, and you will see all four in real code.

```
// 1. Field names. Use this one. It survives a field being added.
u := User{ID: 1, Name: "Ada", Email: "ada@example.com"}

// 2. Positional. Fragile: adding a field breaks every call site.
u := User{1, "Ada", "ada@example.com", false}

// 3. Zero value, then assign.
var u User
u.Name = "Ada"

// 4. new(), giving a pointer to a zero value.
u := new(User)   // *User
```

Notice what is missing from the first example: `Admin` was not set, so it is `false`. There is no `undefined`. Every field always has a value, and a struct you create is immediately complete.

> **IF YOU KNOW JAVASCRIPT**
> The closest JavaScript equivalent is a class with a constructor that sets every property. The difference is that you cannot add a field later. `u.Age = 36` on a `User` with no `Age` field is a compile error, not a new property. This is the whole point: the shape of your data is checked, everywhere, for free.

Methods: functions with a receiver

Go has no classes. You attach behaviour to a type by declaring a function with a **receiver**, the parameter in parentheses before the name.

```
func (u User) DisplayName() string {
    if u.Name == "" {
        return u.Email
    }
    return u.Name
}

fmt.Println(u.DisplayName())
```

`u User` is the receiver. It behaves like a parameter, because that is exactly what it is: `u.DisplayName()` is sugar for passing `u` as the first argument.

> **IF YOU KNOW JAVASCRIPT**
> There is no `this`, and that removes an entire class of JavaScript bug. The receiver is a normal named parameter, bound at the call site, and it cannot be rebound by how the function was called. No `.bind(this)`, no arrow function workaround, no `const self = this`.

Value receivers and pointer receivers

This is the first thing in Go that genuinely trips people up, and it is worth getting right now rather than debugging later.

A **value receiver** gets a copy. Changes to it do not affect the original.

```
func (u User) Rename(name string) {
    u.Name = name       // modifies the copy, then throws it away
}
```

```
u := User{Name: "Ada"}
u.Rename("Grace")
fmt.Println(u.Name)     // still "Ada"
```

A **pointer receiver** gets the address, so changes stick.

```
func (u *User) Rename(name string) {
    u.Name = name       // modifies the original
}

u := User{Name: "Ada"}
u.Rename("Grace")
fmt.Println(u.Name)     // "Grace"
```

You do not write `(&u).Rename(...)`. Go inserts the address-of automatically when the method needs a pointer and you have an addressable value.

> **WATCH OUT**
> The rule in practice: **if a method modifies the receiver, it must be a pointer receiver.** A value receiver that assigns to a field compiles happily and silently does nothing, which is one of the few truly quiet bugs in Go.
>
> The second rule: **be consistent within a type.** If any method on `User` takes a pointer, give them all pointers. Mixing the two is legal and makes the rules about which calls are addressable surface at the worst moment.

A practical reason to prefer pointer receivers even when not mutating: a value receiver copies the whole struct on every call. For a struct with twenty fields called in a loop, that is real work for no benefit.

Composition instead of inheritance

Go has no `extends`. It has **embedding**, which looks similar and behaves differently in one important way.

```
type Timestamps struct {
    CreatedAt time.Time
    UpdatedAt time.Time
}

type Post struct {
    Timestamps          // embedded: no field name
    Title string
    Body  string
}
```

```
p := Post{Title: "Hello"}
p.CreatedAt = time.Now()        // promoted, as if declared on Post
fmt.Println(p.Timestamps.CreatedAt)  // the long way, same field
```

The embedded type's fields and methods are **promoted** to the outer type. If `Timestamps` had a method `Age()`, then `p.Age()` would work.

> **IF YOU KNOW JAVASCRIPT**
> This resembles `class Post extends Timestamps`, and the difference matters: there is no polymorphism here. A `Post` is **not** a `Timestamps` and cannot be passed where one is expected. Embedding is a convenience for forwarding fields and methods, not a type relationship. In Go, the thing that lets one type stand in for another is an interface, which is chapter ten.

Struct tags: metadata for encoders

A field can carry a string of metadata, used by libraries through reflection. You will meet this constantly, because it is how Go does JSON.

```
type User struct {
    ID    int    `json:"id"`
    Name  string `json:"name"`
    Email string `json:"email"`
    Admin bool   `json:"-"`                  // never encoded
    Bio   string `json:"bio,omitempty"`      // dropped when empty
}
```

Without tags, `json.Marshal` uses the Go field names, so you get `{"ID":1, "Name":"Ada"}`. With them you get the lowercase JSON your frontend expects.

> **WATCH OUT**
> Those backticks are a raw string literal, and the format inside is strict: `json:"name"` with no space after the colon. `json: "name"` silently does nothing, the tag is ignored, and the field encodes as `Name`. There is no error. If a field is coming out with the wrong key, check for that space first.

Constructors are just functions

Go has no `constructor` keyword. The convention is a package-level function named `New` or `NewThing` that returns the type.

```
func NewUser(name, email string) (*User, error) {
    if email == "" {
        return nil, errors.New("email is required")
    }
    return &User{
        Name:  name,
        Email: email,
    }, nil
}
```

Returning `*User` rather than `User` is the usual choice, because the caller almost always wants to pass it around without copying, and because it lets you return `nil` on failure.

> **IF YOU KNOW JAVASCRIPT**
> A JavaScript constructor cannot fail cleanly: it either throws or returns a half-built object. A Go constructor is an ordinary function, so it returns an error like everything else, and you cannot get a `*User` back unless it succeeded.

Comparing structs

Structs are comparable with `==` when all their fields are, and the comparison is by value, field by field.

```
a := User{ID: 1, Name: "Ada"}
b := User{ID: 1, Name: "Ada"}
fmt.Println(a == b)     // true
```

That is deep equality for free, and it is a genuine surprise if you are used to `a === b` on objects being an identity check.

> **WATCH OUT**
> It stops working the moment a struct contains a slice, a map or a function, because those are not comparable. The compiler tells you. For those, use `reflect.DeepEqual`, or in tests `google/go-cmp`, which prints a readable diff instead of just `false`.

> **PROJECT 4: A CONTACT BOOK IN MEMORY**
> Build a small type with behaviour, and exercise value and pointer receivers.
>
> ```
> package main
>
> import (
>     "errors"
>     "fmt"
> ```

```
    "strings"
)

type Contact struct {
    Name  string
    Email string
    Tags  []string
}

// Value receiver: reads only, so a copy is fine.
func (c Contact) Display() string {
    if len(c.Tags) == 0 {
        return c.Name
    }
    return fmt.Sprintf("%s [%s]", c.Name, strings.Join(c.Tags, ", "))
}

// Pointer receiver: it modifies, so it must be a pointer.
func (c *Contact) Tag(tag string) {
    for _, t := range c.Tags {
        if t == tag {
            return // already there, do nothing
        }
    }
    c.Tags = append(c.Tags, tag)
}

type Book struct {
    contacts []Contact
}

func (b *Book) Add(c Contact) error {
    if c.Email == "" {
        return errors.New("contact needs an email")
    }
    b.contacts = append(b.contacts, c)
    return nil
}

func (b *Book) Find(email string) (*Contact, bool) {
    for i := range b.contacts {
        if b.contacts[i].Email == email {
            return &b.contacts[i], true   // address of the real one
        }
    }
    return nil, false
}

func main() {
    var book Book
    ada := Contact{Name: "Ada", Email: "ada@example.com"}
```

```
    if err := book.Add(ada); err != nil {
        fmt.Println(err)
        return
    }

    if c, ok := book.Find("ada@example.com"); ok {
        c.Tag("maths")
        c.Tag("maths") // ignored, already present
    }

    for _, c := range book.contacts {
        fmt.Println(c.Display())  // Ada [maths]
    }
}
```

The subtle line is `return &b.contacts[i], true`. Ranging with `for _, c := range b.contacts` would give you a **copy** of each contact, and tagging that copy would change nothing. Indexing with `i` and taking the address reaches the real element. Change it to the copying version and watch the tag vanish: that is the lesson of this chapter in one experiment.

YOUR TURN

Add a `Remove(email string) bool` method to `Book`. You will need to build a new slice or use the `slices.Delete` helper from the standard library. Then add a `Count()` method and decide, with a reason you can say out loud, whether it takes a value or a pointer receiver.

ASSIGNMENT

Model something real, with the receiver choice made deliberately.

In `part1-language/05-structs`, build an `Invoice` with a `Customer`, a slice of `Line` items, and a `Status`.

One. A constructor `NewInvoice` that returns a usable zero-free value.

Two. `Total()` on a value receiver, and `AddLine()` on a pointer receiver. Then call `AddLine` on a value and show, in a comment, why the line vanished.

Three. Embed an `Audit` struct with `CreatedAt` and `UpdatedAt` and reach the fields directly, which is composition doing what inheritance was for.

Four. Add JSON tags, marshal one invoice, and commit the output in `sample.json`. Chapter nineteen will come back to those tags.

COMMIT AS

```
git commit -m "Structs - Assignment One"
```

6

Slices: the array that is not an array

Go has arrays and it has slices. You will use slices for essentially everything, and the distinction matters exactly twice: when you declare one, and when a surprising bug appears. This chapter covers both.

> **THINGS YOU HAVE NOT MET YET**
> One example counts words with a map. Maps are chapter 7; here it is just somewhere to put a count.

Arrays have a fixed length, in the type

```
var a [3]int        // an array of exactly three ints
a[0] = 1
fmt.Println(len(a)) // 3
```

The length is **part of the type**. `[3]int` and `[4]int` are different types, and a function taking `[3]int` will not accept `[4]int`. Arrays are also copied whole on assignment. This is almost never what you want, which is why arrays are rare in Go code.

Slices are a view onto an array

```
nums := []int{1, 2, 3}      // no length in the brackets: a slice
nums = append(nums, 4)
fmt.Println(len(nums))      // 4
```

A slice is a small three-field value: a pointer to an underlying array, a length, and a capacity. That is the whole model, and understanding it explains every slice surprise you will ever hit.

`len(s)`	How many elements you can index right now
`cap(s)`	How many fit before the underlying array must be reallocated

```
s := make([]int, 2, 5)   // len 2, cap 5
fmt.Println(len(s), cap(s))  // 2 5
```

```
s = append(s, 10)            // fits in the existing array
fmt.Println(len(s), cap(s))  // 3 5
```

`make([]T, len, cap)` creates one with room reserved. When you know roughly how many elements are coming, say so: it avoids repeated reallocation.

> **IF YOU KNOW JAVASCRIPT**
> A JavaScript array handles all of this invisibly. `push` grows it and you never think about capacity. Go exposes the machinery, which costs you some attention and buys you control over allocation, which matters a great deal in a hot loop and not at all in a handler that runs once per request.

append, and the thing it does not promise

`append` returns a slice. **You must use the return value.**

```
nums := []int{1, 2, 3}
append(nums, 4)          // compile error: result not used
nums = append(nums, 4)   // correct
```

The reason is that `append` may or may not allocate a new array. If there is spare capacity it writes in place and returns a slice with a longer length. If there is not, it allocates a bigger array, copies everything over, and returns a slice pointing at the **new** array. The caller's original slice still points at the old one.

This is where the famous slice bug lives.

```
a := []int{1, 2, 3, 4, 5}
b := a[1:3]              // [2 3], sharing a's array
b[0] = 99
fmt.Println(a)           // [1 99 3 4 5]  <- a changed
```

Slicing does not copy. `b` is a window onto the same memory. Writing through one is visible through the other.

> **WATCH OUT**
> It gets worse with `append`, because whether you corrupt the original depends on capacity, which depends on history:
>
> ```
> a := []int{1, 2, 3, 4, 5}
> b := a[1:3]              // len 2, cap 4 (to the end of a)
> b = append(b, 99)        // fits in cap, writes into a's array
> fmt.Println(a)           // [1 2 3 99 5]  <- clobbered a[3]
> ```

The fix, when you want a genuinely independent slice, is a three-index slice that caps capacity, or an explicit copy:

```
b := a[1:3:3]            // cap is now 2, so append must allocate
// or
b := slices.Clone(a[1:3])
```

This bug is rare in everyday handler code and vicious when it appears. The tell is "a value changed and nothing in my code assigned to it".

Iterating

```
for i, v := range nums {
    fmt.Println(i, v)
}

for _, v := range nums {     // index discarded
    fmt.Println(v)
}

for i := range nums {        // value discarded
    nums[i] *= 2
}
```

`range` gives you a **copy** of each element. For ints that is irrelevant. For structs it is the bug from the previous chapter: modifying `v` changes nothing. Use the index when you need to write.

IF YOU KNOW JAVASCRIPT
Go has no `map`, `filter`, `reduce`, `find` or `some` for slices in the language. Since Go 1.21 the `slices` package covers some of it:

```
slices.Contains(nums, 3)        // bool
slices.Index(nums, 3)           // position or -1
slices.Sort(nums)               // in place
slices.Reverse(nums)            // in place
slices.Max(nums)                // and Min
```

For anything else, write the loop. A filter is four lines:

```
var active []User
for _, u := range users {
    if u.Active {
        active = append(active, u)
```

```
    }
}
```

Passing slices to functions

A slice is cheap to pass: you are copying three words, not the data. That means a function **can** modify the caller's elements.

```
func zero(nums []int) {
    for i := range nums {
        nums[i] = 0
    }
}

nums := []int{1, 2, 3}
zero(nums)
fmt.Println(nums)   // [0 0 0]
```

But it cannot change the caller's **length**, because the slice header itself was copied:

```
func addOne(nums []int) {
    nums = append(nums, 1)   // changes the local copy's header only
}

nums := []int{1, 2, 3}
addOne(nums)
fmt.Println(len(nums))   // still 3
```

The convention, and the reason `append` works the way it does: a function that grows a slice returns it.

```
func addOne(nums []int) []int {
    return append(nums, 1)
}
nums = addOne(nums)
```

Sorting

```
people := []User{{Name: "Zoe"}, {Name: "Ada"}}

slices.SortFunc(people, func(a, b User) int {
    return strings.Compare(a.Name, b.Name)
})
```

The comparison function returns a negative number, zero, or a positive number, the same contract as JavaScript's `Array.prototype.sort` comparator. For stable sorting use `slices.SortStableFunc`.

PROJECT 5: WORD FREQUENCY, TOP TEN

Read a file, count how often each word appears, and print the ten most common. This uses slices, sorting and a map together, and it is a real tool you can point at a book.

```
package main

import (
    "fmt"
    "os"
    "slices"
    "strings"
)

type pair struct {
    word  string
    count int
}

func main() {
    if len(os.Args) < 2 {
        fmt.Fprintln(os.Stderr, "usage: freq <file>")
        os.Exit(1)
    }

    data, err := os.ReadFile(os.Args[1])
    if err != nil {
        fmt.Fprintln(os.Stderr, "error:", err)
        os.Exit(1)
    }

    counts := map[string]int{}
    for _, w := range strings.Fields(string(data)) {
        w = strings.ToLower(strings.Trim(w, ".,!?;:\"'()"))
        if w != "" {
            counts[w]++
        }
    }

    pairs := make([]pair, 0, len(counts))
    for w, c := range counts {
        pairs = append(pairs, pair{w, c})
    }

    slices.SortFunc(pairs, func(a, b pair) int {
        if b.count != a.count {
            return b.count - a.count       // count descending
        }
```

```
        return strings.Compare(a.word, b.word) // then alphabetical
    })

    for _, p := range pairs[:min(10, len(pairs))] {
        fmt.Printf("%6d  %s\n", p.count, p.word)
    }
}
```

Three things worth noticing. `make([]pair, 0, len(counts))` reserves capacity up front because we know exactly how many there will be. The tie-break on word makes the output deterministic, which matters because map iteration order is random and without it two runs would disagree. And `min` is a builtin since Go 1.21, so `pairs[:min(10, len(pairs))]` does not panic on a file with fewer than ten distinct words.

YOUR TURN

Add a flag so the user can ask for a different number of results, for example `freq book.txt 25`. Then add a stop-word list so "the", "a" and "and" are excluded, and decide whether a slice or a map is the better structure for holding those stop words. The next chapter will tell you whether you chose right.

ASSIGNMENT

Write the three helpers you miss from JavaScript, then prove the trap.

In `part1-language/06-slices`:

One. `Filter`, `Chunk` and `Dedupe` over `[]string`, each with a table-driven test. You will write these generically in chapter seventeen, so keep them.

Two. Reproduce the aliasing bug on purpose. Take a slice of a slice, append to it, and show the original changing. Then fix it with a copy, and write the one-line rule in a comment.

Three. Sort a slice of structs by two keys, falling back to the second when the first ties.

```
sort.Slice(xs, func(i, j int) bool { ... })
```

Four. In `NOTES.md`, state in one sentence what `append` does not promise.

COMMIT AS

```
git commit -m "Slices - Assignment One"
```

7

Maps, and why iteration order is random on purpose

A Go map is a hash table with typed keys and typed values. It is the closest thing Go has to a JavaScript object used as a dictionary, and it differs in four ways that each cause a specific bug the first time you meet them.

> **THINGS YOU HAVE NOT MET YET**
> The concurrency-safe example at the end uses sync.Mutex (chapter 15) and defer (chapter 12). Skip it on a first read if you like: the map itself is the chapter.

Declaring and using

```
ages := map[string]int{
    "ada":   36,
    "grace": 45,
}

ages["alan"] = 41
fmt.Println(ages["ada"])    // 36
fmt.Println(len(ages))      // 3
delete(ages, "alan")
```

The type is `map[KeyType]ValueType`. Keys must be comparable, so strings, ints, booleans and structs of those are fine; slices, maps and functions are not.

> **WATCH OUT**
> A `nil` map is readable but **not writable**, and this is the first map bug everyone hits:
>
> ```
> var m map[string]int      // nil
> fmt.Println(m["a"])       // 0, fine
> fmt.Println(len(m))       // 0, fine
> m["a"] = 1                // panic: assignment to entry in nil map
> ```

Always create one before writing: `m := map[string]int{}` or `m := make(map[string]int)`. The same trap hides inside structs, where a map field is nil until you assign one, so a constructor that forgets it produces a struct that panics on first write.

Missing keys return the zero value

Reading a key that is not there is not an error. You get the zero value of the value type.

```
counts := map[string]int{}
fmt.Println(counts["nothing"])   // 0, not undefined
counts["word"]++                 // works: reads 0, writes 1
```

That last line is why counting in Go is so tidy: no "if the key exists, add one, otherwise set it to one".

To tell "absent" from "present but zero", use the two-value form:

```
v, ok := ages["nobody"]
if !ok {
    fmt.Println("not in the map")
}
```

`ok` is a bool. This is the **comma ok** idiom, and you will see the same shape for type assertions and channel receives.

IF YOU KNOW JAVASCRIPT

In JavaScript, `obj.missing` gives `undefined`, and you distinguish absent from set-to-undefined with `in` or `hasOwnProperty`. Go's `v, ok :=` is the same idea with better ergonomics, because the check and the read are one operation and you cannot forget which you did.

Iteration order is deliberately random

```
for k, v := range ages {
    fmt.Println(k, v)
}
```

Run that twice and the order differs. This is not an implementation accident that might settle down: the runtime randomises the starting point on purpose, so that code which accidentally depends on order fails immediately and loudly rather than in production two years later when the table grows.

To iterate in a stable order, sort the keys:

```
keys := make([]string, 0, len(ages))
for k := range ages {
    keys = append(keys, k)
}
slices.Sort(keys)

for _, k := range keys {
    fmt.Println(k, ages[k])
}
```

> **IF YOU KNOW JAVASCRIPT**
> JavaScript objects do have an order: integer-like keys ascending, then string keys in insertion order. Plenty of code quietly relies on it. Go removes the temptation. If your output needs an order, you must say what that order is, which is usually an improvement to the code anyway.

Maps are reference types

Passing a map to a function does not copy it. The function can modify the caller's map.

```
func add(m map[string]int) {
    m["new"] = 1        // visible to the caller
}
```

This is unlike a struct and unlike an array, and like a slice's elements. There is no `const` map and no way to pass one read-only. If a function must not modify a map, the only guards are a copy (`maps.Clone`) or a comment.

Structs as values, and the thing you cannot do

A map of structs has one sharp edge:

```
type Stat struct{ Count int }
m := map[string]Stat{"a": {Count: 1}}

m["a"].Count++      // compile error: cannot assign to struct field
```

The map returns a **copy** of the struct, so incrementing it would modify something about to be discarded, and Go refuses rather than letting you write code that does nothing. Two fixes:

```
// 1. Read, modify, write back.
s := m["a"]
s.Count++
m["a"] = s
```

```
// 2. Store pointers instead.
m := map[string]*Stat{"a": {Count: 1}}
m["a"].Count++      // fine
```

The second is usually what you want when the values are large or frequently updated. The first is fine for small structs and avoids a layer of indirection.

Sets

Go has no set type. The idiom is a map with an empty struct value:

```
seen := map[string]struct{}{}
seen["ada"] = struct{}{}

if _, ok := seen["ada"]; ok {
    fmt.Println("already seen")
}
```

`struct{}` is a struct with no fields and it occupies zero bytes, so the map stores only keys. Using `map[string]bool` is also common and slightly more readable at the cost of one byte per entry; both are accepted Go.

> **IF YOU KNOW JAVASCRIPT**
> JavaScript has a real `Set` with `.has()` and `.add()`. Go's version is wordier, and that `struct{}{}` reads as line noise until you have seen it a dozen times. Read it as: the type `struct{}`, then an empty literal of it.

Maps are not safe for concurrent use

Two goroutines writing to the same map at the same time will crash the program with a runtime error, deliberately. Go detects it and stops rather than corrupting the table silently.

The answer is usually a mutex, which is chapter fifteen:

```
type Counter struct {
    mu     sync.Mutex
    counts map[string]int
}

func (c *Counter) Inc(key string) {
    c.mu.Lock()
    defer c.mu.Unlock()
    c.counts[key]++
}
```

IF YOU KNOW JAVASCRIPT

This has no JavaScript equivalent, because the event loop means two pieces of your code never run at literally the same instant. In Go they do. Any value shared between goroutines needs protection, and a map is the one where forgetting is immediately fatal rather than subtly wrong, which is mercifully the better failure.

PROJECT 6: A WORD INDEX WITH LINE NUMBERS

Extend the frequency counter into something genuinely useful: for each word, record every line it appears on. This is a `map[string][]int`, a map whose values are slices, and it is a shape you will use often.

```
package main

import (
    "bufio"
    "fmt"
    "os"
    "slices"
    "strings"
)

func main() {
    if len(os.Args) < 2 {
        fmt.Fprintln(os.Stderr, "usage: index <file>")
        os.Exit(1)
    }

    f, err := os.Open(os.Args[1])
    if err != nil {
        fmt.Fprintln(os.Stderr, "error:", err)
        os.Exit(1)
    }
    defer f.Close()

    index := map[string][]int{}

    scanner := bufio.NewScanner(f)
    line := 0
    for scanner.Scan() {
        line++
        for _, w := range strings.Fields(scanner.Text()) {
            w = strings.ToLower(strings.Trim(w, ".,!?;:\"'()"))
            if len(w) < 4 {
                continue    // skip short words
            }
            // Appending to a nil slice works, so no "if absent" branch.
            if !slices.Contains(index[w], line) {
                index[w] = append(index[w], line)
```

```
            }
        }
    }
    if err := scanner.Err(); err != nil {
        fmt.Fprintln(os.Stderr, "reading:", err)
        os.Exit(1)
    }

    words := make([]string, 0, len(index))
    for w := range index {
        words = append(words, w)
    }
    slices.Sort(words)

    for _, w := range words {
        if len(index[w]) > 2 {   // only words on three or more lines
            fmt.Printf("%-16s %v\n", w, index[w])
        }
    }
}
```

The line that rewards attention is `index[w] = append(index[w], line)`. If `w` is not in the map, `index[w]` is a nil slice, and appending to a nil slice is legal and returns a one-element slice. So the "create the entry if missing" branch that you would write in most languages simply does not exist here. Two zero-value rules, slices and maps, combining to delete a line of code.

Also note `bufio.Scanner`: it reads line by line without loading the file into memory, so this works on a file larger than your RAM, unlike the `os.ReadFile` in the last project. And `scanner.Err()` must be checked, because the loop ends identically on end-of-file and on a read error.

YOUR TURN

Change the value type to a small struct holding both the line numbers and a total count, and watch the compiler stop you from writing `index[w].Count++`. Fix it both ways described above, then decide which you prefer and why. That decision, made consciously once, is the difference between copying Go idioms and understanding them.

ASSIGNMENT

A word counter, which is the smallest program that forces every map lesson.

In `part1-language/07-maps`, read a text file and report the ten most common words.

One. Count with a `map[string]int`, using the fact that a missing key reads as zero rather than checking first.

Two. Sort the results before printing, and say in a comment why printing straight from the map would make the output differ between runs.

Three. Group with a `map[string][]string`: words by first letter. Note what happens when you append to a key that does not exist yet.

Four. Try to assign to a field of a struct stored in a map, keep the compiler error in a comment, and fix it the way the chapter says.

The website's **Grouping With a Map of Slices** is the same shape, smaller.

COMMIT AS

```
git commit -m "Maps - Assignment One"
```

8

Pointers, without the fear

Pointers have a reputation they do not deserve, earned in languages where you can do arithmetic on them and walk off the end of memory. Go's pointers cannot do that. There is no pointer arithmetic, and the garbage collector means you never free anything. What is left is a small, useful idea: **a pointer is the address of a value rather than a copy of it**.

You have already been using them. This chapter names what was happening.

> **THINGS YOU HAVE NOT MET YET**
> A couple of examples borrow from later chapters: errors.Is is chapter 11, goroutines are chapter 13, and the json tags are chapter 19. They are there because pointers are what those chapters trip over.

Everything in Go is passed by value

This is the single sentence that explains pointers, and it is worth reading twice because it is the opposite of your JavaScript instinct.

When you pass anything to a function, Go copies it. An int, a struct, a slice header, all copied. So a function cannot change what you passed:

```
func rename(u User) {
    u.Name = "changed"      // changes the copy
}

u := User{Name: "Ada"}
rename(u)
fmt.Println(u.Name)         // "Ada"
```

> **IF YOU KNOW JAVASCRIPT**
> In JavaScript, objects are passed by **reference to the same object**, so a function that sets `user.name` changes the caller's object. Primitives are copied. Go removes the distinction: everything is copied, every time, and when you want the other behaviour you ask for it explicitly with a pointer. Less to remember, more to type.

Two operators

```
u := User{Name: "Ada"}

p := &u        // & takes the address: p is a *User
fmt.Println(*p)     // * dereferences: the User itself
fmt.Println(p.Name) // but you rarely need *, see below
```

`&x` means "the address of x". `*T` in a type means "pointer to T". `*p` as an expression means "the value at p".

Now the function that works:

```
func rename(u *User, name string) {
    u.Name = name
}

rename(&u, "Grace")
fmt.Println(u.Name)     // "Grace"
```

> **WATCH OUT**
> You almost never write `(*u).Name`. Go dereferences automatically for field access and method calls, so `u.Name` works whether `u` is a `User` or a `*User`. This is a deliberate convenience and it means pointer-ness mostly disappears from the code that uses the value. The place it reappears is `nil`.

nil, and the panic you will meet

A pointer's zero value is `nil`. Using a field on a nil pointer crashes:

```
var u *User          // nil
fmt.Println(u.Name)  // panic: runtime error: invalid memory address
                     // or nil pointer dereference
```

This is Go's `Cannot read properties of undefined`, with the same cause and a louder failure: the program stops rather than continuing with a bad value.

The guard is an ordinary `if`:

```
func display(u *User) string {
    if u == nil {
        return "(none)"
    }
    return u.Name
}
```

> **IF YOU KNOW JAVASCRIPT**
> TypeScript's `strictNullChecks` would make this a compile error. Go has no equivalent: a `*User` might be nil and the compiler will not stop you. This is the one area where TypeScript's type system is genuinely ahead, and it is worth saying rather than glossing over. Your defences in Go are: return values rather than pointers where you can, check at the boundary, and let the panic happen in development rather than coding defensively everywhere.

When to use a pointer

Three reasons, and nothing else.

One: you need to modify the caller's value. This is the common case, and it is why methods that change state take pointer receivers.

Two: the struct is large and copied often. A struct with twenty fields copied in a tight loop is real work. For a three-field struct it does not matter, and chasing this before measuring is how code gets uglier for no gain.

Three: nil is a meaningful value. A `*string` field distinguishes "the user did not supply this" from "the user supplied an empty string". You met this in chapter three as the limit of zero values. This is the fix.

```
type UpdateUser struct {
    Name  *string `json:"name"`   // nil means "leave it alone"
    Email *string `json:"email"`
}

func apply(u *User, patch UpdateUser) {
    if patch.Name != nil {
        u.Name = *patch.Name       // explicitly set, even to ""
    }
    if patch.Email != nil {
        u.Email = *patch.Email
    }
}
```

That is how a PATCH endpoint tells "clear this field" apart from "do not touch this field", and you will write it for real in part four.

When not to use a pointer

Slices, maps and channels already hold an internal pointer. Passing `*[]int` or `*map[string]int` is almost always wrong and marks code as written by someone still finding their feet.

```
func addItem(items []string) []string {   // correct
    return append(items, "new")
}

func addItem(items *[]string) {           // almost always wrong
    *items = append(*items, "new")
}
```

The slice version returns the new slice, which is the convention `append` itself follows. The pointer version works and reads badly.

Small structs you only read are also better by value. A `Point{X, Y int}` is sixteen bytes; copying it is cheaper than following a pointer.

The loop variable, and a trap that Go fixed

For years this was the most common Go bug in existence:

```
// Go 1.21 and earlier
for _, u := range users {
    go func() {
        fmt.Println(u.Name)   // every goroutine saw the SAME u
    }()
}
```

The loop variable was reused across iterations, so all the goroutines shared one variable and usually printed the last element several times.

Go 1.22 changed this. Loop variables are now per-iteration, and the code above does what it looks like it does.

> **WATCH OUT**
> You will still find the old workaround everywhere in blog posts and older codebases:
>
> ```
> for _, u := range users {
>     u := u              // "shadow" it, making a per-iteration copy
>     go func() { ... }()
> }
> ```
>
> It is harmless on modern Go and no longer needed. But check your `go.mod`: the new behaviour only applies if the `go` directive says 1.22 or higher. A module that still says `go 1.21` gets the old semantics **even on a new toolchain**, which is a subtle and genuinely nasty way to inherit this bug.

Pointers to struct fields

You can take the address of a field, and of a slice element, which is how the contact book in chapter five worked:

```
for i := range book.contacts {
    if book.contacts[i].Email == email {
        return &book.contacts[i]    // address of the real element
    }
}
```

What you cannot take the address of is a map entry, because the map may move its contents when it grows:

```
m := map[string]User{"a": {Name: "Ada"}}
p := &m["a"]        // compile error: cannot take address of m["a"]
```

This is the same restriction that stopped `m["a"].Name++` in chapter seven, and it has the same two fixes: read-modify-write, or store `*User` in the map.

PROJECT 7: AN INVENTORY WITH ADJUSTABLE STOCK

Build something where the pointer decisions actually matter, and where getting one wrong is visible.

```
package main

import (
    "errors"
    "fmt"
)

type Item struct {
    SKU   string
    Name  string
    Stock int
}

type Inventory struct {
    items map[string]*Item    // pointers: we mutate these in place
}

func NewInventory() *Inventory {
    return &Inventory{items: map[string]*Item{}}
}

func (inv *Inventory) Add(sku, name string, stock int) {
    inv.items[sku] = &Item{SKU: sku, Name: name, Stock: stock}
}
```

```
// Get returns nil when absent, so callers must check. The bool makes
// that
// impossible to forget by accident.
func (inv *Inventory) Get(sku string) (*Item, bool) {
    it, ok := inv.items[sku]
    return it, ok
}

var ErrInsufficientStock = errors.New("insufficient stock")

func (inv *Inventory) Take(sku string, n int) error {
    it, ok := inv.items[sku]
    if !ok {
        return fmt.Errorf("taking %s: not found", sku)
    }
    if it.Stock < n {
        return fmt.Errorf("taking %d of %s: %w",
            n, sku, ErrInsufficientStock)
    }
    it.Stock -= n       // through the pointer, so the map sees it
    return nil
}

func main() {
    inv := NewInventory()
    inv.Add("A1", "Blue widget", 10)

    if err := inv.Take("A1", 3); err != nil {
        fmt.Println(err)
    }
    if err := inv.Take("A1", 99); err != nil {
        fmt.Println("expected:", err)
        fmt.Println("is stock error:",
            errors.Is(err, ErrInsufficientStock))
    }

    if it, ok := inv.Get("A1"); ok {
        fmt.Printf("%s: %d left\n", it.Name, it.Stock)   // 7 left
    }
}
```

Now do the experiment that teaches the chapter. Change the map to `map[string]Item`, holding values rather than pointers, and try to compile. The line `it.Stock -= n` will still compile, because `it` is now a copy, and the stock will never change. Run it and watch `10 left` print after you took three. Then make it work by reading, modifying and writing back.

That bug, with values where you meant pointers, is the single most common mistake newcomers make with Go collections, and having caused it once on purpose you will recognise it instantly.

YOUR TURN

Add a `Restock(sku string, n int) error` method, then add a `Low(threshold int) []*Item` that returns every item below a stock level. Decide whether `Low` should return `[]*Item` or `[]Item` and be able to say why. One of them lets a caller change your inventory by accident, which might be exactly what you want or exactly what you do not.

ASSIGNMENT

Three programs, each one line different.

In `part1-language/08-pointers`:

One. A `Counter` with `Inc()` on a value receiver. Call it ten times and print zero. Then change one character and print ten. Commit both, the broken one as `value_receiver.go.txt` so it does not compile into the build.

Two. A function taking a struct by value and another taking a pointer. Print the addresses with `%p` to show what was copied.

Three. Cause a nil pointer panic deliberately, read the stack trace, and write in `NOTES.md` which line of it actually told you where the bug was.

Four. Write a function returning `*User` that returns `nil` for not found, and a caller that checks before dereferencing. That shape is everywhere in Part III.

COMMIT AS

```
git commit -m "Pointers - Assignment One"
```

9

Packages, modules and imports, against npm and ESM

You have been writing single-file programs. Real projects are many files, and Go's rules for organising them are stricter and simpler than what you are used to. This chapter is the one that makes a Go repository stop looking strange.

> **THINGS YOU HAVE NOT MET YET**
> errors.Is appears in one example and is properly chapter 11.

A package is a directory

Every `.go` file belongs to a package, declared on its first line. **Every file in a directory must declare the same package name**, and a directory is exactly one package. That is the whole rule.

```
myapp/
├── go.mod              module github.com/you/myapp
├── main.go             package main
├── store/
│   ├── store.go        package store
│   └── user.go         package store
└── money/
    └── money.go        package money
```

Files in `store/` can see each other's identifiers with no import at all, even the lowercase ones. There is no equivalent of importing between files in the same folder, because they are already the same package.

> **IF YOU KNOW JAVASCRIPT**
> This is the biggest structural difference from JavaScript. In a Next.js project every file is its own module, and you import between them constantly. In Go, `store/store.go` and `store/user.go` share a namespace: a function declared in one is callable from the other with no ceremony. You import across **directories**, never within one.

Importing

```
package main

import (
    "fmt"                // standard library
    "net/http"

    "github.com/google/uuid"          // third party

    "github.com/you/myapp/store"      // your own
)

func main() {
    s := store.New()
    fmt.Println(uuid.NewString())
}
```

You refer to things by the **package name**, which is the last element of the path by convention but is really whatever the files declare. `store.New`, not `New`.

The grouping above is conventional: standard library first, then third party, then your own module, separated by blank lines. `gofmt` sorts within groups and `goimports` adds and removes imports as you type.

> **WATCH OUT**
> An unused import is a compile error, exactly like an unused variable. This is why `goimports` in your editor is not optional: without it you will spend real time adding and removing import lines by hand.
>
> The escape hatch, for a package imported only for its side effects, is the blank identifier:
>
> ```
> import _ "github.com/lib/pq"   // registers a database driver
> ```

The import path is the module path plus the directory

There is no `baseUrl`, no `paths` mapping, no `@/components` alias, and no relative imports. `import "./store"` is not valid Go.

If your module is `github.com/you/myapp`, then the directory `store/` is imported as `github.com/you/myapp/store`. Always the full path.

> **IF YOU KNOW JAVASCRIPT**
> Verbose compared to `@/store`, and it buys one thing: an import path is globally unambiguous. You can paste it into a browser and often land on the source. There is no

build-time configuration that has to agree with your editor, your bundler and your test runner, because there is no mapping at all.

internal/ is a real access rule

A directory named `internal` is enforced by the compiler: only code rooted at its parent may import it.

```
myapp/
├── internal/
│   └── db/            importable by myapp and anything under it
└── pkg/
    └── client/        importable by anybody
```

If someone imports `github.com/you/myapp/internal/db` from a different module, the build fails. This is how a library publishes a small surface while keeping the rest private, and it is why nearly every serious Go project puts most of its code under `internal/`.

IF YOU KNOW JAVASCRIPT
The nearest npm equivalent is the `exports` field in `package.json`, which controls what consumers can reach. Go's version is a directory name, needs no configuration and cannot be worked around.

Naming, and the stutter rule

Package names are short, lowercase, single words. No underscores, no camelCase, no plurals.

The identifier is read **with** the package name, which leads to the most quoted style rule in Go:

```
// Bad: the package name is already there.
store.NewStore()
http.HTTPServer
money.MoneyAmount

// Good:
store.New()
http.Server
money.Amount
```

WATCH OUT

Avoid `util`, `helpers`, `common` and `shared`. They are not packages, they are places things go to hide, and they grow into a dependency that everything imports and nobody understands. If a function does not belong anywhere, that usually means the thing it operates on wants a package of its own.

Modules, versions and go.mod

A module is a tree of packages with one `go.mod` at the root. You saw it in chapter two; here is what the rest of it means.

```
module github.com/you/myapp

go 1.25

require (
    github.com/gin-gonic/gin v1.10.0
    gorm.io/gorm v1.25.12
)

require (
    github.com/bytedance/sonic v1.12.0 // indirect
)
```

The second `require` block is indirect dependencies: things your dependencies need. `go mod tidy` manages both blocks, adding what is imported and removing what is not. Run it before every commit.

IF YOU KNOW JAVASCRIPT

`go.sum` is the lockfile and must be committed. Unlike `package-lock.json` it is not a full dependency tree: it is a list of cryptographic hashes. Go uses **minimal version selection**, meaning it picks the lowest version that satisfies everyone, so builds are reproducible from `go.mod` alone and the sum file exists to verify integrity rather than to pin.

There is also no `devDependencies`. A tool you use at development time is either run with `go run` or tracked in a `tools.go` file.

Upgrading is explicit:

```
go get -u ./...          # upgrade everything (minor and patch)
go get github.com/gin-gonic/gin@v1.10.1   # one package, one version
go mod tidy
```

Major versions are in the path

This surprises everyone. From v2 onward, the major version is part of the import path:

```
import "github.com/you/myapp/v2/store"
```

It looks strange and it solves a real problem: two major versions of the same library can coexist in one build, because as far as the compiler is concerned they are different packages. You will see this in Grit itself, which imports as `github.com/MUKE-coder/grit/v3`.

init, and why to avoid it

A package can declare `func init()`, which runs automatically before `main`.

```
func init() {
    // runs once, before main, in no order you control across packages
}
```

Use it rarely. It makes startup order implicit, it cannot return an error, and it runs even when a test only wanted one function from the package. Explicit setup called from `main` is almost always better.

PROJECT 8: SPLIT THE INVENTORY INTO PACKAGES

Take the inventory from chapter seven and give it a real project layout. The point is to feel the import rules rather than to add features.

```
inventory/
├── go.mod                  module github.com/you/inventory
├── main.go                 package main
└── internal/
    └── inventory/
        ├── item.go         package inventory
        └── store.go        package inventory
```

`internal/inventory/item.go`:

```
package inventory

import "errors"

var ErrInsufficientStock = errors.New("insufficient stock")

type Item struct {
    SKU   string
    Name  string
    Stock int
}
```

```
// low is unexported, so only this package can call it. item.go and
// store.go
// are the same package, so store.go can use it with no import.
func (i *Item) low(threshold int) bool {
    return i.Stock < threshold
}
```

`internal/inventory/store.go`:

```
package inventory

import "fmt"

type Store struct {
    items map[string]*Item
}

func New() *Store {
    return &Store{items: map[string]*Item{}}
}

func (s *Store) Add(sku, name string, stock int) {
    s.items[sku] = &Item{SKU: sku, Name: name, Stock: stock}
}

func (s *Store) Take(sku string, n int) error {
    it, ok := s.items[sku]
    if !ok {
        return fmt.Errorf("taking %s: not found", sku)
    }
    if it.Stock < n {
        return fmt.Errorf("taking %d of %s: %w",
            n, sku, ErrInsufficientStock)
    }
    it.Stock -= n
    return nil
}

// Low uses the unexported method from the other file. No import needed.
func (s *Store) Low(threshold int) []*Item {
    var out []*Item
    for _, it := range s.items {
        if it.low(threshold) {
            out = append(out, it)
        }
    }
    return out
}
```

`main.go`:

```go
package main

import (
    "errors"
    "fmt"

    "github.com/you/inventory/internal/inventory"
)

func main() {
    inv := inventory.New()
    inv.Add("A1", "Blue widget", 10)

    if err := inv.Take("A1", 99); err != nil {
        // true
        fmt.Println(errors.Is(err, inventory.ErrInsufficientStock))
    }

    for _, it := range inv.Low(20) {
        fmt.Println("low:", it.Name, it.Stock)
    }
}
```

Now run three experiments. Call `inv.low(5)` from `main.go` and read the error: lowercase means package-private, and the compiler enforces it. Move the whole `internal` directory to `pkg` and confirm it still builds, then put it back. And try `import "./internal/inventory"` to see that relative imports genuinely do not exist.

YOUR TURN

Add a second package, `internal/report`, with a function that takes a `*inventory.Store` and prints a summary. You will hit a design question immediately: `report` needs to read every item, but `Store.items` is unexported. Solve it by adding an exported method to `inventory` rather than by exporting the field, and notice that the package boundary just forced you into a better API than you would have written in one file.

ASSIGNMENT

Take one of your earlier programs apart.

Copy your chapter seven word counter into `part1-language/09-packages` and split it.

One. `internal/count` holds the counting, `internal/report` holds the printing, and `main` wires them. Nothing in `internal/` may import `main`.

Two. Name the exported functions so the call site reads well: `count.Words(r)` rather than `count.CountWords(r)`.

Three. Try to import your `internal/count` from a second module on disk, and keep the error in `NOTES.md`. That rule is enforced by the compiler rather than by review, which is the whole point.

Four. Add one `init()` and then remove it, writing one sentence on what you replaced it with.

COMMIT AS

```
git commit -m "Packages - Assignment One"
```

10

Interfaces, and why Go's point the other way

If you take one idea from this book, take this one. Go's interfaces look like TypeScript's and work backwards from them, and once that clicks a great deal of Go code stops looking odd and starts looking inevitable.

> **THINGS YOU HAVE NOT MET YET**
> Two examples look ahead: context.Context is chapter 16 and the test is chapter 18. Interfaces are what make both of them work, which is why they are here rather than there.

The mechanical part

An interface is a set of method signatures.

```
type Shape interface {
    Area() float64
    Perimeter() float64
}
```

Any type with those two methods **is** a `Shape`. You do not say so anywhere.

```
type Rect struct{ W, H float64 }

func (r Rect) Area() float64      { return r.W * r.H }
func (r Rect) Perimeter() float64 { return 2 * (r.W + r.H) }

// Rect is now a Shape. Nothing was declared. Nothing imports Shape.
var s Shape = Rect{W: 3, H: 4}
```

This is **structural typing**, and it is called implicit interface satisfaction.

> **IF YOU KNOW JAVASCRIPT**
> TypeScript interfaces are structural too, so this much is familiar: an object with the right shape satisfies the interface. The difference is where the interface **lives** and who declares it, which is the next section and is the part that actually matters.

The direction: interfaces belong to the consumer

In Java or C#, the implementer declares what it implements: `class Rect implements Shape`. The interface is defined near the thing that provides it, and the provider must import it.

In Go, the convention is the reverse. **The interface is declared by the package that consumes it, and lists only what that consumer needs.**

Consider a function that writes a report somewhere.

```
// In package report. It does not import os, or a logging library, or a
// storage SDK. It says what it needs and nothing more.
type Writer interface {
    Write(p []byte) (int, error)
}

func Generate(w Writer, data []Row) error {
    for _, r := range data {
        _, err := fmt.Fprintf(w, "%s,%d\n", r.Name, r.Count)
        if err != nil {
            return err
        }
    }
    return nil
}
```

Now `Generate` accepts a file, a network connection, an HTTP response, an in-memory buffer, a gzip stream, or anything else with a `Write` method. None of those types know `report` exists. The dependency points **from** the consumer **to** nothing at all.

> **IF YOU KNOW JAVASCRIPT**
> In TypeScript you would often import a type from the module that provides it, or define a shared `types.ts` that both sides import. Go's version removes the shared file: the consumer owns a small interface, the provider owns a concrete type, and neither imports the other. That is the decoupling, and it is why Go codebases have surprisingly few imports between their own packages.

Keep them small

That `Writer` above is the real one. `io.Writer` in the standard library is exactly those four words, and it is arguably the most successful interface ever designed:

```
type Writer interface {
    Write(p []byte) (int, error)
}
```

One method. Hundreds of types satisfy it. Everything composes.

The proverb is **the bigger the interface, the weaker the abstraction**. A one-method interface is satisfied by nearly anything and is easy to fake in a test. A ten-method interface is satisfied by one type and is a class declaration wearing a disguise.

WATCH OUT

The beginner mistake, carried in from other languages, is to define an interface for every struct up front:

```
// Do not do this.
type UserService interface {
    Create(...) error
    Update(...) error
    Delete(...) error
    List(...) ([]User, error)
    // ... nine more
}

type userService struct{}   // the only implementation, ever
```

One implementation and one interface that mirrors it exactly is pure ceremony: you have doubled the code and decoupled nothing. The Go advice is **accept interfaces, return structs**: return the concrete type from your constructor, and let whichever caller needs an abstraction declare the two methods it actually uses.

The empty interface, and `any`

An interface with no methods is satisfied by everything.

```
var x any = 42          // any is an alias for interface{}
x = "hello"
x = User{}
```

`any` is Go's `unknown`, not its `any`. You cannot do anything with the value until you establish what it is. The tool is a **type assertion**:

```
s, ok := x.(string)
if !ok {
    // x was not a string
}
```

Or a type switch, which is the readable form when several types are possible:

```
switch v := x.(type) {
case string:
    fmt.Println("string of length", len(v))
```

```
case int:
    fmt.Println("int doubled:", v*2)
case User:
    fmt.Println("user named", v.Name)
default:
    fmt.Println("something else")
}
```

WATCH OUT
The single-value assertion `s := x.(string)` **panics** if the type is wrong. The two-value form `s, ok := x.(string)` never panics. Use the two-value form unless you genuinely want to crash, which in a server you almost never do.

Reach for `any` rarely. It throws away everything the type system knows, and most uses of it in a new codebase are better served by an interface with one method or, since Go 1.18, by generics.

The nil interface trap

This one catches experienced people and is worth the page.

An interface value has two parts internally: a **type** and a **value**. It is `nil` only when both are nil.

```
type MyError struct{}
func (e *MyError) Error() string { return "boom" }

func doWork() error {
    var e *MyError = nil    // a nil pointer
    return e                // wrapped in an error interface
}

err := doWork()
fmt.Println(err == nil)     // false!
```

The returned interface holds type `*MyError` and value `nil`. The type is not nil, so the interface is not nil, so `err != nil` is true and your error handling runs for an error that is not there.

IF YOU KNOW JAVASCRIPT
There is no JavaScript equivalent because there is no such two-part value. The practical rule is the one from chapter four, and now you know why it exists: **declare the return type as `error` and return a literal `nil`**, never a typed nil variable.

```go
func doWork() error {
    if somethingWrong {
        return &MyError{}
    }
    return nil          // literal nil, not a typed nil
}
```

Interfaces you will meet constantly

Four from the standard library carry an enormous amount of Go code.

`io.Reader`	`Read(p []byte) (int, error)`. Anything you can read bytes from
`io.Writer`	`Write(p []byte) (int, error)`. Anything you can write bytes to
`error`	`Error() string`. You met it in chapter four
`fmt.Stringer`	`String() string`. How a type prints itself

`Stringer` is a quick win. Implement it and `fmt.Println` uses it:

```go
func (u User) String() string {
    return fmt.Sprintf("%s <%s>", u.Name, u.Email)
}

fmt.Println(u)      // Ada <ada@example.com>
```

IF YOU KNOW JAVASCRIPT

That is `toString()`, with one important difference: nothing is called implicitly during string concatenation, because Go has no implicit conversion. `"x" + u` does not compile. Only the `fmt` family looks for `String()`.

Why this matters for testing

The payoff arrives when you test. A function that takes an interface can be handed a fake with no mocking library and no dependency injection framework.

```go
// In the package that needs it, listing only what it uses.
type Mailer interface {
    Send(to, subject, body string) error
}

func Register(u User, m Mailer) error {
    // ...
```

```
    return m.Send(u.Email, "Welcome", "Hello "+u.Name)
}
```

The test:

```
type fakeMailer struct {
    sent []string
}

func (f *fakeMailer) Send(to, subject, body string) error {
    f.sent = append(f.sent, to)
    return nil
}

func TestRegisterSendsWelcome(t *testing.T) {
    m := &fakeMailer{}
    if err := Register(User{Email: "ada@example.com"}, m); err != nil {
        t.Fatal(err)
    }
    if len(m.sent) != 1 || m.sent[0] != "ada@example.com" {
        t.Errorf("got %v, want one mail to ada@example.com", m.sent)
    }
}
```

Six lines of fake, no library, no configuration. That is the whole argument for consumer-declared interfaces, and it only works because `Mailer` has one method rather than the fifteen a real mail client has.

PROJECT 9: A NOTIFIER WITH THREE BACKENDS

Build something that uses an interface the way Go intends, then prove the decoupling by adding a backend without touching the code that uses it.

```
package main

import (
    "fmt"
    "io"
    "os"
    "strings"
)

// Notifier is declared here, by the consumer, and has one method.
type Notifier interface {
    Notify(message string) error
}

// Three unrelated implementations. None of them mention Notifier.
```

```
type ConsoleNotifier struct{}

func (ConsoleNotifier) Notify(msg string) error {
    _, err := fmt.Println("[console]", msg)
    return err
}

type WriterNotifier struct{ W io.Writer }

func (w WriterNotifier) Notify(msg string) error {
    _, err := fmt.Fprintf(w.W, "[file] %s\n", msg)
    return err
}

type CollectingNotifier struct{ Messages []string }

func (c *CollectingNotifier) Notify(msg string) error {
    c.Messages = append(c.Messages, msg)
    return nil
}

// The consumer. It never learns which backend it has.
func alertAll(ns []Notifier, msg string) error {
    for _, n := range ns {
        if err := n.Notify(msg); err != nil {
            return fmt.Errorf("notifying: %w", err)
        }
    }
    return nil
}

func main() {
    var buf strings.Builder
    collector := &CollectingNotifier{}

    notifiers := []Notifier{
        ConsoleNotifier{},
        WriterNotifier{W: &buf},
        WriterNotifier{W: os.Stdout},
        collector,
    }

    if err := alertAll(notifiers, "disk almost full"); err != nil {
        fmt.Fprintln(os.Stderr, err)
        os.Exit(1)
    }

    fmt.Print("buffer contains: ", buf.String())
    fmt.Println("collector saw:", collector.Messages)
}
```

Notice three things. `WriterNotifier` is constructed twice with completely different destinations, a string builder and standard output, because both satisfy `io.Writer`. `CollectingNotifier` uses a pointer receiver because it mutates, so the slice holds `collector` rather than `CollectingNotifier{}`, and putting the value in instead is a compile error worth seeing. And `alertAll` would work unchanged against an email sender, a Slack client or a test fake.

Now add a fourth: a `PrefixNotifier` that wraps another `Notifier` and adds a prefix before delegating. That is the decorator pattern, in Go, in eight lines, with no base class:

```
type PrefixNotifier struct {
    Prefix string
    Inner  Notifier
}

func (p PrefixNotifier) Notify(msg string) error {
    return p.Inner.Notify(p.Prefix + msg)
}
```

YOUR TURN

Give `CollectingNotifier` a `String() string` method so it satisfies `fmt.Stringer`, then print the collector directly with `fmt.Println(collector)` and watch the output change. Then write a test for `alertAll` that proves it stops and returns a wrapped error when one notifier fails, using a fake that always errors. That test is four lines and needs no library, which is the point of the chapter.

ASSIGNMENT

Declare an interface where it is used, and feel the difference.

In `part2-ideas/10-interfaces`, write a `notify` package that sends a message.

One. The package defines what it needs, not what anybody provides:

```
type Sender interface {
    Send(ctx context.Context, to, body string) error
}
```

Two. Two implementations in other packages: one printing to stdout, one collecting into a slice. Neither imports `notify` to say it implements anything.

Three. A test that uses the collecting one and asserts on what was sent. No mocking library, no interface file, no registration.

Four. Add a third method to the interface and watch both implementations break at compile time. Then decide whether the method belonged there, and say so in `NOTES.md`.

The website's **Interfaces** and **Swapping Implementations** cover the mechanics if this is new.

COMMIT AS

```
git commit -m "Interfaces - Assignment One"
```

11

Designing with errors: what a package should return

Chapter four covered the mechanics: `if err != nil`, wrapping with `%w`, `errors.Is` and `errors.As`. This chapter is about the decision those mechanics leave you with, which is harder and matters more: **when someone calls your function and it fails, what should they be able to do about it?**

Get this right and the caller writes three lines. Get it wrong and they end up matching on error message strings, which is the clearest sign of an API that did not think about failure.

Three kinds of error, and when to use each

Opaque. The caller can log it and give up. Nothing else.

```
return fmt.Errorf("parsing config: %w", err)
```

This is the default and it is correct far more often than people expect. If there is nothing useful a caller could do differently, do not invent a way for them to try.

Sentinel. A specific, exported value the caller compares against.

```
var ErrNotFound = errors.New("not found")

func (s *Store) Get(id int) (*User, error) {
    ...
    return nil, ErrNotFound
}
```

The caller:

```
u, err := store.Get(id)
if errors.Is(err, ErrNotFound) {
    http.Error(w, "no such user", http.StatusNotFound)
    return
}
```

Use a sentinel when the condition is a **fact with no detail**: not found, already exists, timed out. The name is the entire payload.

Typed. A struct implementing `error`, carrying fields.

```
type ValidationError struct {
    Field  string
    Reason string
}

func (e *ValidationError) Error() string {
    return fmt.Sprintf("%s: %s", e.Field, e.Reason)
}
```

The caller:

```
var ve *ValidationError
if errors.As(err, &ve) {
    respond(w, 422, map[string]string{ve.Field: ve.Reason})
}
```

Use a typed error when the caller needs **data** from the failure, not just its identity. A validation failure needs the field name. A rate limit needs the retry-after. A conflict needs the version that is current.

Opaque	Caller logs and gives up. The default
Sentinel	Caller branches on identity. `errors.Is`
Typed	Caller needs fields out of it. `errors.As`

> **WATCH OUT**
> A sentinel or an exported error type is **part of your public API**. Renaming `ErrNotFound`, or changing which conditions return it, breaks callers exactly like changing a function signature. Declare them deliberately and few.
>
> The corollary: do not export an error type just because you made one. If nothing outside the package uses `errors.As` on it, keep it unexported and return it as a plain `error`.

Wrap, but not at every level

`%w` preserves the chain, and the temptation is to wrap at every function boundary. Do not. The result is a message like:

```
handling request: getting user: loading user: querying users:
executing query:
scanning row: sql: no rows in result set
```

Every layer added a word and none added information.

The rule worth following: **wrap when you add context the caller does not already have**. The database layer knows the query. The service layer knows the user ID. The handler knows the request. Each of those is worth one wrap. "I am a function and I called another function" is not context.

```
// Good: adds the thing the caller cannot see.
return fmt.Errorf("loading user %d: %w", id, err)

// Noise: the caller already knows they called LoadUser.
return fmt.Errorf("LoadUser: %w", err)
```

> **IF YOU KNOW JAVASCRIPT**
> This is the same discipline as a JavaScript stack trace, with one difference: Go errors have no stack trace by default, so the chain of wraps **is** your trace. That is why each one should earn its place. Some teams add stack traces with a library; the standard library deliberately does not, on the grounds that a well-wrapped error is more readable than sixty frames.

Error strings are a contract, lowercase and unpunctuated

Convention, followed throughout the standard library:

```
errors.New("connection refused")     // good
errors.New("Connection refused.")    // wrong: capital and full stop
```

The reason is composition. Errors get wrapped into sentences, and a capital in the middle reads badly:

```
loading config: Connection refused.: retrying
```

Also: do not end with a newline, and do not include the word "error". The reader knows.

Do not return both a value and an error

Pick one. A function that returns a partially filled struct **and** an error invites the caller to use the broken value.

```
// Bad: what is the caller supposed to do with a half-built User?
func Load(id int) (User, error)

// Good: nil on failure, impossible to misuse.
func Load(id int) (*User, error)
```

The exceptions are real but narrow: `io.Reader` returns a byte count with an error because a partial read is meaningful, and `strconv.Atoi` returns zero because it has nothing better to offer. If you are not implementing something in that shape, return nil.

Handle an error once

The most common error-handling bug in any language is doing two things with one failure:

```
if err != nil {
    log.Printf("saving user: %v", err)   // handled it here
    return err                           // and again, up the stack
}
```

Now it is logged at every level, and the operator reading the logs sees the same failure five times and has to work out whether it happened once or five times.

Choose: **either** handle it, **or** return it. The usual split is that libraries and services return, and the top layer, a handler or `main`, logs once and decides what the user sees.

Mapping errors at the boundary

This is where the three kinds pay off. One place in your program turns errors into HTTP responses:

```
func writeError(w http.ResponseWriter, err error) {
    var ve *ValidationError
    switch {
    case errors.Is(err, ErrNotFound):
        respond(w, 404, "not found")
    case errors.Is(err, ErrConflict):
        respond(w, 409, "already exists")
    case errors.As(err, &ve):
        respond(w, 422, map[string]string{ve.Field: ve.Reason})
    default:
        log.Printf("unhandled: %v", err)   // the only place that logs
        respond(w, 500, "internal error")
    }
}
```

Everything below this function returns errors and logs nothing. Everything above it gets a correct status code. That is the whole architecture of error handling in a Go service, and you will meet it again in part four, because it is exactly what Grit generates.

> **IF YOU KNOW JAVASCRIPT**
> In an Express app this is the error middleware, `app.use((err, req, res, next) => ...)`, and the design problem is identical: the middleware has to work out what kind of failure it received. JavaScript usually does it with `err instanceof ValidationError` or a `code` property. Go's `errors.As` is the same idea with the chain-walking built in, so it still works after the error has been wrapped three times.

panic is not an error

Go has `panic`, and it is not for expected failures. A panic unwinds the stack and, uncaught, kills the program.

Use `panic` for **programmer error that cannot be recovered from**: an impossible state, a nil dependency at startup, a configuration that cannot be parsed before the server starts serving.

```
func MustCompile(pattern string) *Regexp {
    re, err := Compile(pattern)
    if err != nil {
        panic("regexp: Compile(" + pattern + "): " + err.Error())
    }
    return re
}
```

The `Must` prefix is the convention: it means "this panics instead of returning an error, and you should only call it where failure means the program is broken", typically at package initialisation with a literal you wrote yourself.

Never panic for a bad request, a missing row, or a failed network call. Those are expected, and expected failures are values.

> **PROJECT 10: A USER STORE WITH A DESIGNED ERROR SURFACE**
> Build a small store that uses all three error kinds deliberately, plus the boundary mapping. This is a miniature of the structure every Go service ends up with.
>
> ```
> package main
>
> import (
>     "errors"
> ```

```
    "fmt"
    "strings"
)

// Sentinels: facts with no detail. Part of the public API.
var (
    ErrNotFound = errors.New("user not found")
    ErrConflict = errors.New("email already registered")
)

// Typed: the caller needs the field name out of it.
type ValidationError struct {
    Field  string
    Reason string
}

func (e *ValidationError) Error() string {
    return fmt.Sprintf("%s: %s", e.Field, e.Reason)
}

type User struct {
    ID    int
    Name  string
    Email string
}

type Store struct {
    users  map[int]*User
    nextID int
}

func NewStore() *Store {
    return &Store{users: map[int]*User{}, nextID: 1}
}

func (s *Store) Create(name, email string) (*User, error) {
    if strings.TrimSpace(name) == "" {
        return nil, &ValidationError{
            Field: "name", Reason: "must not be empty"}
    }
    if !strings.Contains(email, "@") {
        return nil, &ValidationError{
            Field: "email", Reason: "must be an email"}
    }
    for _, u := range s.users {
        if u.Email == email {
            // Wrapped, so the caller gets the identity and the detail.
            return nil, fmt.Errorf("creating user %s: %w",
                email, ErrConflict)
        }
    }
```

```
    u := &User{ID: s.nextID, Name: name, Email: email}
    s.users[u.ID] = u
    s.nextID++
    return u, nil
}

func (s *Store) Get(id int) (*User, error) {
    u, ok := s.users[id]
    if !ok {
        return nil, fmt.Errorf("getting user %d: %w", id, ErrNotFound)
    }
    return u, nil
}

// The boundary. The only place that decides what a failure means, and
// the
// only place that logs.
func describe(err error) string {
    var ve *ValidationError
    switch {
    case err == nil:
        return "200 ok"
    case errors.Is(err, ErrNotFound):
        return "404 not found"
    case errors.Is(err, ErrConflict):
        return "409 conflict"
    case errors.As(err, &ve):
        return fmt.Sprintf("422 invalid %s (%s)", ve.Field, ve.Reason)
    default:
        return "500 internal: " + err.Error()
    }
}

func main() {
    s := NewStore()

    _, err := s.Create("Ada", "ada@example.com")
    fmt.Println(describe(err))                  // 200 ok

    _, err = s.Create("", "x@example.com")
    fmt.Println(describe(err))                  // 422 invalid name

    _, err = s.Create("Grace", "not-an-email")
    fmt.Println(describe(err))                  // 422 invalid email

    _, err = s.Create("Someone", "ada@example.com")
    fmt.Println(describe(err))                  // 409 conflict

    _, err = s.Get(99)
    fmt.Println(describe(err))                  // 404 not found
}
```

Two things to prove to yourself. First, `errors.Is(err, ErrConflict)` returns true even though the error was wrapped with extra text, which is the entire reason to use `%w` rather than `%v`. Change that one verb to `%v` and watch the conflict case fall through to 500.

Second, note that `Create` never logs. It returns. `describe` is the only function that decides what a failure means to the outside world, so moving this store behind an HTTP API means changing one function.

YOUR TURN

Add an `Update(id int, name string) error` that can fail in three ways: the user is missing, the name is invalid, or the name is unchanged. Decide for each whether it is opaque, sentinel or typed, and be able to defend the choice. "Unchanged" is the interesting one: is that an error at all, or should `Update` quietly succeed? There is a defensible answer either way, and the act of choosing is the skill this chapter is teaching.

ASSIGNMENT

Design the errors of a package before writing its happy path.

In `part2-ideas/11-errors`, write a `store` package with `Get`, `Create` and `Delete` over an in-memory map.

One. A sentinel `ErrNotFound`, a typed `ValidationError` carrying the field name, and an unexpected error wrapped with context.

Two. A boundary function that maps any of them to an HTTP status and a code, using `errors.Is` and `errors.As` rather than string matching.

Three. Check every error string against the chapter's rule: lowercase, no full stop, no "failed to". Fix the ones that fail.

Four. Write a test for the boundary that asserts the status for each kind. That test is the contract your handlers will rely on in Part IV.

COMMIT AS

```
git commit -m "Errors - Assignment One"
```

12

defer, panic and recover

Three keywords that look like `finally`, `throw` and `catch` and are not. The resemblance is close enough to mislead, which is why they get a chapter.

> **THINGS YOU HAVE NOT MET YET**
> One example defers inside a goroutine. Goroutines are chapter 13; the point here is only that defer belongs to the function it is written in, not to the one that started it.

defer: cleanup that cannot be forgotten

`defer` schedules a function call to run when the **surrounding function** returns, by any path.

```
func readConfig(path string) error {
    f, err := os.Open(path)
    if err != nil {
        return err
    }
    defer f.Close()     // runs on every return below, and on a panic

    // twenty lines with six return statements
    return nil
}
```

Write the `defer` immediately after the thing that needs cleaning up. That adjacency is the point: open and close are one line apart, so a reviewer can see the pair without scrolling.

> **IF YOU KNOW JAVASCRIPT**
> `try { ... } finally { f.close() }` does the same job by wrapping, which indents the body and separates the close from the open. `defer` keeps them together and costs no indentation. It also composes: ten resources means ten `defer` lines, not ten nested blocks.

The three rules of defer

One: arguments are evaluated immediately, the call happens later.

```
func main() {
    msg := "first"
    defer fmt.Println(msg)   // prints "first", not "second"
    msg = "second"
}
```

`msg` was read when the `defer` statement ran. To defer the **evaluation**, wrap it in a closure:

```
defer func() { fmt.Println(msg) }()   // prints "second"
```

Two: deferred calls run last in, first out.

```
for i := 0; i < 3; i++ {
    defer fmt.Println(i)
}
// prints 2, then 1, then 0
```

This is right for cleanup: resources are released in the reverse of the order they were acquired, like unwinding a stack.

Three: defer is function-scoped, not block-scoped.

WATCH OUT

This is the one that causes real bugs:

```
func processAll(paths []string) error {
    for _, p := range paths {
        f, err := os.Open(p)
        if err != nil {
            return err
        }
        defer f.Close()     // does NOT run at the end of this iteration
        // ... use f
    }
    return nil
}
```

All the files stay open until `processAll` returns. With a thousand paths you run out of file descriptors.

The fix is to give each iteration its own function, so there is a return for the `defer` to fire on:

```
for _, p := range paths {
    if err := processOne(p); err != nil {
        return err
    }
}

func processOne(p string) error {
    f, err := os.Open(p)
    if err != nil {
        return err
    }
    defer f.Close()     // runs when processOne returns
    // ... use f
    return nil
}
```

Extracting the loop body into a function is the standard answer, and it usually improves the code for other reasons too.

defer can change the return value

A deferred closure can modify a **named** return value after the `return` statement has chosen it. This is the one legitimate use of named returns.

```
func doWork() (err error) {
    defer func() {
        if err != nil {
            err = fmt.Errorf("doWork: %w", err)
        }
    }()

    return doSomethingThatFails()    // gets wrapped on the way out
}
```

Used sparingly this is tidy. Used often it makes a function's return value hard to trace, so prefer explicit wrapping at each return unless you have many of them.

defer and errors from Close

A deferred `Close` discards its error, which is usually fine for reading and sometimes wrong for writing: a failed `Close` on a file you wrote may mean the data never reached the disk.

```
func writeReport(path string, data []byte) (err error) {
    f, err := os.Create(path)
    if err != nil {
        return err
```

```
    }
    defer func() {
        if cerr := f.Close(); cerr != nil && err == nil {
            err = fmt.Errorf("closing %s: %w", path, cerr)
        }
    }()

    _, err = f.Write(data)
    return err
}
```

That is verbose and correct. For a read-only file, `defer f.Close()` is accepted practice and nobody will object.

panic: stop, this should not have happened

`panic` unwinds the stack, running deferred functions as it goes, and if it reaches the top of a goroutine the program crashes with a stack trace.

```
func main() {
    defer fmt.Println("this still runs")
    panic("something impossible")
}
```

```
this still runs
panic: something impossible

goroutine 1 [running]:
main.main()
    /tmp/main.go:7 +0x65
exit status 2
```

Go panics on its own for a handful of runtime faults: nil pointer dereference, index out of range, division by zero, sending on a closed channel, and writing to a nil map.

> **IF YOU KNOW JAVASCRIPT**
> `throw` in JavaScript is for **expected** failures: a 404, a validation problem, a parse error. `panic` is not. Go's expected failures are returned values, and `panic` is reserved for "this program's assumptions are broken".
>
> If you find yourself panicking in a request handler because a user sent bad input, that is the JavaScript habit talking. Return an error.

recover: only for a boundary

`recover` stops a panic, and it only works **inside a deferred function**.

```
func safe() (err error) {
    defer func() {
        if r := recover(); r != nil {
            err = fmt.Errorf("recovered: %v", r)
        }
    }()

    panic("boom")
}

err := safe()
fmt.Println(err)    // recovered: boom
```

There are exactly two good reasons to use it.

One: a server that must not die because one request was bad. A panic in one handler should return 500 for that request, not take down the process serving everybody else.

```
func Recover(next http.Handler) http.Handler {
    return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
        defer func() {
            if rec := recover(); rec != nil {
                log.Printf("panic on %s %s: %v\n%s",
                    r.Method, r.URL.Path, rec, debug.Stack())
                http.Error(w, "internal server error",
                    http.StatusInternalServerError)
            }
        }()
        next.ServeHTTP(w, r)
    })
}
```

Log the stack. A recovered panic with no stack trace is a bug you will never find. Every Go web framework ships this middleware, Gin included, and Grit wires it in for you.

Two: a library turning an internal panic into an error at its edge. Some parsers panic internally to unwind out of deep recursion, then recover at the public function and return an error. The caller never sees a panic. This is legitimate and rare.

> **WATCH OUT**
> `recover` only catches panics in **its own goroutine**. A panic in a goroutine you started is not caught by a recover in the function that started it:
>
> ```
> func main() {
>     defer func() { recover() }()     // does not help
> ```

```
    go func() {
        panic("in a goroutine")      // kills the whole program
    }()
    time.Sleep(time.Second)
}
```

Every goroutine that could panic needs its own recover. This is the single most common way a Go server dies in production: a background worker with no recover, panicking on a nil map six hours after deployment.

PROJECT 11: A WORKER THAT SURVIVES BAD INPUT

Build a small job runner where one bad job must not stop the rest, and where cleanup is guaranteed. This is the panic boundary in miniature.

```
package main

import (
    "errors"
    "fmt"
    "strings"
)

type Job struct {
    Name string
    Run  func() error
}

// runOne is the boundary. Whatever the job does, this returns an error
// rather than letting a panic escape.
func runOne(j Job) (err error) {
    defer func() {
        if r := recover(); r != nil {
            err = fmt.Errorf("job %q panicked: %v", j.Name, r)
        }
    }()
    defer fmt.Printf("  [%s] finished\n", j.Name)   // runs even on panic

    if j.Run == nil {
        return errors.New("job has no work")
    }
    return j.Run()
}

func runAll(jobs []Job) (failed int) {
    for _, j := range jobs {
        fmt.Printf("running %s\n", j.Name)
        if err := runOne(j); err != nil {
            fmt.Println("  error:", err)
```

```
            failed++
        }
    }
    return failed
}

func main() {
    jobs := []Job{
        {Name: "ok", Run: func() error { return nil }},
        {Name: "fails", Run: func() error {
            return errors.New("disk full")
        }},
        {Name: "panics", Run: func() error {
            var m map[string]int
            m["boom"] = 1              // panic: assignment to nil map
            return nil
        }},
        {Name: "index", Run: func() error {
            s := []int{1, 2, 3}
            fmt.Println(s[10])         // panic: index out of range
            return nil
        }},
        {Name: "no work"},
        {Name: "still runs", Run: func() error {
            fmt.Println("  the runner survived four failures")
            return nil
        }},
    }

    failed := runAll(jobs)
    fmt.Printf("\n%d of %d jobs failed\n", failed, len(jobs))
    _ = strings.TrimSpace("")
}
```

Three things to notice when you run it.

The `finished` line prints for every job including the ones that panicked, because deferred calls run while the stack unwinds. That is what makes `defer` safe for cleanup in a way that a line at the end of the function is not.

The two deferred functions run in reverse order, so `finished` prints before the recover's error is assembled.

And the program exits normally with a count, rather than dying on job three. Delete the `recover` block and run it again: you get a stack trace and jobs four, five and six never run. That difference is the reason every server has this middleware.

YOUR TURN

Move the `recover` from `runOne` up into `runAll`, wrapping the whole loop, and run it again. You will catch the first panic and then the loop stops, because the panic unwound past the loop. That experiment shows exactly what "a boundary" means: it has to be inside the unit of work you want to isolate, not outside it.

ASSIGNMENT

Copy a file correctly, which is harder than it looks.

In `part2-ideas/12-defer`, write `CopyFile(dst, src string) (err error)`.

One. Open both files, `defer` both closes, and copy with `io.Copy`.

Two. Return the error from closing the destination, because a failed flush is a failed copy. Use the named return and the deferred assignment from this chapter.

Three. Write the loop version that opens a hundred files and defers every close inside the loop. Watch the file descriptors, then fix it, and write the rule in a comment.

Four. Add a `recover` boundary to a tiny handler, log the panic with the stack, and return a 500. Then panic on purpose and confirm the process survives.

COMMIT AS

```
git commit -m "Defer - Assignment One"
```

13

Goroutines, against the event loop

You already write concurrent code. Every `await` you have typed was concurrency. What you have not had is **parallelism**: two pieces of your own code running at the same instant on two cores. This chapter is about the difference, because it is the whole reason Go's model exists.

> **THINGS YOU HAVE NOT MET YET**
> This chapter starts goroutines and mostly waits for them crudely. Channels are chapter 14 and sync.WaitGroup and sync.Mutex are chapter 15. Those are the proper answers; this chapter is about what a goroutine is.

The one-keyword version

Put `go` in front of a function call and it runs independently.

```
func main() {
    go fmt.Println("from a goroutine")
    fmt.Println("from main")
    time.Sleep(100 * time.Millisecond)
}
```

That is the entire syntax. No `async` on the function, no `await` at the call, no promise returned. Any function can be a goroutine; nothing about its declaration changes.

> **WATCH OUT**
> The `time.Sleep` is there because **when `main` returns, the program exits immediately**, killing every goroutine still running. Remove it and you will probably see only "from main".
>
> Sleeping is never the real answer. The real answers are `sync.WaitGroup` and channels, both below. But almost every Go tutorial starts with a sleep, and it is worth knowing that it is a placeholder for synchronisation rather than a technique.

What is actually different

In Node, your JavaScript runs on **one** thread. The event loop interleaves work: while one request waits on the database, another gets a turn. That is excellent for I/O and useless for computation, because a function that spends two hundred milliseconds doing arithmetic blocks everything.

Go runs goroutines on a pool of operating system threads, usually one per CPU core. Two goroutines can execute at literally the same moment.

	Node	Go
Unit	Promise / task	Goroutine
Threads	One for your code	One per core, by default
Waiting on I/O	Excellent	Excellent
CPU work	Blocks everything	Uses every core
Cost each	Small object	About 2 KB of stack
Practical limit	Many thousands	Hundreds of thousands

A goroutine starts with a two kilobyte stack that grows as needed. An operating system thread starts with one or two **megabytes**. That thousand-fold difference is why “start a goroutine per request” is a normal design in Go and “start a thread per request” was never viable in Java or C#.

> **IF YOU KNOW JAVASCRIPT**
> The mental shift: in Node you **structure** code so it never blocks, because blocking is catastrophic. In Go you write straight-line blocking code, and the runtime moves the goroutine off the thread while it waits. There is no colour to functions, no `async` infecting every caller up the stack. A function that reads a file looks like a function that reads a file.

Waiting properly: sync.WaitGroup

A `WaitGroup` counts outstanding work.

```
func main() {
    var wg sync.WaitGroup

    for i := 1; i <= 3; i++ {
        wg.Add(1)                  // one more to wait for
        go func() {
            defer wg.Done()        // one fewer, even if this panics
```

```
            fmt.Println("worker", i)
        }()
    }

    wg.Wait()                       // blocks until the count is zero
    fmt.Println("all done")
}
```

Three rules, and the bugs that come from breaking them:

`wg.Add` before the `go`, never inside it. Inside, `Wait` may run before the goroutine starts and see a count of zero.

`defer wg.Done()` as the first line. A panic or an early return otherwise leaves the count above zero and `Wait` blocks forever.

Pass the `WaitGroup` by pointer if it crosses a function boundary. Copying one gives the callee its own counter, and the original never reaches zero.

Since Go 1.25 there is a shorthand that removes the first two:

```
var wg sync.WaitGroup
for i := 1; i <= 3; i++ {
    wg.Go(func() {              // Add and Done handled for you
        fmt.Println("worker", i)
    })
}
wg.Wait()
```

Collecting results

Goroutines return nothing. A `go` call is a statement, not an expression, so there is no value to take.

Write into a pre-sized slice, one index per goroutine. No lock is needed because no two goroutines touch the same element.

```
type result struct {
    url string
    ms  int64
    err error
}

func fetchAll(urls []string) []result {
    results := make([]result, len(urls))      // sized up front
    var wg sync.WaitGroup

    for i, url := range urls {
        wg.Go(func() {
```

```
            start := time.Now()
            resp, err := http.Get(url)
            if err != nil {
                results[i] = result{url: url, err: err}
                return
            }
            defer resp.Body.Close()
            // Read it, or the connection leaks.
            io.Copy(io.Discard, resp.Body)
            results[i] = result{
                url: url, ms: time.Since(start).Milliseconds()}
        })
    }

    wg.Wait()
    return results
}
```

IF YOU KNOW JAVASCRIPT

This is `Promise.all`, with the ordering preserved the same way: by index rather than by completion. The difference is that an error in one goroutine does not reject the whole batch. Here every result carries its own error, which is usually what you wanted from `Promise.allSettled` anyway.

Appending to a shared slice instead of indexing would be a data race, because `append` reads and writes the slice header. Index assignment to distinct elements is safe.

Races, and the detector that finds them

Two goroutines touching the same variable, at least one of them writing, with no synchronisation, is a **data race**. The result is not "the wrong number", it is undefined.

```
func main() {
    count := 0
    var wg sync.WaitGroup
    for i := 0; i < 1000; i++ {
        wg.Go(func() { count++ })    // RACE
    }
    wg.Wait()
    fmt.Println(count)               // 973, 981, 1000, who knows
}
```

`count++` is three operations: read, add, write. Two goroutines interleaving them lose updates.

Go ships a race detector, and it is the most valuable tool in the toolchain:

```
go run -race main.go
go test -race ./...
```

```
WARNING: DATA RACE
Write at 0x00c0000140a8 by goroutine 8:
  main.main.func1()
      /tmp/main.go:12 +0x3c
Previous write at 0x00c0000140a8 by goroutine 7:
  ...
```

WATCH OUT

Run your tests with `-race` in CI. Always. A data race can run correctly ten million times and corrupt data on the ten-million-and-first, on a different machine, under load. The detector finds it the first time the racing access happens, which turns an unreproducible production mystery into a failing test.

It costs roughly ten times the CPU and a few times the memory, so it is a CI and development tool rather than something you ship.

The fix for the counter is a mutex:

```
var (
    mu    sync.Mutex
    count int
)

wg.Go(func() {
    mu.Lock()
    defer mu.Unlock()
    count++
})
```

Or, for exactly this shape, an atomic:

```
var count atomic.Int64
wg.Go(func() { count.Add(1) })
fmt.Println(count.Load())
```

Bounding how many run at once

Unbounded goroutines are a real hazard. Ten thousand URLs means ten thousand simultaneous connections, which will exhaust file descriptors or get you blocked.

A buffered channel used as a semaphore caps the concurrency:

```
func fetchAll(urls []string, limit int) []result {
    results := make([]result, len(urls))
    sem := make(chan struct{}, limit)    // room for `limit` tokens
    var wg sync.WaitGroup

    for i, url := range urls {
        wg.Go(func() {
            sem <- struct{}{}            // take a token, blocking if full
            defer func() { <-sem }()     // give it back

            // ... fetch, as before
            _ = i
            _ = url
        })
    }

    wg.Wait()
    return results
}
```

Channels are the next chapter. For now, read `sem <- struct{}{}` as "wait until there is room, then claim a slot".

> **IF YOU KNOW JAVASCRIPT**
> The JavaScript equivalent is a library like `p-limit`, because the language gives you nothing for it. Here it is two lines and a channel, which is a fair summary of the trade Go makes across the board: fewer batteries, better primitives.

When not to use a goroutine

Starting one is cheap, which makes it tempting. Three cases where it is wrong:

When the work is faster than the coordination. Spawning a goroutine to add two numbers costs more than adding them.

When you do not wait for it. A goroutine whose result nobody collects, with no `WaitGroup` and no channel, is a leak. If `main` exits it is killed silently; if it blocks forever it holds its stack forever.

When it needs ordering. Goroutines have no execution order. If step two must follow step one, write them in sequence.

> **PROJECT 12: A CONCURRENT LINK CHECKER**
> Check a list of URLs in parallel, with a concurrency limit and a timeout, and report which are broken. This is a genuinely useful tool and it exercises everything in this chapter.

```go
package main

import (
    "fmt"
    "io"
    "net/http"
    "os"
    "sort"
    "sync"
    "time"
)

type check struct {
    url    string
    status int
    ms     int64
    err    error
}

func checkAll(urls []string, limit int) []check {
    results := make([]check, len(urls))
    sem := make(chan struct{}, limit)
    var wg sync.WaitGroup

    client := &http.Client{Timeout: 10 * time.Second}

    for i, url := range urls {
        wg.Go(func() {
            sem <- struct{}{}
            defer func() { <-sem }()

            start := time.Now()
            resp, err := client.Get(url)
            if err != nil {
                results[i] = check{url: url, err: err}
                return
            }
            defer resp.Body.Close()
            io.Copy(io.Discard, resp.Body)

            results[i] = check{
                url:    url,
                status: resp.StatusCode,
                ms:     time.Since(start).Milliseconds(),
            }
        })
    }

    wg.Wait()
    return results
}
```

```
func main() {
    urls := os.Args[1:]
    if len(urls) == 0 {
        fmt.Fprintln(os.Stderr, "usage: check <url>...")
        os.Exit(1)
    }

    start := time.Now()
    results := checkAll(urls, 5)
    total := time.Since(start)

    // Slowest first, so the problems are at the top.
    sort.Slice(results, func(a, b int) bool {
        return results[a].ms > results[b].ms
    })

    broken := 0
    for _, r := range results {
        switch {
        case r.err != nil:
            broken++
            fmt.Printf("  ERR  %-45s %v\n", r.url, r.err)
        case r.status >= 400:
            broken++
            fmt.Printf("  %d  %-45s %dms\n", r.status, r.url, r.ms)
        default:
            fmt.Printf("  %d  %-45s %dms\n", r.status, r.url, r.ms)
        }
    }

    fmt.Printf("\nchecked %d urls in %v, %d broken\n",
        len(results), total.Round(time.Millisecond), broken)
    if broken > 0 {
        os.Exit(1)
    }
}
```

Run it against a handful of sites and notice that the total time is roughly the **slowest** request, not the sum of all of them. That is the payoff.

Three details that matter in real code. The `http.Client` has a timeout: without one, a hung server holds a goroutine forever and `wg.Wait` never returns. The response body is read and closed, because an unread body leaks the connection. And the exit code is non-zero when something is broken, which is what makes this usable in CI.

Now run it with `-race`. It will be clean, because each goroutine writes only to `results[i]`. Then introduce a race on purpose: add a shared `var count int` and `count++` inside the goroutine, and run with `-race` again to see the detector's report.

YOUR TURN

Add a `-n` flag for the concurrency limit and try the same URL list at 1, 5 and 50. At 1 the total is the sum of all requests. At 5 it drops sharply. At 50 it probably stops improving, because you are now limited by the network or by the servers themselves. Finding the point where more concurrency stops helping is the actual skill, and it is measured rather than guessed.

ASSIGNMENT

Fetch many things at once, correctly, and prove it with the race detector.

In `part2-ideas/13-goroutines`, fetch twenty URLs and report each status.

One. Sequentially first. Time it with `time.Since` and keep the number.

Two. Concurrently with a `sync.WaitGroup`, collecting into a slice indexed by position so there is no shared counter. Time it again.

Three. Now write the broken version that appends to one slice from every goroutine, and run it:

```
go run -race .
```

Paste the detector's output into `NOTES.md`. It names both goroutines and both lines, which is the thing worth remembering.

Four. Bound it to four at a time with a buffered channel, and explain in one sentence why twenty thousand goroutines against one host is rude rather than fast.

COMMIT AS

```
git commit -m "Goroutines - Assignment One"
```

14

Channels, and the select statement

Goroutines run independently. Channels are how they talk. The slogan Go is famous for is **do not communicate by sharing memory; share memory by communicating**, which sounds like a fortune cookie until you have written both versions of the same program.

> **THINGS YOU HAVE NOT MET YET**
> sync.WaitGroup turns up in a couple of examples and is chapter 15. A channel can do the same job, less clearly, which is the comparison.

A channel is a typed pipe

```
ch := make(chan int)    // unbuffered
ch <- 42                // send
v := <-ch               // receive
```

The arrow always points in the direction the value travels. `ch <- v` puts a value in, `<-ch` takes one out.

```
func main() {
    ch := make(chan string)

    go func() {
        ch <- "hello from the goroutine"
    }()

    msg := <-ch          // blocks until something arrives
    fmt.Println(msg)
}
```

No `time.Sleep` this time. The receive blocks until the send happens, which is synchronisation and message passing in one operation.

> **IF YOU KNOW JAVASCRIPT**

The closest JavaScript analogy is an async generator or a stream, with one crucial difference: a channel receive **blocks the goroutine**. There is no callback, no `await`, and no promise. The code reads top to bottom, and the runtime parks the goroutine while it waits.

If you have used `postMessage` between workers, the mental model is closer: separate lines of execution passing values, except channels are typed and synchronous by default.

Unbuffered channels are a handshake

An unbuffered channel has no room. A send blocks until a receiver is ready, and a receive blocks until a sender is ready. The two goroutines meet.

```
ch := make(chan int)
ch <- 1                 // deadlock: nobody is receiving
```

```
fatal error: all goroutines are asleep - deadlock!
```

Go detects when every goroutine is blocked and stops the program rather than hanging forever. This is a feature, and it is how you will find most channel bugs.

Buffered channels have room

```
ch := make(chan int, 3)  // capacity 3
ch <- 1                  // does not block
ch <- 2
ch <- 3
ch <- 4                  // now it blocks: the buffer is full
```

A send blocks only when the buffer is full; a receive blocks only when it is empty.

WATCH OUT

Buffering is not a performance trick to sprinkle on. The capacity is a design decision: it is how many items may be in flight before the producer must slow down. That back pressure is usually what you want, because an unbounded queue in front of a slow consumer is just a memory leak with extra steps.

Choose zero when you want a handshake, and a specific number when you know how much slack the system should tolerate. “I made it 100 so it would not block” is how a queue silently grows to 100 and then blocks anyway.

Closing, and ranging

The sender closes a channel to say “no more values”.

```
func produce(ch chan<- int) {
    for i := 0; i < 5; i++ {
        ch <- i
    }
    close(ch)           // only the sender closes
}

func main() {
    ch := make(chan int)
    go produce(ch)

    for v := range ch {  // ends when the channel is closed
        fmt.Println(v)
    }
}
```

Receiving from a closed channel returns the zero value immediately. The two-value form tells you which happened:

```
v, ok := <-ch
if !ok {
    // the channel is closed and drained
}
```

> **WATCH OUT**
> Three rules that cover every close-related bug:
>
> **Only the sender closes.** A receiver closing a channel causes the sender to panic on its next send.
>
> **Closing twice panics.** So does sending on a closed channel.
>
> **You do not have to close.** A channel that is garbage collected is fine. Close only to signal completion to a `range` or a `select`.

Note `chan<- int` in that signature: a **directional** channel type, meaning send-only. `<-chan int` is receive-only. Using them in function parameters makes the data flow obvious and lets the compiler stop a receiver from closing.

select: waiting on several channels

`select` blocks until one of its cases can proceed.

```
select {
case msg := <-ch1:
    fmt.Println("from ch1:", msg)
case msg := <-ch2:
```

```
    fmt.Println("from ch2:", msg)
case ch3 <- "value":
    fmt.Println("sent to ch3")
}
```

If several are ready it picks one at random, which prevents one busy channel starving another.

A `default` case makes it non-blocking:

```
select {
case msg := <-ch:
    fmt.Println(msg)
default:
    fmt.Println("nothing waiting")   // does not block
}
```

> **IF YOU KNOW JAVASCRIPT**
> `select` is `Promise.race`, with two differences worth noting. It can wait on sends as well as receives. And the losing cases are not "cancelled", they simply were not chosen: the other channels still hold their values for the next receive. `Promise.race` leaves the losers running and their results are discarded, which is a frequent source of leaked work in JavaScript.

Timeouts, the most common select

```
select {
case result := <-work:
    fmt.Println("finished:", result)
case <-time.After(2 * time.Second):
    fmt.Println("timed out")
}
```

`time.After` returns a channel that delivers a value after the duration. This is the idiomatic timeout, and it is three lines.

> **WATCH OUT**
> `time.After` leaks until it fires. In a tight loop, creating one per iteration piles up timers for the full duration. For a loop, create a `time.NewTimer` once and reset it, or use a context deadline, which is the next chapter.

Signalling with a done channel

A channel carrying no data, used only to signal, is a common shape:

```
func worker(done <-chan struct{}) {
    for {
        select {
        case <-done:
            fmt.Println("stopping")
            return
        default:
            // do a unit of work
        }
    }
}

func main() {
    done := make(chan struct{})
    go worker(done)

    time.Sleep(time.Second)
    close(done)          // closing broadcasts to every receiver
    time.Sleep(10 * time.Millisecond)
}
```

Closing is the broadcast mechanism: **every** goroutine blocked on `<-done` wakes up at once, because a closed channel always yields immediately. Sending a value would wake only one.

This pattern is so common that the standard library wrapped it, and that wrapper is `context`, which is chapter sixteen. Understand the done channel first and `context` becomes obvious rather than magical.

The nil channel, and why it is useful

Sending or receiving on a nil channel blocks forever. That sounds like a pure bug and it has one real use: disabling a `select` case.

```
for {
    select {
    case v, ok := <-in:
        if !ok {
            in = nil        // disable this case, keep the loop alive
            continue
        }
        process(v)
    case <-done:
        return
    }
}
```

Once `in` is nil its case can never be chosen, so the loop keeps serving `done` without spinning on a closed channel returning zeros forever.

Worker pools

The canonical channel program: a fixed number of workers pulling from a jobs channel.

```
func worker(id int, jobs <-chan int, results chan<- string,
    wg *sync.WaitGroup) {
    defer wg.Done()
    for j := range jobs {           // ends when jobs is closed
        results <- fmt.Sprintf("worker %d did job %d", id, j)
    }
}

func main() {
    jobs := make(chan int, 100)
    results := make(chan string, 100)
    var wg sync.WaitGroup

    for w := 1; w <= 3; w++ {
        wg.Add(1)
        go worker(w, jobs, results, &wg)
    }

    for j := 1; j <= 9; j++ {
        jobs <- j
    }
    close(jobs)                     // tells the workers no more are coming

    wg.Wait()                       // wait for them to finish
    close(results)                  // now safe to close

    for r := range results {
        fmt.Println(r)
    }
}
```

The ordering is the whole lesson. Close `jobs` so the workers' `range` ends. `Wait` for the workers. **Then** close `results`, because closing it earlier would panic a worker still writing.

> **IF YOU KNOW JAVASCRIPT**
> Compare this to the semaphore from chapter thirteen. Both bound concurrency. The semaphore starts a goroutine per item and limits how many run; the worker pool starts a fixed number and feeds them. The pool is better when goroutine startup matters or

when each worker holds an expensive resource, like a database connection. The semaphore is simpler and usually enough.

PROJECT 13: A PIPELINE WITH CANCELLATION

Build a three-stage pipeline where each stage is a goroutine connected by channels, and where the whole thing can be stopped early. This is the shape of a great deal of real Go.

```
package main

import (
    "fmt"
    "strings"
    "sync"
    "time"
)

// Stage 1: produce lines. Returns a channel rather than taking one,
// which is the convention: the producer owns and closes what it
// creates.
func generate(done <-chan struct{}, lines []string) <-chan string {
    out := make(chan string)
    go func() {
        defer close(out)
        for _, l := range lines {
            select {
            case out <- l:
            case <-done:
                return          // stop early without leaking
            }
        }
    }()
    return out
}

// Stage 2: normalise.
func normalise(done <-chan struct{}, in <-chan string) <-chan string {
    out := make(chan string)
    go func() {
        defer close(out)
        for l := range in {
            select {
            case out <- strings.ToLower(strings.TrimSpace(l)):
            case <-done:
                return
            }
        }
    }()
    return out
}
```

```
// Stage 3: count words, fanned out across several goroutines.
func count(done <-chan struct{}, in <-chan string,
    workers int) <-chan int {
    out := make(chan int)
    var wg sync.WaitGroup

    for i := 0; i < workers; i++ {
        wg.Go(func() {
            for l := range in {
                time.Sleep(10 * time.Millisecond)   // pretend it is slow
                select {
                case out <- len(strings.Fields(l)):
                case <-done:
                    return
                }
            }
        })
    }

    // One goroutine whose only job is to close out once every worker is
    // finished. Without this, nothing ever closes out and main hangs.
    go func() {
        wg.Wait()
        close(out)
    }()

    return out
}

func main() {
    lines := []string{
        "  The quick brown fox  ", "jumps over", "THE LAZY DOG",
        "and then", "runs away quickly", "into the woods",
    }

    done := make(chan struct{})
    defer close(done)            // every stage stops when main returns

    start := time.Now()
    total := 0
    pipeline := count(done, normalise(done, generate(done, lines)), 3)
    for n := range pipeline {
        total += n
    }

    fmt.Printf("%d words in %v\n", total,
        time.Since(start).Round(time.Millisecond))
}
```

Run it and note the time: six lines at ten milliseconds each would be sixty milliseconds serially, and with three workers it is closer to twenty.

Then do the experiment that proves the cancellation works. Change the loop in `main` to stop after the first two results:

```
n := 0
for range count(done, normalise(done, generate(done, lines)), 3) {
    n++
    if n == 2 {
        break
    }
}
```

The `defer close(done)` fires, every stage's `select` takes the `<-done` case, and all the goroutines exit. Remove the `done` channel entirely and the producer blocks forever trying to send into a pipeline nobody is reading: a goroutine leak, which in a long-running server is a slow memory leak you will eventually have to find.

Run the whole thing with `-race` as well. It is clean, because every value moves through a channel and nothing is shared.

YOUR TURN

Add a fourth stage between normalise and count that drops any line shorter than three words, and confirm the total changes. Then count how many goroutines the program runs at its peak. If your answer is not "one per stage, plus the workers, plus the closer", re-read `count`: that little goroutine whose only job is `wg.Wait()` then `close(out)` is the piece people forget, and forgetting it is a deadlock rather than a wrong answer.

ASSIGNMENT

A worker pool that shuts down cleanly, which is the part people skip.

In `part2-ideas/14-channels`:

One. A pool of five workers reading jobs from one channel and writing results to another. Close the jobs channel to end it, and `range` in the workers.

Two. A `select` with a timeout, so a job that takes too long is abandoned rather than waited on.

Three. A done channel that stops everything from outside, and a run where you trigger it halfway. No goroutine may be left behind: check with

```
fmt.Println(runtime.NumGoroutine())
```

Four. Send on a closed channel on purpose, read the panic, and write in `NOTES.md` the rule about who closes a channel.

COMMIT AS

```
git commit -m "Channels - Assignment One"
```

15

sync: mutexes, and when a channel is the wrong tool

The previous chapter sold channels. This one is the correction: **channels are not always the answer**, and reaching for one where a mutex belongs produces code that is slower and harder to read.

The Go team's own guidance, from the wiki: use a channel to pass ownership of data or to distribute work; use a mutex to protect shared state. The rest of this chapter is what that means in practice.

Mutex: one goroutine at a time

```
type Counter struct {
    mu sync.Mutex
    n  int
}

func (c *Counter) Inc() {
    c.mu.Lock()
    defer c.mu.Unlock()
    c.n++
}

func (c *Counter) Value() int {
    c.mu.Lock()
    defer c.mu.Unlock()
    return c.n
}
```

`Lock` blocks until no other goroutine holds the lock. `defer c.mu.Unlock()` on the line after is the pattern: a path that returns early without unlocking deadlocks the whole program.

> **WATCH OUT**
> Four rules, each the source of a real bug:

The zero value works. `var mu sync.Mutex` is ready. Never `mu := sync.Mutex{}` then copy it around.

Never copy a struct containing a mutex. Copying duplicates the lock, so two copies protect nothing. This is why `Inc` has a pointer receiver. `go vet` catches most of these.

Lock the reads too. A read racing with a write is still a data race, and the race detector will say so. Reading an `int` is not atomic in any sense you can rely on.

Keep the critical section small. Never do I/O, call an unknown function, or send on a channel while holding a lock. That is how a lock becomes a queue and then a deadlock.

Embedding the mutex

Common style, and it reads well:

```
type SafeMap struct {
    sync.Mutex                  // embedded: m.Lock() works directly
    items map[string]int
}

func (m *SafeMap) Set(k string, v int) {
    m.Lock()
    defer m.Unlock()
    m.items[k] = v
}
```

The cost is that `Lock` and `Unlock` become part of the type's public API, so callers can lock your struct. Keep the mutex a named field when the type is exported from a package.

RWMutex: many readers, one writer

When reads vastly outnumber writes, `sync.RWMutex` lets readers run concurrently.

```
type Cache struct {
    mu    sync.RWMutex
    items map[string]string
}

func (c *Cache) Get(k string) (string, bool) {
    c.mu.RLock()                // read lock: others may also read
    defer c.mu.RUnlock()
    v, ok := c.items[k]
    return v, ok
}

func (c *Cache) Set(k, v string) {
    c.mu.Lock()                 // write lock: exclusive
```

```
    defer c.mu.Unlock()
    c.items[k] = v
}
```

> **WATCH OUT**
> `RWMutex` is not automatically faster. It has more bookkeeping than a plain `Mutex`, so for short critical sections with moderate contention a `Mutex` often wins. Use `RWMutex` when reads are frequent **and** the critical section is long enough to matter, and confirm with a benchmark rather than a feeling.
>
> Also: never upgrade an `RLock` to a `Lock` while holding it. That deadlocks. Release, acquire the write lock, and re-check the condition, because things may have changed in between.

sync.Once: exactly once, no matter who asks

```
var (
    once   sync.Once
    client *http.Client
)

func Client() *http.Client {
    once.Do(func() {
        client = &http.Client{Timeout: 10 * time.Second}
    })
    return client
}
```

`Do` runs the function once, ever. Concurrent callers block until the first one finishes, so nobody sees a half-built value.

> **IF YOU KNOW JAVASCRIPT**
> This is the lazy singleton you would write in JavaScript as a module-level `let instance` with an `if (!instance)` check. That check is a race in Go, because two goroutines can both see nil. `sync.Once` is the correct version and is shorter anyway.

atomic: for single values

For a counter or a flag, a lock is heavier than necessary.

```
var requests atomic.Int64

func handle() {
    requests.Add(1)
```

```
}

func report() {
    fmt.Println(requests.Load())
}
```

The typed atomics (`atomic.Int64`, `atomic.Bool`, `atomic.Pointer[T]`) arrived in Go 1.19 and are much nicer than the old function forms. They cannot be copied, which the compiler enforces.

Use atomics only for a **single** value. The moment two values must change together, you need a mutex, because two atomic operations are not one atomic operation:

```
// WRONG: another goroutine can see total updated and count not.
total.Add(n)
count.Add(1)
```

sync.Map, and why you probably do not want it

There is a concurrent map in the standard library. It is specialised, and the documentation says so plainly: it is optimised for two cases, where a key is written once and read many times, or where goroutines work on disjoint key sets.

For everything else, a plain map with a mutex is faster and far more readable, because `sync.Map` has no type safety: everything is `any` and you assert on the way out.

```
// Usually better:
type Store struct {
    mu sync.RWMutex
    m  map[string]User
}
```

Reach for `sync.Map` after measuring, not before.

When to use which

Channel	Passing ownership of a value, distributing work, signalling an event
Mutex	Protecting state that several goroutines read and write in place
RWMutex	The same, when reads dominate and the section is long
atomic	One number or flag, nothing else
Once	Lazy initialisation
WaitGroup	Waiting for a known set of goroutines to finish

The counter from chapter thirteen makes the point. With a channel:

```
// Works, and is the wrong tool.
func counter(inc <-chan int, get chan<- int) {
    n := 0
    for {
        select {
        case v := <-inc:
            n += v
        case get <- n:
        }
    }
}
```

That is a goroutine, two channels, a `select` loop that never exits, and a lifetime you now have to manage. Against:

```
var n atomic.Int64
n.Add(1)
```

The channel version is the one that looks clever and the atomic is the one you want. **Sharing a number is not communication.**

PROJECT 14: AN IN-MEMORY CACHE WITH EXPIRY

Build something with real shared state: a cache with per-entry expiry and a background cleaner. It needs a mutex, a `Once`, an atomic and a goroutine, and the combination is typical of real service code.

```
package main

import (
    "fmt"
    "sync"
    "sync/atomic"
    "time"
)

type entry struct {
    value   string
    expires time.Time
}

type Cache struct {
    mu      sync.RWMutex
    items   map[string]entry
    hits    atomic.Int64      // counters: atomic, not under the lock
    misses  atomic.Int64
    stopper sync.Once
```

```
    done    chan struct{}
}

func NewCache(cleanEvery time.Duration) *Cache {
    c := &Cache{
        items: map[string]entry{},
        done:  make(chan struct{}),
    }
    go c.clean(cleanEvery)
    return c
}

func (c *Cache) Set(k, v string, ttl time.Duration) {
    c.mu.Lock()
    defer c.mu.Unlock()
    c.items[k] = entry{value: v, expires: time.Now().Add(ttl)}
}

func (c *Cache) Get(k string) (string, bool) {
    c.mu.RLock()
    e, ok := c.items[k]
    c.mu.RUnlock()            // released before the branch below

    if !ok || time.Now().After(e.expires) {
        c.misses.Add(1)
        return "", false
    }
    c.hits.Add(1)
    return e.value, true
}

func (c *Cache) clean(every time.Duration) {
    t := time.NewTicker(every)
    defer t.Stop()
    for {
        select {
        case <-t.C:
            now := time.Now()
            c.mu.Lock()
            for k, e := range c.items {
                if now.After(e.expires) {
                    // deleting while ranging is legal
                    delete(c.items, k)
                }
            }
            c.mu.Unlock()
        case <-c.done:
            return
        }
    }
}
```

```
// Close is safe to call more than once, which a Once gives us free.
func (c *Cache) Close() {
    c.stopper.Do(func() { close(c.done) })
}

func (c *Cache) Stats() (hits, misses int64) {
    return c.hits.Load(), c.misses.Load()
}

func main() {
    c := NewCache(50 * time.Millisecond)
    defer c.Close()

    c.Set("a", "alpha", 200*time.Millisecond)
    c.Set("b", "bravo", 40*time.Millisecond)

    // Hammer it from several goroutines to prove the locking holds.
    var wg sync.WaitGroup
    for i := 0; i < 50; i++ {
        wg.Go(func() {
            c.Get("a")
            c.Set(fmt.Sprintf("k%d", i), "v", time.Second)
        })
    }
    wg.Wait()

    time.Sleep(100 * time.Millisecond)   // let b expire and be cleaned

    if _, ok := c.Get("b"); !ok {
        fmt.Println("b expired, as expected")
    }
    h, m := c.Stats()
    fmt.Printf("hits=%d misses=%d\n", h, m)

    c.Close()
    c.Close()                            // safe, thanks to Once
    fmt.Println("closed twice without panicking")
}
```

Run it with `-race`. It must be clean.

Then break it three ways, one at a time, and run with `-race` after each. Change `hits` from an `atomic.Int64` to a plain `int64` and watch the detector fire. Change `RLock` to no lock at all in `Get`, same result. Finally remove the `sync.Once` from `Close` and call it twice: the second `close(c.done)` panics, which is the “closing twice” rule from the last chapter meeting a real program.

One design note worth absorbing: `Get` releases the read lock **before** checking expiry and updating counters. The counters are atomic so they need no lock, and holding a lock while doing work you do not have to is the habit that turns a cache into a bottleneck.

YOUR TURN

Add a `Len() int` method and decide which lock it takes. Then add a maximum size, evicting the entry closest to expiry when the cache is full. That second one is harder than it looks: finding the minimum requires scanning under the write lock, which makes `Set` O(n). Think about what you would need to make it fast, and notice you are now designing an LRU, which is why real cache libraries exist.

ASSIGNMENT

Build a cache, then find out which lock it wanted.

In `part2-ideas/15-sync`, write a concurrent cache with `Get` and `Set`.

One. A `sync.Mutex` version with the mutex embedded, and a test that hits it from fifty goroutines under `-race`.

Two. A `sync.RWMutex` version. Benchmark both at ninety percent reads:

```
go test -bench . -benchmem
```

Commit the numbers in `NOTES.md` and say which you would ship.

Three. A `sync.Once` that loads the cache the first time anybody asks, no matter how many ask at once.

Four. Replace the counter inside it with `atomic.Int64` and note what you gave up by doing so.

COMMIT AS

```
git commit -m "Sync - Assignment One"
```

16

Context: cancellation, deadlines, and request scope

You will see `ctx context.Context` as the first parameter of almost every function in a Go codebase, and it looks like noise until you understand what it buys. It is the done channel from chapter fourteen, standardised so that every library in the ecosystem agrees on how to be cancelled.

The problem it solves

A user opens a page, your handler starts a database query, and the user closes the tab. Without cancellation the query runs to completion, the connection stays busy, and the result is thrown away. Multiply by a thousand users on a slow page and the database falls over serving nobody.

Context is a signal that travels **down** the call stack so that work can stop.

Creating one

You rarely create a context from nothing. Two roots exist:

```
ctx := context.Background()   // the root, in main or a top-level handler
ctx := context.TODO()         // a placeholder: "a context belongs here"
```

`TODO` is for when you are threading context through an existing codebase and have not reached the caller yet. It behaves identically; it exists so that `grep TODO` finds the unfinished work.

From a root you **derive**:

```
ctx, cancel := context.WithCancel(parent)
ctx, cancel := context.WithTimeout(parent, 5*time.Second)
ctx, cancel := context.WithDeadline(parent, someTime)
```

Each returns a child and a `cancel` function.

WATCH OUT

Always call cancel, even on the timeout versions. It is not optional:

```
ctx, cancel := context.WithTimeout(ctx, 5*time.Second)
defer cancel()
```

Not calling it leaks the timer and the goroutine watching it until the deadline passes. `go vet` catches the obvious cases and not all of them. Writing `defer cancel()` on the very next line is the habit to build.

Using one

A context has a `Done()` channel that closes when the context is cancelled, and an `Err()` saying why.

```
func work(ctx context.Context) error {
    for i := 0; i < 1000; i++ {
        select {
        case <-ctx.Done():
            return ctx.Err()           // Canceled or DeadlineExceeded
        default:
        }
        doChunk(i)
    }
    return nil
}
```

That is the shape: check between units of work, return promptly when told.

For anything that blocks, put `ctx.Done()` in the `select` directly:

```
select {
case result := <-ch:
    return result, nil
case <-ctx.Done():
    return nil, ctx.Err()
}
```

IF YOU KNOW JAVASCRIPT

This is `AbortController` and `AbortSignal`. You create a controller, pass `signal` to `fetch`, and call `abort()`. Go's version has two differences: the convention of passing it as the **first parameter of every function** is near-universal, so the whole ecosystem supports it, and contexts form a tree where cancelling a parent cancels every descendant.

Cancellation is cooperative

Nothing is killed. A context cancellation sets a flag and closes a channel; code that does not check it keeps running.

```
func ignoresCancellation(ctx context.Context) {
    time.Sleep(10 * time.Second)    // runs the full ten seconds regardless
}
```

There are no interrupts and no thread killing in Go. If a library does not take a context, it cannot be cancelled, and that is a reason to prefer libraries that do.

The standard library already honours it

This is where it pays off, because you mostly pass a context rather than check one:

```
req, _ := http.NewRequestWithContext(ctx, "GET", url, nil)
resp, err := http.DefaultClient.Do(req)   // aborts if ctx is cancelled

rows, err := db.QueryContext(ctx, "SELECT ...")   // cancels the query
```

An incoming HTTP request already carries one, cancelled automatically when the client disconnects:

```
func handler(w http.ResponseWriter, r *http.Request) {
    // cancelled if the client goes away
    ctx := r.Context()
    rows, err := db.QueryContext(ctx, "SELECT ...")
    ...
}
```

So passing `r.Context()` into your service layer, which passes it to the database, gives you end-to-end cancellation for free. That is the single most valuable thing context does, and it requires no `select` anywhere in your code.

Timeouts compose

A derived context cannot outlive its parent, which gives you sane behaviour without arithmetic.

```
func handler(w http.ResponseWriter, r *http.Request) {
    // Overall budget for this request.
    ctx, cancel := context.WithTimeout(r.Context(), 3*time.Second)
    defer cancel()

    // One slow dependency gets a tighter budget.
    userCtx, cancel2 := context.WithTimeout(ctx, 500*time.Millisecond)
```

```
    defer cancel2()
    u, err := users.Get(userCtx, id)
    ...
}
```

If the request's three seconds elapse first, the inner context is cancelled too, even though its own 500ms had not expired.

Values, and why to be sparing

A context can carry request-scoped values.

```
type ctxKey string
const userKey ctxKey = "user"

ctx = context.WithValue(ctx, userKey, currentUser)

u, ok := ctx.Value(userKey).(*User)
```

Use an unexported custom key type, never a plain string, or two packages using `"user"` collide silently.

> **WATCH OUT**
> Context values are the most abused part of the package. They are untyped, invisible in a function signature, and found at runtime or not at all.
>
> The rule from the standard library documentation: use them for **request-scoped data that crosses API boundaries**, such as a request ID, an authenticated user, or a trace span. Never for optional parameters. If a function needs a value to do its job, that value is a parameter:
>
> ```
> // No.
> func Send(ctx context.Context) error    // reads the recipient from ctx
>
> // Yes.
> func Send(ctx context.Context, to string) error
> ```

The conventions

Four, and they are followed almost universally.

Context is the first parameter, named `ctx`. `func Get(ctx context.Context, id int)`.

Never store it in a struct. A context describes one call's lifetime; a struct outlives calls. The exception is `http.Request`, which has its own reasons.

Never pass nil. Use `context.TODO()` if you have nothing better.

Do not pass a context to something that outlives the request. Starting a background job with the request's context means the job is cancelled when the response is sent. Use `context.WithoutCancel(ctx)` to keep the values and drop the cancellation.

PROJECT 15: A SEARCH THAT RACES TWO SOURCES WITH A BUDGET

Query two backends at once, take whichever answers first, give up on both after a deadline, and never leak a goroutine. This is context, channels and select doing the job they exist for.

```
package main

import (
    "context"
    "errors"
    "fmt"
    "math/rand"
    "time"
)

type Result struct {
    Source string
    Hits   int
}

// search simulates a backend that takes an unpredictable amount of time
// and
// respects cancellation.
func search(ctx context.Context, source string,
    slow time.Duration) (Result, error) {
    select {
    case <-time.After(slow):
        return Result{Source: source, Hits: rand.Intn(100)}, nil
    case <-ctx.Done():
        // Stop promptly and say why.
        return Result{}, fmt.Errorf("%s: %w", source, ctx.Err())
    }
}

// first returns whichever source answers first, cancelling the other.
func first(ctx context.Context, budget time.Duration) (Result, error) {
    ctx, cancel := context.WithTimeout(ctx, budget)
    defer cancel()              // cancels the loser, and the timer

    type outcome struct {
        res Result
        err error
    }
    // Buffered with room for both, so the loser's send never blocks and
    // its goroutine always exits. With an unbuffered channel the loser
    // would block forever on the send: a goroutine leak.
```

```
    ch := make(chan outcome, 2)

    go func() {
        r, err := search(ctx, "index-a", 300*time.Millisecond)
        ch <- outcome{r, err}
    }()
    go func() {
        r, err := search(ctx, "index-b", 120*time.Millisecond)
        ch <- outcome{r, err}
    }()

    var lastErr error
    for i := 0; i < 2; i++ {
        select {
        case o := <-ch:
            if o.err == nil {
                return o.res, nil      // first success wins
            }
            lastErr = o.err
        case <-ctx.Done():
            return Result{}, ctx.Err()
        }
    }
    return Result{}, lastErr
}

func main() {
    // Generous budget: the fast index wins.
    r, err := first(context.Background(), time.Second)
    fmt.Printf("got %+v err=%v\n", r, err)

    // Tight budget: both are cancelled.
    r, err = first(context.Background(), 50*time.Millisecond)
    fmt.Printf("got %+v err=%v\n", r, err)
    fmt.Println("deadline:",
        errors.Is(err, context.DeadlineExceeded))
}
```

Three things this program teaches.

The buffered channel is a correctness requirement, not an optimisation. When `first` returns after the winner, the loser still sends. With no buffer that send blocks forever and the goroutine never exits. Change the `2` to a `0`, run with `-race` and then inspect goroutine counts, and you have reproduced one of the most common leaks in Go services.

The `defer cancel()` is what makes the loser stop early rather than running its full three hundred milliseconds into a void.

And `errors.Is(err, context.DeadlineExceeded)` works through the `%w` wrap in `search`, which is chapter eleven's design paying off: the caller can tell a timeout from any other failure.

YOUR TURN

Add a third source and make `first` return as soon as **two** of the three agree, rather than on the first answer. Then add a `context.WithoutCancel` branch that logs the losing results to an audit sink after the winner has been returned, and confirm that work still completes after `first` has returned. That is the pattern for "respond now, finish the bookkeeping afterwards", and getting the context wrong there is why audit records go missing under load.

ASSIGNMENT

Make a cancellation actually reach the bottom of a call stack.

In `part2-ideas/16-context`, write a handler that calls a service that calls a slow HTTP endpoint.

One. Thread one `context.Context` from the handler to the request. Use `http.NewRequestWithContext`, not the plain one.

Two. Give the handler a two second timeout and point it at a server that takes five. Show the error chain reporting the deadline.

Three. Write a worker loop that honours cancellation with a `select` on `ctx.Done()`, and one that ignores it. Cancel both and time how long each takes to stop.

Four. Put a request ID in the context with a typed key, read it two layers down, and write one sentence on why the key is not a string.

COMMIT AS

```
git commit -m "Context - Assignment One"
```

17

Generics, briefly and honestly

Go got generics in 1.18, after a decade of arguing about whether it should. The result is deliberately less powerful than TypeScript's, and the community consensus is that you should use them rarely. This chapter covers enough to read other people's code and to recognise the handful of cases where writing your own is right.

The syntax

Type parameters go in square brackets before the arguments.

```
func First[T any](items []T) (T, bool) {
    var zero T
    if len(items) == 0 {
        return zero, false
    }
    return items[0], true
}

n, ok := First([]int{1, 2, 3})        // T inferred as int
s, ok := First([]string{"a", "b"})    // T inferred as string
```

`T` is a type parameter and `any` is its **constraint**, meaning "any type at all". Inference usually works, so you rarely write `First[int](...)`.

Note `var zero T`. You cannot return `nil` for an arbitrary `T`, because `T` might be `int`. The zero value is the only thing you can produce without knowing the type, and this little declaration appears in almost every generic function.

> **IF YOU KNOW JAVASCRIPT**
> TypeScript's `function first<T>(items: T[]): T | undefined` is the same idea, with a much more capable type system behind it. TypeScript has conditional types, mapped types, template literal types and inference that borders on computation. Go has type parameters and constraints, and that is the lot. If you arrive expecting the TypeScript toolkit you will be disappointed, which is by design: the proposal was accepted on the promise that it would not make Go code hard to read.

Constraints are interfaces

A constraint says what you may **do** with a value of that type.

```
// Will not compile: you cannot add two values of an arbitrary type.
func Sum[T any](items []T) T {
    var total T
    for _, v := range items {
        // invalid operation: operator + not defined on total
        total += v
    }
    return total
}
```

You must constrain `T` to types that support `+`. Constraints are written as interfaces containing **type sets**:

```
type Number interface {
    ~int | ~int64 | ~float64
}

func Sum[T Number](items []T) T {
    var total T
    for _, v := range items {
        total += v
    }
    return total
}

Sum([]int{1, 2, 3})          // 6
Sum([]float64{1.5, 2.5})     // 4.0
```

The `~` means "any type whose **underlying** type is this", so a named type still qualifies:

```
type Celsius float64
Sum([]Celsius{20.5, 21.0})   // works because of ~float64
```

Without the tilde, `Celsius` would be rejected, which is almost never what you want. Write `~` unless you have a reason not to.

The constraints you will actually use

Two come from the standard library and cover most cases.

```
import "cmp"

func Max[T cmp.Ordered](a, b T) T {    // anything with < and >
    if a > b { return a }
```

```
    return b
}
```

`cmp.Ordered` is every type supporting `<`, which is the numeric types and `string`. `comparable` is built in and means "supports `==`", which is what map keys require:

```
func Keys[K comparable, V any](m map[K]V) []K {
    out := make([]K, 0, len(m))
    for k := range m {
        out = append(out, k)
    }
    return out
}
```

Generic types

Types can take parameters too.

```
type Stack[T any] struct {
    items []T
}

func (s *Stack[T]) Push(v T) {
    s.items = append(s.items, v)
}

func (s *Stack[T]) Pop() (T, bool) {
    var zero T
    if len(s.items) == 0 {
        return zero, false
    }
    v := s.items[len(s.items)-1]
    s.items = s.items[:len(s.items)-1]
    return v, true
}

s := &Stack[string]{}
s.Push("a")
v, ok := s.Pop()
```

WATCH OUT

Methods cannot introduce new type parameters. This compiles nowhere:

```
func (s *Stack[T]) Map[U any](f func(T) U) *Stack[U] {   // illegal
```

If you want a `Map` that changes the element type, it has to be a package function, not a method:

```
func MapStack[T, U any](s *Stack[T], f func(T) U) *Stack[U] { ... }
```

This single restriction is why Go will never have fluent chained collection methods like `items.map(...).filter(...)`. It was a deliberate choice to keep the type checker simple, and it is the main reason generics changed Go code far less than people expected.

What the standard library gives you

Since 1.21 there are generic helpers, and you should use these rather than write your own.

```
import ("slices"; "maps")

slices.Contains(s, v)        // bool
slices.Index(s, v)           // int, -1 if absent
slices.Sort(s)               // for cmp.Ordered
slices.SortFunc(s, cmp)      // for anything
slices.Reverse(s)
slices.Max(s); slices.Min(s)
slices.Clone(s)
slices.Equal(a, b)

maps.Keys(m)                 // an iterator, not a slice
maps.Values(m)
maps.Clone(m)
```

WATCH OUT

`maps.Keys` returns an **iterator** (`iter.Seq[K]`), not a slice. To get a slice, collect it:

```
keys := slices.Collect(maps.Keys(m))
```

This trips up anybody reading older blog posts, where an earlier experimental version did return a slice.

When to write your own, and when not to

The honest guidance, which matches what the Go team says: **write the concrete version first**. Write the generic version when you have written the same function for the third type and the bodies are identical.

Good reasons:

- A container: a stack, a set, a typed cache.

- A function over any slice or map where the body does not care about the type.
- Removing an `any` and the type assertions that came with it.

Bad reasons:

- "It might be reused later." It usually is not.
- To avoid writing a four-line loop. The loop is clearer.
- To model a hierarchy. Go has interfaces for that, and they are better at it.

> **IF YOU KNOW JAVASCRIPT**
> The decisive difference from TypeScript: TypeScript's generics are erased at runtime and cost nothing, so using them liberally is free. Go's are compiled, and a generic function is often **slower** than the concrete one because of how the compiler shares instantiations for pointer-shaped types. If you are reaching for generics for performance, measure. The usual answer is that the concrete function wins.

Interfaces are still usually the answer

The mistake to avoid is reaching for a type parameter where an interface does the job better.

```
// Generic: the caller must have a concrete type, and the body can only do
// what the constraint allows.
func PrintAll[T fmt.Stringer](items []T) {
    for _, v := range items {
        fmt.Println(v.String())
    }
}

// Interface: accepts a mixed slice of different types. Usually better.
func PrintAll(items []fmt.Stringer) {
    for _, v := range items {
        fmt.Println(v.String())
    }
}
```

The rule of thumb: if the function's behaviour depends only on **methods**, use an interface. Use a type parameter when you need the **actual type** back out, which is what interfaces cannot do.

> **PROJECT 16: A TYPED RESULT CACHE**
> A small generic container that is a genuinely good use of type parameters: the body does not care what it stores, and the caller gets its type back with no assertion.

```
package main

import (
    "fmt"
    "sync"
    "time"
)

type cached[V any] struct {
    value   V
    expires time.Time
}

// Cache is generic over both key and value. K must be comparable because
// it is a map key; V can be anything at all.
type Cache[K comparable, V any] struct {
    mu    sync.RWMutex
    items map[K]cached[V]
    ttl   time.Duration
}

func NewCache[K comparable, V any](ttl time.Duration) *Cache[K, V] {
    return &Cache[K, V]{items: map[K]cached[V]{}, ttl: ttl}
}

func (c *Cache[K, V]) Set(k K, v V) {
    c.mu.Lock()
    defer c.mu.Unlock()
    c.items[k] = cached[V]{value: v, expires: time.Now().Add(c.ttl)}
}

func (c *Cache[K, V]) Get(k K) (V, bool) {
    c.mu.RLock()
    e, ok := c.items[k]
    c.mu.RUnlock()

    var zero V
    if !ok || time.Now().After(e.expires) {
        return zero, false
    }
    return e.value, true
}

// GetOrLoad is where the design earns itself: the loader returns a V and
// the caller receives a V, with no any and no type assertion anywhere.
func (c *Cache[K, V]) GetOrLoad(k K,
    load func(K) (V, error)) (V, error) {
    if v, ok := c.Get(k); ok {
        return v, nil
    }
    v, err := load(k)
```

```
    if err != nil {
        var zero V
        return zero, err
    }
    c.Set(k, v)
    return v, nil
}

type User struct {
    ID   int
    Name string
}

func main() {
    users := NewCache[int, User](time.Minute)
    counts := NewCache[string, int](time.Minute)

    calls := 0
    loadUser := func(id int) (User, error) {
        calls++
        return User{ID: id, Name: fmt.Sprintf("user-%d", id)}, nil
    }

    u, _ := users.GetOrLoad(1, loadUser)
    fmt.Println(u.Name, "loader calls:", calls)   // user-1  1

    u, _ = users.GetOrLoad(1, loadUser)
    fmt.Println(u.Name, "loader calls:", calls)   // user-1  1 (cached)

    // u is a User, not an any. This compiles, and would not if the cache
    // stored any.
    fmt.Println("id plus one:", u.ID+1)

    counts.Set("hits", 42)
    n, _ := counts.Get("hits")
    fmt.Println("count doubled:", n*2)            // 84, n is an int
}
```

Now prove the value. Rewrite `Cache` with `map[any]any` and no type parameters, then try `u.ID+1`: you need `u.(User).ID+1`, an assertion that can panic, and the compiler no longer checks that you put a `User` in and got a `User` out. That assertion, repeated at every call site, is exactly what generics removed.

Then try to add a `Map` method that converts a `Cache[K, V]` into a `Cache[K, W]` and watch the compiler refuse, because methods cannot introduce type parameters. Make it a package function instead and notice how that constraint pushes Go code away from fluent chaining.

YOUR TURN

Add a `Keys() []K` method, then write a package-level `Filter[K comparable, V any](c *Cache[K, V], keep func(V) bool) []V`. Then stop and ask whether either of them earns its place, or whether the caller ranging over a plain map would have been clearer. That question, asked honestly every time, is the whole discipline of generics in Go.

ASSIGNMENT

Write the generic versions of what you already wrote, then judge them.

In `part2-ideas/17-generics`:

One. Rewrite chapter six's `Filter`, `Chunk` and `Dedupe` to work for any element type. `Dedupe` needs `comparable`, which is the constraint doing real work.

Two. A generic `Set[T comparable]` with `Add`, `Has` and `Items`.

Three. A `Map[T, U any]` that changes the element type, and one use of it that would have needed `any` and a cast before.

Four. Now the honest part. In `NOTES.md`, name one place in your own code where an interface is the better answer than a type parameter, and say why. A chapter on generics that does not leave you more careful has failed.

COMMIT AS

```
git commit -m "Generics - Assignment One"
```

18

Testing: the runner is already installed

Go ships a test runner, an assertion-free testing philosophy, a benchmark harness, a coverage tool, a fuzzer and a race detector. There is nothing to install and nothing to configure. After a JavaScript project's testing setup, this takes some adjusting to, mostly because you keep looking for the config file.

The rules

A test file ends in `_test.go`, sits beside the code it tests, and contains functions named `TestXxx` taking `*testing.T`.

```
// math.go
package math

func Add(a, b int) int { return a + b }
```

```
// math_test.go
package math

import "testing"

func TestAdd(t *testing.T) {
    got := Add(2, 3)
    if got != 5 {
        t.Errorf("Add(2, 3) = %d, want 5", got)
    }
}
```

```
go test ./...
```

```
ok      github.com/you/myapp/math    0.002s
```

> **IF YOU KNOW JAVASCRIPT**
> No `describe`, no `it`, no `expect`. A test is a function, and it fails by calling `t.Errorf`. There is no assertion library in the standard library, and this is deliberate: the Go team's

> position is that `if got != want` is clearer than learning which of `toBe`, `toEqual` and `toStrictEqual` you wanted.
>
> You will still meet `testify` in real codebases, which adds `assert.Equal(t, want, got)`. It is popular and it is fine. Start without it.

Error messages are the whole skill

A failing test must tell you what happened without opening the source. The convention is `got, want`:

```
t.Errorf("Add(2, 3) = %d, want 5", got)
```

```
    math_test.go:8: Add(2, 3) = 6, want 5
```

That line names the input, the result and the expectation. Compare it to `t.Error("failed")`, which tells you nothing you did not already know.

Errorf against Fatalf

`t.Errorf` records a failure and **keeps going**. `t.Fatalf` records it and stops the test function immediately.

```
u, err := Load(1)
if err != nil {
    t.Fatalf("Load(1): unexpected error: %v", err)   // stop: u is unusable
}
if u.Name != "Ada" {
    t.Errorf("Name = %q, want Ada", u.Name)          // keep checking
}
```

Use `Fatalf` when continuing would panic or produce noise. Use `Errorf` when you want to see every failure in one run.

Table-driven tests

This is the dominant Go testing idiom. One test function, a slice of cases.

```
func TestSlugify(t *testing.T) {
    tests := []struct {
        name string
        in   string
        want string
    }{
        {"simple", "Hello World", "hello-world"},
        {"punctuation", "Hello, World!", "hello-world"},
```

```
        {"already a slug", "hello-world", "hello-world"},
        {"leading space", "  padded  ", "padded"},
        {"empty", "", ""},
        {"unicode", "Café Noir", "cafe-noir"},
    }

    for _, tt := range tests {
        t.Run(tt.name, func(t *testing.T) {
            if got := Slugify(tt.in); got != tt.want {
                t.Errorf("Slugify(%q) = %q, want %q", tt.in, got, tt.want)
            }
        })
    }
}
```

`t.Run` creates a subtest with its own name, so output identifies the case:

```
--- FAIL: TestSlugify (0.00s)
    --- FAIL: TestSlugify/unicode (0.00s)
        slug_test.go:21: Slugify("Café Noir") =
            "caf-noir", want "cafe-noir"
```

You can run one case directly:

```
go test -run 'TestSlugify/unicode' ./...
```

> **IF YOU KNOW JAVASCRIPT**
> This replaces `describe.each` and `test.each`. The advantage over both is that the cases are an ordinary slice of an ordinary struct, so adding a field to the table is a normal code change and the compiler checks every row.

Comparing things that are not numbers

`==` works on structs of comparable fields. For slices, maps and nested structs, use `reflect.DeepEqual` or, much better, `go-cmp`:

```
import "github.com/google/go-cmp/cmp"

if diff := cmp.Diff(want, got); diff != "" {
    t.Errorf("Load() mismatch (-want +got):\n%s", diff)
}
```

```
    user_test.go:31: Load() mismatch (-want +got):
          user.User{
                ID:    1,
```

```
        -       Name:  "Ada",
        +       Name:  "ada",
                Email: "ada@example.com",
          }
```

A readable diff instead of `false`. This is the one testing dependency worth adding on day one.

Helpers and t.Helper

A helper that reports failures should call `t.Helper()` so the line number points at the **caller**:

```
func mustLoad(t *testing.T, id int) *User {
    t.Helper()                        // failures blame the caller's line
    u, err := Load(id)
    if err != nil {
        t.Fatalf("Load(%d): %v", id, err)
    }
    return u
}
```

Without it, every failure points at the line inside `mustLoad`, which is useless when ten tests call it.

Setup and cleanup

`t.Cleanup` registers work to run when the test finishes, including on failure.

```
func newTestDB(t *testing.T) *sql.DB {
    t.Helper()
    db, err := sql.Open("sqlite", ":memory:")
    if err != nil {
        t.Fatal(err)
    }
    t.Cleanup(func() { db.Close() })    // runs even if the test fails
    return db
}
```

For package-wide setup, `TestMain`:

```
func TestMain(m *testing.M) {
    // before all tests
    code := m.Run()
    // after all tests
    os.Exit(code)
}
```

> **IF YOU KNOW JAVASCRIPT**
> `t.Cleanup` is `afterEach`, scoped to one test and registered where the resource is created rather than in a separate block. `TestMain` is `globalSetup` and `globalTeardown`. There is no `beforeEach`: you call a helper at the top of each test, which is more typing and means a reader can see what a test depends on without scrolling.

Parallel tests

```
func TestSomething(t *testing.T) {
    t.Parallel()
    // ...
}
```

Tests in different packages already run in parallel. `t.Parallel()` marks tests **within** a package as safe to interleave.

> **WATCH OUT**
> A parallel test must not share mutable state with another test. Writing to the same database table, the same temp file or the same package-level variable will fail intermittently, which is worse than failing reliably.
>
> Combine `t.Parallel()` with `-race` so that a shared-state mistake is reported rather than merely flaky.

Testing errors

Test the **kind**, not the message. Message text is not a contract.

```
// Brittle: breaks when someone improves the wording.
if err.Error() != "user not found" { ... }

// Correct.
if !errors.Is(err, ErrNotFound) {
    t.Errorf("got %v, want ErrNotFound", err)
}
```

And always test that an error **is** returned when you expect one:

```
{"missing email", "", "", true},   // a wantErr field in the table
...
_, err := Create(tt.name, tt.email)
if (err != nil) != tt.wantErr {
    t.Errorf("Create() error = %v, wantErr %v", err, tt.wantErr)
}
```

Coverage

```
go test -cover ./...
go test -coverprofile=cover.out ./... && go tool cover -html=cover.out
```

The second opens a browser showing every line in green or red. It is the fastest way to find the branch nobody tested.

> **WATCH OUT**
> Coverage measures lines executed, not behaviour verified. A test that calls every function and asserts nothing reports high coverage and proves nothing. Use it to find what you **missed**, never as a target to hit.

Benchmarks

```
func BenchmarkSlugify(b *testing.B) {
    for b.Loop() {
        Slugify("Hello, World! This is a title")
    }
}
```

```
go test -bench=. -benchmem ./...
```

```
BenchmarkSlugify-8    2419371    489.2 ns/op    144 B/op    3 allocs/op
```

Operations per second, nanoseconds each, bytes allocated and allocation count. `b.Loop()` replaced the older `for i := 0; i < b.N; i++` in Go 1.24 and is harder to get wrong, because it stops the compiler optimising your work away.

> **PROJECT 17: TEST THE USER STORE PROPERLY**
> Take the store from chapter eleven and give it a real test suite: table driven, testing error kinds rather than messages, with a helper and cleanup.
>
> ```
> package store
>
> import (
>     "errors"
>     "testing"
> )
>
> func newStore(t *testing.T) *Store {
>     t.Helper()
>     s := NewStore()
>     t.Cleanup(func() {
> ```

```
        // Nothing to close here, but the hook is where a real test would
        // drop a temp database. Wire it now so it is not retrofitted
        // later.
    })
    return s
}

func TestCreate(t *testing.T) {
    tests := []struct {
        name    string
        user    string
        email   string
        wantErr error       // nil, a sentinel, or errSomeValidation
        field   string      // for validation errors
    }{
        {"valid", "Ada", "ada@example.com", nil, ""},
        {"empty name", "", "x@example.com", errValidation, "name"},
        {"bad email", "Ada", "not-an-email", errValidation, "email"},
    }

    for _, tt := range tests {
        t.Run(tt.name, func(t *testing.T) {
            t.Parallel()
            s := newStore(t)

            u, err := s.Create(tt.user, tt.email)

            if tt.wantErr == nil {
                if err != nil {
                    t.Fatalf("Create(%q, %q): unexpected error: %v",
                        tt.user, tt.email, err)
                }
                if u.Name != tt.user {
                    t.Errorf("Name = %q, want %q", u.Name, tt.user)
                }
                if u.ID == 0 {
                    t.Error("ID was not assigned")
                }
                return
            }

            if err == nil {
                t.Fatalf("Create(%q, %q): want error, got nil",
                    tt.user, tt.email)
            }
            var ve *ValidationError
            if !errors.As(err, &ve) {
                t.Fatalf("got %v, want a *ValidationError", err)
            }
            if ve.Field != tt.field {
                t.Errorf("Field = %q, want %q", ve.Field, tt.field)
```

```
            }
        })
    }
}

func TestCreateDuplicateEmail(t *testing.T) {
    t.Parallel()
    s := newStore(t)

    if _, err := s.Create("Ada", "ada@example.com"); err != nil {
        t.Fatalf("first create: %v", err)
    }
    _, err := s.Create("Grace", "ada@example.com")

    // The identity of the failure is the contract. The wording is not.
    if !errors.Is(err, ErrConflict) {
        t.Errorf("got %v, want ErrConflict", err)
    }
}

func TestGetMissing(t *testing.T) {
    t.Parallel()
    s := newStore(t)

    if _, err := s.Get(999); !errors.Is(err, ErrNotFound) {
        t.Errorf("got %v, want ErrNotFound", err)
    }
}
```

Run it four ways and notice what each tells you.

`go test ./...` gives you pass or fail. `go test -v ./...` names every subtest, which is how you confirm the table ran every row rather than silently skipping. `go test -race ./...` would catch the shared-state bug if `newStore` returned a package-level store instead of a fresh one, which is the mistake `t.Parallel()` makes dangerous. And `go test -cover ./...` tells you which branch of `Create` you forgot.

The important design point is in `TestCreateDuplicateEmail`: it asserts `errors.Is(err, ErrConflict)` and never looks at the message. Change the wording of that error and the test still passes, which is correct, because the wording was never the promise.

YOUR TURN

Add a benchmark for `Create` and run it with `-benchmem`. Then change the duplicate-email check from a linear scan to a second map keyed by email, and run the benchmark again with a thousand existing users. The allocation count and the nanoseconds per

operation will both move, and seeing a design change show up as a number is the habit this chapter is really teaching.

ASSIGNMENT

Go back and test something you already wrote.

Take your chapter eleven `store` package and bring it to a state you would defend in review.

One. Table-driven tests for every exported function, including the error cases, with the table naming each case.

Two. A failure message that says what it wanted and what it got. Break one test on purpose and check the output names the case.

Three. A `t.Helper()` assertion helper, and `t.Cleanup` for anything the test created.

Four. Coverage, recorded rather than chased:

```
go test -cover ./... | tee COVERAGE.txt
```

Five. One benchmark, with `-benchmem`, and one sentence on whether the allocations surprised you.

COMMIT AS

```
git commit -m "Testing - Assignment One"
```

19

JSON, struct tags, and the capital letter that bites

JSON is where your Go backend meets your React frontend, so this chapter earns its place twice: once now, and again in part four when Grit generates the types on both sides of that boundary.

Encoding and decoding

```
type User struct {
    ID    int    `json:"id"`
    Name  string `json:"name"`
    Email string `json:"email"`
}

u := User{ID: 1, Name: "Ada", Email: "ada@example.com"}

data, err := json.Marshal(u)
// {"id":1,"name":"Ada","email":"ada@example.com"}

var back User
err = json.Unmarshal(data, &back)     // note the &
```

`Unmarshal` takes a pointer, because it fills in the value you give it. Passing `back` instead of `&back` compiles and silently does nothing useful, which is the chapter's first trap.

For readable output:

```
data, _ := json.MarshalIndent(u, "", "  ")
```

The capital letter rule, again

This is the single most common Go and JSON bug, and you met the rule in chapter two.

```
type User struct {
    id    int       // lowercase: unexported
    name  string
}
```

```
json.Marshal(User{id: 1, name: "Ada"})   // {}
```

An empty object. No error, no warning. The `encoding/json` package lives in another package and literally cannot see unexported fields.

> **WATCH OUT**
> **If your JSON comes out as `{}`, check for lowercase field names first.** It is the answer about nine times out of ten. The same applies on the way in: an unexported field is never populated by `Unmarshal`.

Struct tags in full

```
type Product struct {
    ID        int       `json:"id"`
    Name      string    `json:"name"`
    PriceCents int      `json:"price_cents"`  // money is never a float
    Notes     string    `json:"notes,omitempty"`   // omitted when ""
    Internal  string    `json:"-"`                 // never encoded
    CreatedAt time.Time `json:"created_at"`
    Count     int       `json:"count,string"`      // encoded as "42"
}
```

`json:"name"`	Use this key
`json:"-"`	Never encode or decode this field
`json:",omitempty"`	Omit when the value is the zero value
`json:",string"`	Encode a number or bool as a JSON string
no tag	Use the Go field name, capital included

> **WATCH OUT**
> The tag syntax is unforgiving and silent. All of these are wrong and none produce an error:
>
> ```
> Name string `json: "name"`     // space after the colon: tag ignored
> Name string `json:'name'`      // single quotes: tag ignored
> Name string "json:\"name\""    // works, but use backticks
> ```
>
> The field then encodes as `Name`. If one key in your JSON is capitalised and the rest are not, you have found a stray space.
>
> `go vet` catches malformed tags. Run it.

omitempty, and what it actually means

`omitempty` omits the **zero value**, which is rarely what people assume.

```
type Filter struct {
    Page     int    `json:"page,omitempty"`      // 0 is omitted
    Active   bool   `json:"active,omitempty"`    // false is omitted!
    Query    string `json:"query,omitempty"`     // "" is omitted
}
```

`Active: false` disappears from the output. If false is a meaningful value and not merely "unset", `omitempty` is wrong and will cause a real bug.

The fix is a pointer, which is chapter eight's lesson in its natural habitat:

```
type Filter struct {
    Active *bool `json:"active,omitempty"`   // nil omitted, &false encoded
}
```

> **IF YOU KNOW JAVASCRIPT**
> JavaScript has no such ambiguity because `undefined` and `false` are different values. Go's zero values collapse them, so a `*bool` is how you get the distinction back. You will see `*string` and `*bool` all over the struct definitions of any API that supports partial updates, and now you know exactly why.

Decoding unknown shapes

Without a struct, JSON decodes into `map[string]any`:

```
var data map[string]any
json.Unmarshal(raw, &data)

name := data["name"].(string)       // panics if it is not a string
name, ok := data["name"].(string)   // safe
```

Every number becomes a `float64`, which surprises everyone:

```
json.Unmarshal([]byte(`{"count": 42}`), &data)
n := data["count"].(int)        // PANIC: it is a float64
n := int(data["count"].(float64))   // correct
```

Prefer a struct whenever you know the shape. The type system is the point.

Decoding is case-insensitive, and lenient

Two behaviours that catch people:

```
type User struct {
    Name string `json:"name"`
}

json.Unmarshal([]byte(`{"NAME": "Ada"}`), &u)   // works: case-insensitive
// works: unknown fields ignored
json.Unmarshal([]byte(`{"nope": 1}`), &u)
```

Unknown fields are silently dropped. For an API that should reject nonsense, use a decoder:

```
dec := json.NewDecoder(r.Body)
dec.DisallowUnknownFields()
if err := dec.Decode(&input); err != nil {
    http.Error(w, "bad request: "+err.Error(), http.StatusBadRequest)
    return
}
```

This turns a typo in a client's payload into a 400 instead of a silently ignored field, which is usually what you want for a write endpoint.

Streaming with Encoder and Decoder

For an HTTP body, do not read it all into memory first:

```
func handler(w http.ResponseWriter, r *http.Request) {
    var input CreateUser
    if err := json.NewDecoder(r.Body).Decode(&input); err != nil {
        http.Error(w, "invalid json", http.StatusBadRequest)
        return
    }

    w.Header().Set("Content-Type", "application/json")
    json.NewEncoder(w).Encode(response)
}
```

`Decoder` reads from any `io.Reader` and `Encoder` writes to any `io.Writer`, which is chapter ten's small interfaces paying off again.

Times, and the format you will meet

`time.Time` marshals to RFC 3339:

```
{"created_at": "2026-10-20T14:30:00Z"}
```

That parses directly with `new Date(...)` in JavaScript, so the default is the right one. A zero time encodes as `"0001-01-01T00:00:00Z"`, which looks alarming in output; use a `*time.Time` with `omitempty` when the field is genuinely optional.

Custom marshalling

Implement `MarshalJSON` to control a type's representation:

```
type Money int64      // stored as cents

func (m Money) MarshalJSON() ([]byte, error) {
    return []byte(fmt.Sprintf(`"%d.%02d"`, m/100, m%100)), nil
}

func (m *Money) UnmarshalJSON(data []byte) error {
    var s string
    if err := json.Unmarshal(data, &s); err != nil {
        return err
    }
    // parse "12.34" into 1234 cents
    ...
}
```

Note the receivers: `MarshalJSON` on the value, `UnmarshalJSON` on the pointer, because unmarshalling has to modify. Getting this backwards means your custom decoder is silently never called, which is the second-most-common JSON bug after lowercase fields.

IF YOU KNOW JAVASCRIPT

This is `toJSON()` and a reviver function, combined into an interface. Money is the classic case: a float64 cannot represent 0.10 exactly, so storing cents as an integer and formatting at the boundary is the correct design in any language. Grit has a `money` field type that does exactly this, and part four explains why it is a type rather than advice.

PROJECT 18: A JSON REFORMATTING CLI

Build `jsonfmt`: read JSON on standard input or from a file, validate it, pretty print it, and optionally extract one field. This is a tool you will actually use, and it exercises streaming, unknown shapes and error handling.

```
package main

import (
    "encoding/json"
    "flag"
```

```
    "fmt"
    "io"
    "os"
    "strings"
)

func main() {
    indent := flag.String("indent", "  ", "indent string")
    compact := flag.Bool("c", false,
        "compact output instead of indented")
    path := flag.String("get", "",
        "dotted path to extract, e.g. user.name")
    flag.Parse()

    // Read from a named file, or standard input when piped.
    var r io.Reader = os.Stdin
    if args := flag.Args(); len(args) > 0 {
        f, err := os.Open(args[0])
        if err != nil {
            fail("opening %s: %v", args[0], err)
        }
        defer f.Close()
        r = f
    }

    // any, not map[string]any: the document might be an array.
    var doc any
    dec := json.NewDecoder(r)
    dec.UseNumber()            // keep numbers exact instead of float64
    if err := dec.Decode(&doc); err != nil {
        fail("invalid json: %v", err)
    }

    if *path != "" {
        v, err := get(doc, *path)
        if err != nil {
            fail("%v", err)
        }
        doc = v
    }

    enc := json.NewEncoder(os.Stdout)
    if !*compact {
        enc.SetIndent("", *indent)
    }
    if err := enc.Encode(doc); err != nil {
        fail("writing: %v", err)
    }
}

// get walks a dotted path through decoded JSON.
```

```
func get(doc any, path string) (any, error) {
    cur := doc
    for _, part := range strings.Split(path, ".") {
        m, ok := cur.(map[string]any)
        if !ok {
            return nil, fmt.Errorf("%q: not an object at %q", path, part)
        }
        cur, ok = m[part]
        if !ok {
            return nil, fmt.Errorf("%q: no such key %q", path, part)
        }
    }
    return cur, nil
}

func fail(format string, args ...any) {
    fmt.Fprintf(os.Stderr, "jsonfmt: "+format+"\n", args...)
    os.Exit(1)
}
```

Build it and try it:

```
go build -o jsonfmt .
echo '{"user":{"name":"Ada","id":1},"tags":["a","b"]}' | ./jsonfmt
echo '{"user":{"name":"Ada"}}' | ./jsonfmt -get user.name
echo '{"bad": }' | ./jsonfmt           # exits 1 with a message on stderr
```

Two details worth understanding. `dec.UseNumber()` stops every number becoming a `float64`, so a large integer ID survives the round trip instead of losing precision: without it, `{"id": 12345678901234567}` comes back subtly wrong. And `doc` is declared as `any` rather than `map[string]any`, so a top-level JSON array works too.

The `...any` in `fail` is a variadic parameter, Go's rest argument. Passing it on to `Fprintf` with `args...` spreads it again, which is the Go equivalent of `...args` in both directions.

YOUR TURN

Add a `-keys` flag that prints the top-level keys one per line, sorted. You will need a type switch on `doc`, because the document might be an object or an array, and the two cases want different answers. Then add `DisallowUnknownFields` behind a `-strict` flag and find out what it does to a document with no struct to compare against. The answer tells you something real about what that option is actually for.

ASSIGNMENT

Round-trip something awkward, because the easy cases never bite.

In `part2-ideas/19-json`:

One. Marshal your chapter five `Invoice`, then unmarshal it back and compare. Any field that does not survive is a missing tag or a lowercase name.

Two. The `omitempty` trap: a `bool` field that is deliberately `false`. Show it disappearing, then fix it with a pointer, and write in a comment which choice you would make in an API.

Three. Decode into `map[string]any` for a payload whose shape you do not control, and pull one nested value out with a type assertion that does not panic on the wrong type.

Four. A custom `MarshalJSON` for a type that must serialise as a date rather than a timestamp, with a test for both directions.

COMMIT AS

```
git commit -m "JSON - Assignment One"
```

20

Files, streams and the io interfaces

Two interfaces carry most of Go's I/O, and you met them in chapter ten:

```
type Reader interface { Read(p []byte) (n int, err error) }
type Writer interface { Write(p []byte) (n int, err error) }
```

One method each. Files, network connections, HTTP bodies, gzip streams, in-memory buffers, standard input and standard output all satisfy them, which means a function written against `io.Reader` works with every one of those without knowing which it has.

Reading a whole file

When the file is small and you want it all:

```
data, err := os.ReadFile("config.json")   // []byte
if err != nil {
    return fmt.Errorf("reading config: %w", err)
}
text := string(data)
```

And writing:

```
err := os.WriteFile("out.txt", []byte("hello"), 0o644)
```

`0o644` is the Unix permission mode: owner read and write, everyone else read. On Windows it is mostly ignored. `0o600` for anything secret.

> **WATCH OUT**
> `os.ReadFile` loads the entire file into memory. For a 2GB log file that is 2GB of RAM, and your process is killed. Use it for configuration and small documents; stream anything that could be large, which is the rest of this chapter.

Streaming line by line

```
f, err := os.Open("access.log")
if err != nil {
```

```
    return err
}
defer f.Close()

scanner := bufio.NewScanner(f)
for scanner.Scan() {
    line := scanner.Text()
    // ...
}
if err := scanner.Err(); err != nil {
    return fmt.Errorf("reading access.log: %w", err)
}
```

Memory stays constant regardless of file size.

WATCH OUT

You must check `scanner.Err()`. The loop ends identically whether the file finished or the disk failed, and without the check a read error looks exactly like a successful end. This is the most commonly skipped error check in Go.

Second trap: `Scanner` refuses lines longer than 64KB by default, returning `bufio.Scanner: token too long`. For long lines, enlarge the buffer:

```
scanner.Buffer(make([]byte, 0, 1024*1024), 1024*1024)   // 1MB lines
```

os.Open against os.Create

`os.Open`	Read only. Fails if the file is missing
`os.Create`	Write only. Creates, or truncates an existing file to empty
`os.OpenFile`	Everything else: append, read-write, custom permissions

`os.Create` truncating is worth remembering, because it means opening a file to "check something" with `Create` destroys it. To append:

```
f, err := os.OpenFile("app.log",
    os.O_APPEND|os.O_CREATE|os.O_WRONLY, 0o644)
```

Buffer your writes

Every `Write` on a file is a system call. Writing a million lines one at a time is a million system calls.

```
f, err := os.Create("out.txt")
if err != nil {
    return err
}
defer f.Close()

w := bufio.NewWriter(f)
defer w.Flush()            // without this, you lose the tail of the file

for _, line := range lines {
    fmt.Fprintln(w, line)
}
```

WATCH OUT

A buffered writer must be flushed. The last few kilobytes sit in memory until you do. `defer w.Flush()` immediately after creating it is the habit, and note the order: `defer f.Close()` was registered first so it runs **last**, which is correct, because flushing into a closed file fails.

Better still, check the flush error, because a failed flush means data did not reach the disk:

```
defer func() {
    if ferr := w.Flush(); ferr != nil && err == nil {
        err = fmt.Errorf("flushing: %w", ferr)
    }
}()
```

io.Copy, and the pattern it enables

```
io.Copy(dst, src)      // stream everything from src to dst
```

That one line downloads a file to disk, with constant memory:

```
resp, err := http.Get(url)
if err != nil {
    return err
}
defer resp.Body.Close()

f, err := os.Create("download.bin")
if err != nil {
    return err
}
defer f.Close()
```

```
written, err := io.Copy(f, resp.Body)
```

An HTTP response body is an `io.Reader`. A file is an `io.Writer`. Neither knows about the other.

Useful companions:

```
io.Discard                      // a Writer that throws everything away
// read at most 1MB: use on untrusted input
io.LimitReader(r, 1<<20)
io.MultiWriter(f, os.Stdout)    // write to both at once, like tee
io.TeeReader(r, logFile)        // read from r while copying to logFile
```

WATCH OUT

`io.LimitReader` on anything a user controls is a real defence. Without it, a request body or an uploaded file can be as large as the client wants, and reading it into memory is a denial of service that costs the attacker nothing. For HTTP specifically there is `http.MaxBytesReader`, which also returns a proper error.

Writing testable code against interfaces

This is the practical payoff. A function that takes concrete types is hard to test:

```
// Hard: needs real files on disk.
func Process(inPath, outPath string) error
```

The same function over interfaces is trivial to test:

```
func Process(r io.Reader, w io.Writer) error {
    scanner := bufio.NewScanner(r)
    for scanner.Scan() {
        fmt.Fprintln(w, strings.ToUpper(scanner.Text()))
    }
    return scanner.Err()
}
```

```
func TestProcess(t *testing.T) {
    in := strings.NewReader("hello\nworld\n")
    var out bytes.Buffer

    if err := Process(in, &out); err != nil {
        t.Fatal(err)
    }
    if got, want := out.String(), "HELLO\nWORLD\n"; got != want {
```

```
        t.Errorf("got %q, want %q", got, want)
    }
}
```

No temp files, no cleanup, no filesystem. `strings.NewReader` and `bytes.Buffer` are the two types you will use most in tests.

> **IF YOU KNOW JAVASCRIPT**
> This is why Node's stream API exists, and Go's version is far smaller: one method per direction instead of an event emitter with `data`, `end`, `error` and backpressure semantics you have to learn. The tradeoff is that Go's streams are synchronous and blocking, which is fine because blocking a goroutine is cheap.

Paths, portably

```
import "path/filepath"

// data/users/ada.json or data\users\ada.json
filepath.Join("data", "users", "ada.json")
filepath.Ext("report.pdf")                  // ".pdf"
filepath.Base("/tmp/report.pdf")            // "report.pdf"
filepath.Dir("/tmp/report.pdf")             // "/tmp"
```

Use `filepath`, not `path`, and never build a path with string concatenation. `path` is for URLs and always uses forward slashes; `filepath` adapts to the operating system.

> **WATCH OUT**
> Never join a user-supplied name onto a directory without cleaning it:
>
> ```
> // Directory traversal: "../../etc/passwd" escapes the directory.
> f, _ := os.Open(filepath.Join("uploads", userInput))
> ```
>
> `filepath.Join` does call `Clean`, which resolves the `..`, but the result can still point outside `uploads`. The check is explicit:
>
> ```
> full := filepath.Join(base, filepath.Clean("/"+userInput))
> if !strings.HasPrefix(full, base+string(os.PathSeparator)) {
>     return errors.New("invalid path")
> }
> ```
>
> Go 1.24 added `os.Root` for this, which opens a directory and refuses to escape it, and is the better answer where available.

Temporary files and directories

```
dir, err := os.MkdirTemp("", "myapp-*")
if err != nil {
    return err
}
defer os.RemoveAll(dir)

f, err := os.CreateTemp(dir, "upload-*.tmp")
```

In tests, `t.TempDir()` does this and cleans up automatically:

```
func TestWrite(t *testing.T) {
    dir := t.TempDir()        // removed when the test ends, pass or fail
    path := filepath.Join(dir, "out.txt")
    ...
}
```

PROJECT 19: A LOG ANALYSER

Bring together everything from Parts I and II: stream files that may be large, parse them, aggregate with maps, sort, and process several files concurrently.

```
package main

import (
    "bufio"
    "flag"
    "fmt"
    "io"
    "os"
    "slices"
    "strconv"
    "strings"
    "sync"
)

// A line looks like:
// 192.168.1.1 GET /api/users 200 134ms
type stats struct {
    requests int
    byStatus map[int]int
    byPath   map[string]int
    totalMS  int
    slowest  string
    sloweste int
}

func newStats() *stats {
    return &stats{byStatus: map[int]int{}, byPath: map[string]int{}}
```

```
}

// analyse takes an io.Reader, not a filename, so it is testable with a
// strings.Reader and works on stdin, a file or a network stream alike.
func analyse(r io.Reader) (*stats, error) {
    s := newStats()
    sc := bufio.NewScanner(r)
    sc.Buffer(make([]byte, 0, 256*1024), 256*1024)

    for sc.Scan() {
        f := strings.Fields(sc.Text())
        if len(f) < 5 {
            // skip malformed lines rather than failing
            continue
        }
        status, err := strconv.Atoi(f[3])
        if err != nil {
            continue
        }
        ms, err := strconv.Atoi(strings.TrimSuffix(f[4], "ms"))
        if err != nil {
            continue
        }

        s.requests++
        s.byStatus[status]++
        s.byPath[f[2]]++
        s.totalMS += ms
        if ms > s.sloweste {
            s.sloweste, s.slowest = ms, f[2]
        }
    }
    // The check that is always skipped and always matters.
    if err := sc.Err(); err != nil {
        return nil, fmt.Errorf("scanning: %w", err)
    }
    return s, nil
}

func (s *stats) merge(o *stats) {
    s.requests += o.requests
    s.totalMS += o.totalMS
    for k, v := range o.byStatus {
        s.byStatus[k] += v
    }
    for k, v := range o.byPath {
        s.byPath[k] += v
    }
    if o.sloweste > s.sloweste {
        s.sloweste, s.slowest = o.sloweste, o.slowest
    }
```

```
}

func main() {
    top := flag.Int("top", 5, "how many paths to show")
    flag.Parse()

    paths := flag.Args()
    if len(paths) == 0 {
        // No files given: read standard input, so the tool pipes.
        s, err := analyse(os.Stdin)
        if err != nil {
            fail(err)
        }
        report(s, *top)
        return
    }

    // One goroutine per file, each with its own stats to avoid any
    // shared
    // state, merged at the end under a mutex.
    total := newStats()
    var mu sync.Mutex
    var wg sync.WaitGroup
    var failed int

    for _, p := range paths {
        wg.Go(func() {
            f, err := os.Open(p)
            if err != nil {
                mu.Lock()
                failed++
                mu.Unlock()
                fmt.Fprintf(os.Stderr, "skipping %s: %v\n", p, err)
                return
            }
            defer f.Close()

            s, err := analyse(f)
            if err != nil {
                fmt.Fprintf(os.Stderr, "reading %s: %v\n", p, err)
                return
            }
            mu.Lock()
            total.merge(s)
            mu.Unlock()
        })
    }
    wg.Wait()

    report(total, *top)
    if failed > 0 {
```

```
        os.Exit(1)
    }
}

func report(s *stats, top int) {
    if s.requests == 0 {
        fmt.Println("no requests found")
        return
    }
    fmt.Printf("requests   %d\n", s.requests)
    fmt.Printf("average    %dms\n", s.totalMS/s.requests)
    fmt.Printf("slowest    %s (%dms)\n\n", s.slowest, s.sloweste)

    codes := make([]int, 0, len(s.byStatus))
    for c := range s.byStatus {
        codes = append(codes, c)
    }
    slices.Sort(codes)
    for _, c := range codes {
        fmt.Printf("  %d  %d\n", c, s.byStatus[c])
    }

    type pair struct {
        path string
        n    int
    }
    paths := make([]pair, 0, len(s.byPath))
    for p, n := range s.byPath {
        paths = append(paths, pair{p, n})
    }
    slices.SortFunc(paths, func(a, b pair) int { return b.n - a.n })

    fmt.Println()
    for _, p := range paths[:min(top, len(paths))] {
        fmt.Printf("  %-30s %d\n", p.path, p.n)
    }
}

func fail(err error) {
    fmt.Fprintln(os.Stderr, "loganalyse:", err)
    os.Exit(1)
}
```

Make a test file and run it:

```
cat > access.log <<'EOF'
1.1.1.1 GET /api/users 200 134ms
1.1.1.2 GET /api/users 200 89ms
1.1.1.3 POST /api/login 401 12ms
1.1.1.4 GET /api/reports 200 2400ms
EOF
```

```
go run . access.log
cat access.log | go run .
```

Three design choices worth noticing. `analyse` takes an `io.Reader`, so the same function serves a file, standard input, and a `strings.Reader` in a test: write that test, it is four lines. Each goroutine builds its own `stats` and merges once at the end, so the lock is held briefly rather than on every line. And malformed lines are skipped rather than fatal, because a log analyser that dies on one bad line is useless against real logs.

YOUR TURN

Write the test for `analyse` using `strings.NewReader`, covering a good line, a malformed line and an empty input. Then add a `-json` flag that emits the stats as JSON instead of text, which needs the capital-letter rule and struct tags from the last chapter. Finally, feed it a file with a 100KB single line and watch where the buffer limit bites if you remove the `sc.Buffer` call.

ASSIGNMENT

Write `wc`, then make it testable without touching the disk.

In `part3-standard-library/20-io`:

One. Count lines, words and bytes in a file with `bufio.Scanner`. Handle the scanner's error, which is the line everybody forgets.

Two. Change the signature to take an `io.Reader` rather than a filename, and have `main` pass the file. Now the test passes a `strings.Reader` and there are no fixtures.

Three. Add a mode that copies input to output with `io.Copy` and reports the bytes moved, with the writes buffered and the buffer flushed in a `defer`.

Four. Feed it a file with a very long single line and fix the scanner error you get. Record the fix in `NOTES.md`.

COMMIT AS

```
git commit -m "Files - Assignment One"
```

21

net/http: a server with no framework

Go's standard library includes a production-grade HTTP server. Not a toy, not a reference implementation: the thing you actually deploy. This chapter builds a real one with no dependencies at all, because understanding this makes every framework afterwards, Gin and Grit included, read as a convenience rather than magic.

Five lines

```
package main

import (
    "fmt"
    "net/http"
)

func main() {
    http.HandleFunc("/hello",
        func(w http.ResponseWriter, r *http.Request) {
        fmt.Fprintln(w, "hello")
    })
    http.ListenAndServe(":8080", nil)
}
```

```
go run . &
curl localhost:8080/hello
```

No `npm install`, no framework, no config. That server handles concurrent requests across every core, because `net/http` starts a goroutine per request.

> **IF YOU KNOW JAVASCRIPT**
> Express needs `npm install express` and gives you `app.get('/hello', (req, res) => res.send('hello'))`. The shapes are similar. The differences: Go's server is in the standard library and versioned with the language, and it is concurrent by default rather than on one event loop.

The two core types

```
func handler(w http.ResponseWriter, r *http.Request)
```

`*http.Request` is the incoming request: method, URL, headers, body, context. `http.ResponseWriter` is where you write the response. Note the asymmetry: the request is a pointer to a struct you read, the writer is an interface you call.

```
func handler(w http.ResponseWriter, r *http.Request) {
    fmt.Println(r.Method)                 // "GET"
    fmt.Println(r.URL.Path)               // "/users/42"
    fmt.Println(r.URL.Query().Get("q"))   // ?q=search
    fmt.Println(r.Header.Get("Authorization"))

    w.Header().Set("Content-Type", "application/json")
    w.WriteHeader(http.StatusCreated)     // 201
    w.Write([]byte(`{"ok":true}`))
}
```

WATCH OUT

Order matters and the compiler will not help you. Headers must be set before `WriteHeader`, and `WriteHeader` before `Write`. Writing the body first implicitly sends a 200, and your later `WriteHeader(404)` is ignored with a log line:

```
http: superfluous response.WriteHeader call
```

The rule: set headers, then status, then body. Always that order.

Routing with ServeMux

Since Go 1.22 the standard router handles methods and path parameters, which removed the main reason people reached for a third-party router.

```
mux := http.NewServeMux()

mux.HandleFunc("GET /users", listUsers)
mux.HandleFunc("POST /users", createUser)
mux.HandleFunc("GET /users/{id}", getUser)
mux.HandleFunc("DELETE /users/{id}", deleteUser)

http.ListenAndServe(":8080", mux)
```

Path parameters come out by name:

```
func getUser(w http.ResponseWriter, r *http.Request) {
    id := r.PathValue("id")
    ...
}
```

A request that matches the path but not the method gets a 405 automatically.

> **IF YOU KNOW JAVASCRIPT**
> This is `app.get('/users/:id', ...)` with `req.params.id`. Two differences: the method is part of the pattern string rather than the function name, and there is no `next()`, because middleware composes by wrapping rather than by calling through.

Always configure a Server

`http.ListenAndServe` uses a default server with **no timeouts at all**. A single slow client can hold a connection open indefinitely, and enough of them exhaust your file descriptors. This is a real denial of service and it is the default.

```
srv := &http.Server{
    Addr:              ":8080",
    Handler:           mux,
    ReadHeaderTimeout: 5 * time.Second,   // slowloris defence
    ReadTimeout:       15 * time.Second,
    WriteTimeout:      15 * time.Second,
    IdleTimeout:       60 * time.Second,
    MaxHeaderBytes:    1 << 20,
}
log.Fatal(srv.ListenAndServe())
```

> **WATCH OUT**
> Set `ReadHeaderTimeout` at minimum. Without it, a client that opens a connection and sends one byte per minute occupies a goroutine and a file descriptor forever. This attack is called slowloris, it is twenty years old, and Go's defaults do not defend against it because the standard library cannot know your deployment.

Middleware is a function that wraps a handler

There is no middleware API. A middleware is a function taking a handler and returning a handler.

```
func Logger(next http.Handler) http.Handler {
    return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
        start := time.Now()
```

```
        next.ServeHTTP(w, r)
        log.Printf("%s %s %v", r.Method, r.URL.Path, time.Since(start))
    })
}
```

Compose by nesting:

```
handler := Logger(Recover(Auth(mux)))
```

The outermost runs first. Reading that line right to left gives you the request path, which is why many codebases add a small helper:

```
func Chain(h http.Handler,
    m ...func(http.Handler) http.Handler) http.Handler {
    for i := len(m) - 1; i >= 0; i-- {
        h = m[i](h)
    }
    return h
}

handler := Chain(mux, Logger, Recover, Auth)   // now reads in order
```

IF YOU KNOW JAVASCRIPT

Express middleware takes `(req, res, next)` and calls `next()` to continue. Go's takes a handler and **returns** one, so "continue" is calling `next.ServeHTTP(w, r)` and "stop" is simply not calling it:

```
func Auth(next http.Handler) http.Handler {
    return http.HandlerFunc(
        func(w http.ResponseWriter, r *http.Request) {
        if r.Header.Get("Authorization") == "" {
            http.Error(w, "unauthorized", http.StatusUnauthorized)
            // next is never called: stop here
            return
        }
        next.ServeHTTP(w, r)
    })
}
```

No `next(err)`, no forgetting to call `next()` and hanging the request.

Passing values down: context, not globals

Middleware that authenticates a user needs to hand it to the handler. The mechanism is the request context from chapter sixteen.

```
type ctxKey string
const userKey ctxKey = "user"

func Auth(next http.Handler) http.Handler {
    return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
        u, err := userFromToken(r.Header.Get("Authorization"))
        if err != nil {
            http.Error(w, "unauthorized", http.StatusUnauthorized)
            return
        }
        ctx := context.WithValue(r.Context(), userKey, u)
        // a new request, with the value
        next.ServeHTTP(w, r.WithContext(ctx))
    })
}

func handler(w http.ResponseWriter, r *http.Request) {
    u, ok := r.Context().Value(userKey).(*User)
    if !ok {
        http.Error(w, "no user", http.StatusInternalServerError)
        return
    }
    ...
}
```

`r.WithContext` returns a **copy** of the request. Forgetting to pass the new one to `next.ServeHTTP` is a common bug: the value is set on a request nobody sees.

Graceful shutdown

Killing a server mid-request drops responses. Production code waits.

```
func main() {
    srv := &http.Server{Addr: ":8080", Handler: handler}

    go func() {
        if err := srv.ListenAndServe(); err != nil &&
            !errors.Is(err, http.ErrServerClosed) {
            log.Fatalf("listen: %v", err)
        }
    }()
    log.Println("listening on :8080")

    // Wait for an interrupt.
    stop := make(chan os.Signal, 1)
    signal.Notify(stop, os.Interrupt, syscall.SIGTERM)
    <-stop

    log.Println("shutting down")
```

```
    ctx, cancel := context.WithTimeout(
        context.Background(), 10*time.Second)
    defer cancel()

    if err := srv.Shutdown(ctx); err != nil {
        log.Printf("forced shutdown: %v", err)
    }
    log.Println("stopped")
}
```

`Shutdown` stops accepting new connections and waits for in-flight requests, up to the context's deadline. `ListenAndServe` returns `http.ErrServerClosed` when this happens, which is why that error is checked for and ignored.

This matters for every container deployment: Kubernetes sends `SIGTERM` and waits before `SIGKILL`, and a server without this drops every request in flight on each deploy.

Testing handlers without a server

`net/http/httptest` runs a handler in memory.

```
func TestGetUser(t *testing.T) {
    req := httptest.NewRequest("GET", "/users/42", nil)
    req.SetPathValue("id", "42")
    rec := httptest.NewRecorder()

    getUser(rec, req)

    if rec.Code != http.StatusOK {
        t.Fatalf("status = %d, want 200", rec.Code)
    }
    var got User
    if err := json.NewDecoder(rec.Body).Decode(&got); err != nil {
        t.Fatalf("decoding body: %v", err)
    }
    if got.ID != 42 {
        t.Errorf("ID = %d, want 42", got.ID)
    }
}
```

No port, no network, no cleanup, and it runs in microseconds. For testing a whole mux including middleware, `httptest.NewServer` starts a real one on a random port.

> **PROJECT 20: A COMPLETE HTTP SERVICE, NO DEPENDENCIES**
> Everything above in one runnable file: routing, middleware, JSON, timeouts, graceful shutdown.

```
package main

import (
    "context"
    "encoding/json"
    "errors"
    "log"
    "net/http"
    "os"
    "os/signal"
    "sync"
    "syscall"
    "time"
)

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

type store struct {
    mu     sync.RWMutex
    notes  map[int]Note
    nextID int
}

func newStore() *store {
    return &store{notes: map[int]Note{}, nextID: 1}
}

func (s *store) list() []Note {
    s.mu.RLock()
    defer s.mu.RUnlock()
    out := make([]Note, 0, len(s.notes))
    for _, n := range s.notes {
        out = append(out, n)
    }
    return out
}

func (s *store) add(text string) Note {
    s.mu.Lock()
    defer s.mu.Unlock()
    n := Note{ID: s.nextID, Text: text}
    s.notes[n.ID] = n
    s.nextID++
    return n
}

// One helper for every JSON response, so the header is never forgotten.
func writeJSON(w http.ResponseWriter, status int, v any) {
```

```
    w.Header().Set("Content-Type", "application/json")
    w.WriteHeader(status)
    if err := json.NewEncoder(w).Encode(v); err != nil {
        log.Printf("encoding response: %v", err)
    }
}

func logger(next http.Handler) http.Handler {
    return http.HandlerFunc(
        func(w http.ResponseWriter, r *http.Request) {
        start := time.Now()
        next.ServeHTTP(w, r)
        log.Printf("%s %s %v", r.Method, r.URL.Path, time.Since(start))
    })
}

func recoverer(next http.Handler) http.Handler {
    return http.HandlerFunc(
        func(w http.ResponseWriter, r *http.Request) {
        defer func() {
            if rec := recover(); rec != nil {
                log.Printf("panic: %v", rec)
                writeJSON(w, http.StatusInternalServerError,
                    map[string]string{"error": "internal error"})
            }
        }()
        next.ServeHTTP(w, r)
    })
}

func main() {
    s := newStore()
    mux := http.NewServeMux()

    mux.HandleFunc("GET /notes",
        func(w http.ResponseWriter, r *http.Request) {
        writeJSON(w, http.StatusOK, s.list())
    })

    mux.HandleFunc("POST /notes",
        func(w http.ResponseWriter, r *http.Request) {
        var in struct {
            Text string `json:"text"`
        }
        dec := json.NewDecoder(http.MaxBytesReader(w, r.Body, 1<<20))
        dec.DisallowUnknownFields()
        if err := dec.Decode(&in); err != nil {
            writeJSON(w, http.StatusBadRequest,
                map[string]string{"error": "invalid json"})
            return
        }
```

```
        if in.Text == "" {
            writeJSON(w, http.StatusUnprocessableEntity,
                map[string]string{"text": "must not be empty"})
            return
        }
        writeJSON(w, http.StatusCreated, s.add(in.Text))
    })

    mux.HandleFunc("GET /panic",
        func(w http.ResponseWriter, r *http.Request) {
        panic("deliberate")      // proves the recoverer works
    })

    srv := &http.Server{
        Addr:              ":8080",
        Handler:           logger(recoverer(mux)),
        ReadHeaderTimeout: 5 * time.Second,
        ReadTimeout:       15 * time.Second,
        WriteTimeout:      15 * time.Second,
        IdleTimeout:       60 * time.Second,
    }

    go func() {
        if err := srv.ListenAndServe(); err != nil &&
            !errors.Is(err, http.ErrServerClosed) {
            log.Fatalf("listen: %v", err)
        }
    }()
    log.Println("listening on :8080")

    stop := make(chan os.Signal, 1)
    signal.Notify(stop, os.Interrupt, syscall.SIGTERM)
    <-stop

    ctx, cancel := context.WithTimeout(
        context.Background(), 10*time.Second)
    defer cancel()
    if err := srv.Shutdown(ctx); err != nil {
        log.Printf("forced: %v", err)
    }
    log.Println("stopped cleanly")
}
```

Exercise it:

```
curl localhost:8080/notes
curl -X POST localhost:8080/notes -d '{"text":"buy milk"}'
# 400, unknown field
curl -X POST localhost:8080/notes -d '{"txt":"typo"}'
curl -X POST localhost:8080/notes -d '{"text":""}'        # 422
# 405, automatic
```

```
curl -X DELETE localhost:8080/notes
# 500, and the server survives
curl localhost:8080/panic
```

Then press Ctrl+C and watch it log `stopped cleanly` rather than dying.

Count the dependencies in `go.mod`: there are none. This is a concurrent, panic-resistant, gracefully shutting down JSON API with request logging and body size limits, in one file, with nothing installed. That is the baseline Go gives you, and it is why Go web frameworks are thin.

YOUR TURN

Add `GET /notes/{id}` using `r.PathValue("id")` and `strconv.Atoi`, returning 404 when absent. Then write the handler test with `httptest` and no running server. Finally, remove `ReadHeaderTimeout` and leave a connection open with `telnet localhost 8080` without sending anything, to see for yourself what the default costs you.

ASSIGNMENT

A JSON API with no framework at all, so you know what one is for.

In `part3-standard-library/21-http`, build a notes API with `net/http`.

One. `GET /notes`, `POST /notes`, `GET /notes/{id}` on a `ServeMux`, returning JSON with the right status codes.

Two. A `Server` with read, write and idle timeouts set. Say in a comment what the default is and why it is a problem on the open internet.

Three. Two middlewares, logging and a request ID, composed as functions that wrap a handler. Put the ID in the request context and log it from the handler.

Four. Graceful shutdown on `SIGINT`, with an in-flight request finishing rather than being cut.

Five. Tests with `httptest`, no port bound. These are the four steps a generated Grit handler performs, which is worth having done by hand once.

COMMIT AS

```
git commit -m "HTTP Server - Assignment One"
```

22

Talking to a database with database/sql

Go's `database/sql` is not an ORM. It is a thin, driver-agnostic layer over SQL: you write the queries, it manages connections and scans rows into your variables. Coming from Prisma or Drizzle this feels primitive for about a day and then feels like relief, because nothing is hidden.

Opening a database

```
import (
    "database/sql"
    // the driver, imported for side effects only
    _ "modernc.org/sqlite"
)

db, err := sql.Open("sqlite", "app.db")
if err != nil {
    return fmt.Errorf("opening database: %w", err)
}
defer db.Close()

if err := db.Ping(); err != nil {
    return fmt.Errorf("connecting: %w", err)
}
```

The blank import registers the driver with `database/sql`. You never call the driver package directly, which is why it is imported as `_`.

> **WATCH OUT**
> `sql.Open` **does not connect.** It validates arguments and prepares a pool, lazily. A wrong password or a missing server is not reported until the first query. Always `Ping` at startup, or your service starts happily and fails on the first user request.

The pool is the important part

`*sql.DB` is not a connection. It is a pool, safe for concurrent use, and you should create exactly one and pass it around.

```
db.SetMaxOpenConns(25)                  // hard ceiling on connections
db.SetMaxIdleConns(25)                  // keep them warm
db.SetConnMaxLifetime(5 * time.Minute)  // recycle, for load balancers
db.SetConnMaxIdleTime(5 * time.Minute)
```

WATCH OUT

The default `MaxOpenConns` is **unlimited**. Under load, Go will happily open more connections than your database allows, and Postgres will start refusing them. Set it. A common starting point is 25, and the right number is something like `(cores * 2) + effective spindles`, measured rather than guessed.

Set `MaxIdleConns` to the same value as `MaxOpenConns`. If idle is lower, connections are closed and reopened constantly under steady load, which is pure latency.

Querying one row

```
func GetUser(ctx context.Context, db *sql.DB, id int) (*User, error) {
    var u User
    err := db.QueryRowContext(ctx,
        `SELECT id, name, email FROM users WHERE id = ?`, id,
    ).Scan(&u.ID, &u.Name, &u.Email)

    if errors.Is(err, sql.ErrNoRows) {
        return nil, fmt.Errorf("user %d: %w", id, ErrNotFound)
    }
    if err != nil {
        return nil, fmt.Errorf("querying user %d: %w", id, err)
    }
    return &u, nil
}
```

`Scan` copies columns into the pointers you give it, in order. The number and order must match the `SELECT`.

`sql.ErrNoRows` is the sentinel for "no row", and it is a normal outcome rather than a failure. Translating it into your own `ErrNotFound`, as above, is chapter eleven's design in practice: callers should not have to import `database/sql` to understand your errors.

Querying many rows

```
func ListUsers(ctx context.Context, db *sql.DB) ([]User, error) {
    rows, err := db.QueryContext(ctx,
        `SELECT id, name, email FROM users ORDER BY id`)
    if err != nil {
        return nil, fmt.Errorf("listing users: %w", err)
```

```
    }
    defer rows.Close()

    var users []User
    for rows.Next() {
        var u User
        if err := rows.Scan(&u.ID, &u.Name, &u.Email); err != nil {
            return nil, fmt.Errorf("scanning user: %w", err)
        }
        users = append(users, u)
    }
    // The check everybody forgets: the loop ends the same way on error.
    if err := rows.Err(); err != nil {
        return nil, fmt.Errorf("iterating users: %w", err)
    }
    return users, nil
}
```

Three things are mandatory and two of them are easy to forget.

`defer rows.Close()`. An unclosed `Rows` holds its connection. Leak enough and the pool is exhausted and every query blocks forever.

Check `rows.Err()` after the loop. Exactly like `scanner.Err()` in chapter twenty, and for the same reason.

Check the `Scan` error inside the loop.

Writing, and what you get back

```
res, err := db.ExecContext(ctx,
    `INSERT INTO users (name, email) VALUES (?, ?)`, name, email)
if err != nil {
    return fmt.Errorf("creating user: %w", err)
}

id, err := res.LastInsertId()     // SQLite and MySQL
n, err := res.RowsAffected()
```

WATCH OUT

`LastInsertId` is not supported by Postgres. There you use `RETURNING` and treat the insert as a query:

```
var id int
err := db.QueryRowContext(ctx,
    `INSERT INTO users (name, email) VALUES ($1, $2) RETURNING id`,
```

```
    name, email,
).Scan(&id)
```

Note also the placeholders: SQLite and MySQL use `?`, Postgres uses `$1`, `$2`. This is the main thing that is not portable between drivers, and it is why switching databases is never quite free.

Placeholders are not optional

```
// NEVER do this.
query := "SELECT * FROM users WHERE email = '" + email + "'"

// Always this.
db.QueryRowContext(ctx, "SELECT ... WHERE email = ?", email)
```

A placeholder sends the value separately from the statement, so it can never be parsed as SQL. String concatenation is SQL injection, and `'; DROP TABLE users;--` is only the famous example rather than the dangerous one: the real attacks read data rather than destroy it.

IF YOU KNOW JAVASCRIPT

Prisma and Drizzle protect you by generating the SQL. Here you write the SQL and the protection is the placeholder. The rule is simple: **values are placeholders, always**. Table and column names cannot be placeholders, so if those are dynamic, validate them against an allowlist you control. Grit does exactly this, which is why its generated services carry a `Sortable` map of permitted columns rather than interpolating whatever the client sent.

NULL, and why your struct fields lie

A `NULL` column cannot scan into a `string`. It fails at runtime:

```
sql: Scan error on column index 2: converting NULL to string is unsupported
```

Three options, in order of preference:

```
// 1. Best: make the column NOT NULL DEFAULT '' in the schema.

// 2. A pointer, when NULL is genuinely meaningful.
var bio *string
rows.Scan(&u.ID, &u.Name, &bio)

// 3. sql.NullString, when you want the distinction explicit.
```

```
var bio sql.NullString
rows.Scan(&u.ID, &u.Name, &bio)
if bio.Valid {
    u.Bio = bio.String
}
```

Since Go 1.22 there is also `sql.Null[T]`, the generic version:

```
var bio sql.Null[string]
```

Most NULL problems are schema problems. If a column has no meaningful "absent" state, make it `NOT NULL` and the whole question disappears.

Transactions

```
func Transfer(ctx context.Context, db *sql.DB,
    from, to int, amount int64) error {
    tx, err := db.BeginTx(ctx, nil)
    if err != nil {
        return fmt.Errorf("beginning transaction: %w", err)
    }
    // Safe to call after Commit: it becomes a no-op. This is the guard
    // that
    // makes every early return below correct.
    defer tx.Rollback()

    _, err = tx.ExecContext(ctx,
        `UPDATE accounts SET balance = balance - ? WHERE id = ?`,
        amount, from)
    if err != nil {
        return fmt.Errorf("debiting %d: %w", from, err)
    }

    var balance int64
    err = tx.QueryRowContext(ctx,
        `SELECT balance FROM accounts WHERE id = ?`, from).Scan(&balance)
    if err != nil {
        return fmt.Errorf("checking balance: %w", err)
    }
    if balance < 0 {
        return ErrInsufficientFunds      // the deferred Rollback undoes it
    }

    _, err = tx.ExecContext(ctx,
        `UPDATE accounts SET balance = balance + ? WHERE id = ?`,
           amount, to)
    if err != nil {
        return fmt.Errorf("crediting %d: %w", to, err)
```

```
    }

    return tx.Commit()
}
```

`defer tx.Rollback()` right after `BeginTx` is the pattern. After a successful `Commit` the rollback does nothing, so one line covers every failure path including a panic.

WATCH OUT

Everything inside a transaction must use `tx`, not `db`. A query on `db` inside a transaction takes a **different connection** from the pool, so it cannot see uncommitted changes and does not participate in the rollback. This compiles, passes casual testing, and corrupts data under concurrency.

The usual defence is to pass something that could be either:

```
type Querier interface {
    QueryRowContext(context.Context, string, ...any) *sql.Row
    ExecContext(context.Context, string, ...any) (sql.Result, error)
}
```

Both `*sql.DB` and `*sql.Tx` satisfy it, so a function can be called inside or outside a transaction. This is chapter ten's consumer-declared interface, doing real work.

The N+1 query problem

The most common performance bug in any application, in any language:

```
posts, _ := ListPosts(ctx, db)              // 1 query
for i := range posts {
    posts[i].Author, _ = GetUser(ctx, db, posts[i].AuthorID)   // N queries
}
```

Fifty posts means fifty-one queries. Each is fast, so nothing looks slow, and the page takes two seconds.

The fix is one more query, not fifty:

```
ids := make([]int, 0, len(posts))
for _, p := range posts {
    ids = append(ids, p.AuthorID)
}

// Build the right number of placeholders.
ph := strings.Repeat("?,", len(ids))
ph = ph[:len(ph)-1]
args := make([]any, len(ids))
```

```
for i, id := range ids {
    args[i] = id
}

rows, err := db.QueryContext(ctx,
    `SELECT id, name FROM users WHERE id IN (`+ph+`)`, args...)
// ... build map[int]User, then attach to posts
```

Note that the **placeholders** are generated and the **values** are still passed as arguments. That is safe. Interpolating the IDs themselves would not be.

> **IF YOU KNOW JAVASCRIPT**
> Prisma hides this behind `include: { author: true }`, and under the hood it does the same two-query dance. Here you do it yourself, which is more typing and means the query count is visible in your code rather than in a log you have to go looking for. In part four you will see Grit's profiler report "201 queries, 0 slow" for exactly this pattern.

Should you use an ORM?

You will meet GORM in part four, because Grit uses it. The honest comparison:

`database/sql`	You write SQL. Nothing hidden, no surprises, more code
`sqlc`	You write SQL, it generates typed Go. Excellent, and a build step
GORM	Structs and methods. Fast to write, and you must know what it emits

The position worth holding: an ORM is a productivity tool, not a replacement for knowing SQL. Every team that has been burned by an ORM was burned because nobody could say what query it was running. Learn this chapter first, then use whatever your project uses, and keep the query log open.

> **PROJECT 21: A DATABASE-BACKED NOTE STORE**
> Wire a real SQLite database behind the HTTP service from chapter twenty one. Pure standard library plus one driver.

```
go get modernc.org/sqlite      # pure Go, no cgo, no compiler needed
```

```
package main

import (
    "context"
    "database/sql"
    "errors"
```

```
    "fmt"
    "time"

    _ "modernc.org/sqlite"
)

var ErrNotFound = errors.New("not found")

type Note struct {
    ID        int       `json:"id"`
    Text      string    `json:"text"`
    CreatedAt time.Time `json:"created_at"`
}

type Store struct{ db *sql.DB }

func Open(path string) (*Store, error) {
    db, err := sql.Open("sqlite", path)
    if err != nil {
        return nil, fmt.Errorf("opening %s: %w", path, err)
    }
    db.SetMaxOpenConns(1)     // SQLite: one writer. For Postgres, 25.

    if err := db.Ping(); err != nil {
        return nil, fmt.Errorf("connecting: %w", err)
    }

    // Schema on startup. Real projects use migration files; chapter 31
    // covers how Grit does it.
    _, err = db.Exec(`
        CREATE TABLE IF NOT EXISTS notes (
            id         INTEGER PRIMARY KEY AUTOINCREMENT,
            text       TEXT    NOT NULL,
            created_at TEXT    NOT NULL
        )`)
    if err != nil {
        return nil, fmt.Errorf("creating schema: %w", err)
    }
    return &Store{db: db}, nil
}

func (s *Store) Close() error { return s.db.Close() }

func (s *Store) Create(ctx context.Context, text string) (*Note, error) {
    now := time.Now().UTC()
    res, err := s.db.ExecContext(ctx,
        `INSERT INTO notes (text, created_at) VALUES (?, ?)`,
        text, now.Format(time.RFC3339))
    if err != nil {
        return nil, fmt.Errorf("creating note: %w", err)
    }
```

```
    id, err := res.LastInsertId()
    if err != nil {
        return nil, fmt.Errorf("reading new id: %w", err)
    }
    return &Note{ID: int(id), Text: text, CreatedAt: now}, nil
}

func (s *Store) Get(ctx context.Context, id int) (*Note, error) {
    var n Note
    var created string
    err := s.db.QueryRowContext(ctx,
        `SELECT id, text, created_at FROM notes WHERE id = ?`, id,
    ).Scan(&n.ID, &n.Text, &created)

    if errors.Is(err, sql.ErrNoRows) {
        return nil, fmt.Errorf("note %d: %w", id, ErrNotFound)
    }
    if err != nil {
        return nil, fmt.Errorf("getting note %d: %w", id, err)
    }
    n.CreatedAt, _ = time.Parse(time.RFC3339, created)
    return &n, nil
}

func (s *Store) List(ctx context.Context, limit int) ([]Note, error) {
    rows, err := s.db.QueryContext(ctx,
        `SELECT id, text, created_at FROM notes
         ORDER BY id DESC LIMIT ?`, limit)
    if err != nil {
        return nil, fmt.Errorf("listing notes: %w", err)
    }
    defer rows.Close()

    var out []Note
    for rows.Next() {
        var n Note
        var created string
        if err := rows.Scan(&n.ID, &n.Text, &created); err != nil {
            return nil, fmt.Errorf("scanning note: %w", err)
        }
        n.CreatedAt, _ = time.Parse(time.RFC3339, created)
        out = append(out, n)
    }
    if err := rows.Err(); err != nil {
        return nil, fmt.Errorf("iterating notes: %w", err)
    }
    return out, nil
}

func (s *Store) Delete(ctx context.Context, id int) error {
    res, err := s.db.ExecContext(ctx,
```

```
        `DELETE FROM notes WHERE id = ?`, id)
    if err != nil {
        return fmt.Errorf("deleting note %d: %w", id, err)
    }
    n, err := res.RowsAffected()
    if err != nil {
        return fmt.Errorf("checking delete: %w", err)
    }
    if n == 0 {
        return fmt.Errorf("note %d: %w", id, ErrNotFound)
    }
    return nil
}
```

Now swap this in behind the handlers from chapter twenty one and you have a persistent JSON API with two dependencies: the SQLite driver and nothing else.

Two details worth dwelling on. `RowsAffected() == 0` is how `Delete` knows the row was missing, because a `DELETE` that matches nothing is not an error in SQL. And `SetMaxOpenConns(1)` is correct for SQLite, which allows one writer: leaving it unlimited gives you `database is locked` errors under concurrency, which is the single most common SQLite complaint and is a configuration problem rather than a SQLite problem.

YOUR TURN

Write a test using `t.TempDir()` for the database file, covering create, get, list and a `Get` that returns `ErrNotFound`. Then add an `Update` and give it a transaction that refuses to blank out the text, returning a validation error, and prove with a test that the row is unchanged after the failure. That last test is the one that proves you understand `defer tx.Rollback()`.

ASSIGNMENT

Move the notes API onto a real database.

In `part3-standard-library/22-database`, use SQLite through `database/sql`.

One. A schema, an `Open`, and the pool settings configured rather than defaulted.

Two. All four operations with placeholders. Then write one query by concatenating a string, put a quote in the input, and keep the result in `NOTES.md` as the reason you will not do it again.

Three. A nullable column read into `sql.NullString`, and a decision about what the API returns for it.

Four. A transaction that writes two rows and rolls back on the second failing, with a test that proves the first row is gone.

Five. Create the N+1: list notes, then fetch each note's tags in a loop. Count the queries, fix it with one query, and record both counts.

COMMIT AS

```
git commit -m "Database - Assignment One"
```

23

Project: a URL shortener

Everything from Parts I to III in one finished program: an HTTP server, a database, concurrency, context, error design, tests. Around three hundred lines, no framework, and it is genuinely deployable.

This chapter is one long project. Type it as you go.

What it does

```
POST /shorten  {"url": "https://example.com/very/long"}
               ->  {"code":"aB3dQ"}
GET  /aB3dQ                                ->  302 redirect
GET  /api/stats/aB3dQ                      ->  {"hits": 17}
```

Plus: it refuses nonsense URLs, counts hits without slowing the redirect, and shuts down cleanly.

The schema and the store

```go
// store.go
package main

import (
    "context"
    "database/sql"
    "errors"
    "fmt"
    "time"

    _ "modernc.org/sqlite"
)

var (
    ErrNotFound = errors.New("short code not found")
    ErrInvalid  = errors.New("invalid url")
)

type Link struct {
    Code      string    `json:"code"`
```

```
    URL       string    `json:"url"`
    Hits      int64     `json:"hits"`
    CreatedAt time.Time `json:"created_at"`
}

type Store struct{ db *sql.DB }

func OpenStore(path string) (*Store, error) {
    db, err := sql.Open("sqlite", path)
    if err != nil {
        return nil, fmt.Errorf("opening %s: %w", path, err)
    }
    db.SetMaxOpenConns(1)
    if err := db.Ping(); err != nil {
        return nil, fmt.Errorf("connecting: %w", err)
    }
    if _, err := db.Exec(`
        CREATE TABLE IF NOT EXISTS links (
            code       TEXT PRIMARY KEY,
            url        TEXT    NOT NULL,
            hits       INTEGER NOT NULL DEFAULT 0,
            created_at TEXT    NOT NULL
        )`); err != nil {
        return nil, fmt.Errorf("creating schema: %w", err)
    }
    return &Store{db: db}, nil
}

func (s *Store) Close() error { return s.db.Close() }

func (s *Store) Save(ctx context.Context,
    code, url string) (*Link, error) {
    now := time.Now().UTC()
    _, err := s.db.ExecContext(ctx,
        `INSERT INTO links (code, url, created_at) VALUES (?, ?, ?)`,
        code, url, now.Format(time.RFC3339))
    if err != nil {
        return nil, fmt.Errorf("saving %s: %w", code, err)
    }
    return &Link{Code: code, URL: url, CreatedAt: now}, nil
}

func (s *Store) Lookup(ctx context.Context, code string) (*Link, error) {
    var l Link
    var created string
    err := s.db.QueryRowContext(ctx,
        `SELECT code, url, hits, created_at FROM links
         WHERE code = ?`, code,
    ).Scan(&l.Code, &l.URL, &l.Hits, &created)
```

```
    if errors.Is(err, sql.ErrNoRows) {
        return nil, fmt.Errorf("%q: %w", code, ErrNotFound)
    }
    if err != nil {
        return nil, fmt.Errorf("looking up %q: %w", code, err)
    }
    l.CreatedAt, _ = time.Parse(time.RFC3339, created)
    return &l, nil
}

// RecordHit is deliberately separate from Lookup so the redirect does not
// wait for a write.
func (s *Store) RecordHit(ctx context.Context, code string) error {
    _, err := s.db.ExecContext(ctx,
        `UPDATE links SET hits = hits + 1 WHERE code = ?`, code)
    if err != nil {
        return fmt.Errorf("recording hit for %q: %w", code, err)
    }
    return nil
}
```

Generating codes

```
// code.go
package main

import (
    "crypto/rand"
    "math/big"
)

// Base58: no 0, O, I or l, so a code read aloud or copied by hand
// survives.
const alphabet = "123456789ABCDEFGHJKLMNPQRSTUVWXYZ" +
    "abcdefghijkmnopqrstuvwxyz"

func newCode(n int) (string, error) {
    b := make([]byte, n)
    max := big.NewInt(int64(len(alphabet)))
    for i := range b {
        k, err := rand.Int(rand.Reader, max)
        if err != nil {
            return "", err
        }
        b[i] = alphabet[k.Int64()]
    }
```

```
    return string(b), nil
}
```

> **WATCH OUT**
> `crypto/rand`, not `math/rand`. A short code generated from a predictable sequence can be guessed, which for a link shortener means enumerating everybody's private links. Use `crypto/rand` for anything a user should not be able to predict: codes, tokens, session IDs, password reset links.

Validating the URL

```
// validate.go
package main

import (
    "fmt"
    "net/url"
    "strings"
)

func validateURL(raw string) (string, error) {
    raw = strings.TrimSpace(raw)
    if raw == "" {
        return "", fmt.Errorf("url is empty: %w", ErrInvalid)
    }
    if len(raw) > 2048 {
        return "", fmt.Errorf("url too long: %w", ErrInvalid)
    }

    u, err := url.Parse(raw)
    if err != nil {
        return "", fmt.Errorf("parsing url: %w", ErrInvalid)
    }
    // Scheme must be http or https: javascript: and data: are XSS vectors
    // the moment this code appears in an href.
    if u.Scheme != "http" && u.Scheme != "https" {
        return "", fmt.Errorf("scheme %q not allowed: %w",
            u.Scheme, ErrInvalid)
    }
    if u.Host == "" {
        return "", fmt.Errorf("url has no host: %w", ErrInvalid)
    }
    return u.String(), nil
}
```

WATCH OUT
The scheme check is not optional. Without it, somebody shortens `javascript:alert(document.cookie)` and your redirect endpoint serves it. Any time user input ends up in an `href` or a `Location` header, allowlist the scheme.

The server

```
// main.go
package main

import (
    "context"
    "encoding/json"
    "errors"
    "log"
    "net/http"
    "os"
    "os/signal"
    "syscall"
    "time"
)

type Server struct {
    store *Store
    log   *log.Logger
}

func writeJSON(w http.ResponseWriter, status int, v any) {
    w.Header().Set("Content-Type", "application/json")
    w.WriteHeader(status)
    _ = json.NewEncoder(w).Encode(v)
}

func writeErr(w http.ResponseWriter, status int, msg string) {
    writeJSON(w, status, map[string]string{"error": msg})
}

func (s *Server) handleShorten(w http.ResponseWriter, r *http.Request) {
    var in struct {
        URL string `json:"url"`
    }
    dec := json.NewDecoder(http.MaxBytesReader(w, r.Body, 8<<10))
    dec.DisallowUnknownFields()
    if err := dec.Decode(&in); err != nil {
        writeErr(w, http.StatusBadRequest, "invalid json body")
        return
    }
```

```
    clean, err := validateURL(in.URL)
    if err != nil {
        writeErr(w, http.StatusUnprocessableEntity, err.Error())
        return
    }

    // Retry on the astronomically unlikely collision rather than failing.
    for attempt := 0; attempt < 5; attempt++ {
        code, err := newCode(6)
        if err != nil {
            s.log.Printf("generating code: %v", err)
            writeErr(w, http.StatusInternalServerError, "internal error")
            return
        }
        link, err := s.store.Save(r.Context(), code, clean)
        if err == nil {
            writeJSON(w, http.StatusCreated, link)
            return
        }
        s.log.Printf("save attempt %d: %v", attempt+1, err)
    }
    writeErr(w, http.StatusInternalServerError,
        "could not allocate a code")
}

func (s *Server) handleRedirect(w http.ResponseWriter, r *http.Request) {
    code := r.PathValue("code")

    link, err := s.store.Lookup(r.Context(), code)
    if errors.Is(err, ErrNotFound) {
        writeErr(w, http.StatusNotFound, "no such link")
        return
    }
    if err != nil {
        s.log.Printf("lookup %q: %v", code, err)
        writeErr(w, http.StatusInternalServerError, "internal error")
        return
    }

    // Count the hit without making the user wait for the write. The
    // request
    // context is about to be cancelled, so the goroutine gets its own.
    go func() {
        ctx, cancel := context.WithTimeout(
            context.Background(), 5*time.Second)
        defer cancel()
        if err := s.store.RecordHit(ctx, code); err != nil {
            s.log.Printf("recording hit: %v", err)
```

```
        }
    }()

    http.Redirect(w, r, link.URL, http.StatusFound)
}

func (s *Server) handleStats(w http.ResponseWriter, r *http.Request) {
    link, err := s.store.Lookup(r.Context(), r.PathValue("code"))
    if errors.Is(err, ErrNotFound) {
        writeErr(w, http.StatusNotFound, "no such link")
        return
    }
    if err != nil {
        writeErr(w, http.StatusInternalServerError, "internal error")
        return
    }
    writeJSON(w, http.StatusOK, link)
}

func (s *Server) routes() http.Handler {
    mux := http.NewServeMux()
    mux.HandleFunc("POST /shorten", s.handleShorten)
    mux.HandleFunc("GET /api/stats/{code}", s.handleStats)
    mux.HandleFunc("GET /{code}", s.handleRedirect)
    mux.HandleFunc("GET /healthz",
        func(w http.ResponseWriter, r *http.Request) {
        w.WriteHeader(http.StatusOK)
    })
    return s.recoverer(s.logger(mux))
}

func (s *Server) logger(next http.Handler) http.Handler {
    return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
        start := time.Now()
        next.ServeHTTP(w, r)
        s.log.Printf("%s %s %v", r.Method, r.URL.Path, time.Since(start))
    })
}

func (s *Server) recoverer(next http.Handler) http.Handler {
    return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
        defer func() {
            if rec := recover(); rec != nil {
                s.log.Printf("panic on %s %s: %v",
                    r.Method, r.URL.Path, rec)
                writeErr(w, http.StatusInternalServerError,
                    "internal error")
            }
        }()
```

```
        next.ServeHTTP(w, r)
    })
}

func main() {
    addr := os.Getenv("ADDR")
    if addr == "" {
        addr = ":8080"
    }
    dbPath := os.Getenv("DB_PATH")
    if dbPath == "" {
        dbPath = "links.db"
    }

    logger := log.New(os.Stdout, "", log.LstdFlags)

    store, err := OpenStore(dbPath)
    if err != nil {
        logger.Fatalf("opening store: %v", err)
    }
    defer store.Close()

    srv := &http.Server{
        Addr:              addr,
        Handler:           (&Server{store: store, log: logger}).routes(),
        ReadHeaderTimeout: 5 * time.Second,
        ReadTimeout:       15 * time.Second,
        WriteTimeout:      15 * time.Second,
        IdleTimeout:       60 * time.Second,
    }

    go func() {
        if err := srv.ListenAndServe(); err != nil &&
            !errors.Is(err, http.ErrServerClosed) {
            logger.Fatalf("listen: %v", err)
        }
    }()
    logger.Printf("listening on %s, database %s", addr, dbPath)

    stop := make(chan os.Signal, 1)
    signal.Notify(stop, os.Interrupt, syscall.SIGTERM)
    <-stop

    logger.Println("shutting down")
    ctx, cancel := context.WithTimeout(
        context.Background(), 10*time.Second)
    defer cancel()
    if err := srv.Shutdown(ctx); err != nil {
        logger.Printf("forced shutdown: %v", err)
```

```
    }
    logger.Println("stopped")
}
```

Running it

```
go mod init github.com/you/shortener
go get modernc.org/sqlite
go run .
```

```
curl -X POST localhost:8080/shorten -d '{"url":"https://go.dev/doc/"}'
# {"code":"7fKpQ2","url":"https://go.dev/doc/","hits":0,...}

curl -i localhost:8080/7fKpQ2          # 302, Location: https://go.dev/doc/
curl localhost:8080/api/stats/7fKpQ2   # {"hits":1,...}

curl -X POST localhost:8080/shorten -d '{"url":"javascript:alert(1)"}'
# 422, scheme "javascript" not allowed
```

The tests

```
// server_test.go
package main

import (
    "context"
    "encoding/json"
    "net/http"
    "net/http/httptest"
    "path/filepath"
    "strings"
    "testing"
)

func newTestServer(t *testing.T) *Server {
    t.Helper()
    store, err := OpenStore(filepath.Join(t.TempDir(), "test.db"))
    if err != nil {
        t.Fatalf("opening store: %v", err)
    }
    t.Cleanup(func() { store.Close() })
    return &Server{store: store, log: testLogger(t)}
}

func TestShortenAndRedirect(t *testing.T) {
    s := newTestServer(t)
```

```
    h := s.routes()

    req := httptest.NewRequest("POST", "/shorten",
        strings.NewReader(`{"url":"https://example.com/x"}`))
    rec := httptest.NewRecorder()
    h.ServeHTTP(rec, req)

    if rec.Code != http.StatusCreated {
        t.Fatalf("status = %d, want 201: %s", rec.Code, rec.Body)
    }
    var link Link
    if err := json.NewDecoder(rec.Body).Decode(&link); err != nil {
        t.Fatalf("decoding: %v", err)
    }
    if len(link.Code) != 6 {
        t.Errorf("code = %q, want 6 characters", link.Code)
    }

    req = httptest.NewRequest("GET", "/"+link.Code, nil)
    rec = httptest.NewRecorder()
    h.ServeHTTP(rec, req)

    if rec.Code != http.StatusFound {
        t.Fatalf("redirect status = %d, want 302", rec.Code)
    }
    got := rec.Header().Get("Location")
    if got != "https://example.com/x" {
        t.Errorf("Location = %q", got)
    }
}

func TestRejectsBadSchemes(t *testing.T) {
    s := newTestServer(t)
    h := s.routes()

    for _, bad := range []string{
        `{"url":"javascript:alert(1)"}`,
        `{"url":"data:text/html,<script>"}`,
        `{"url":""}`,
        `{"url":"not a url at all"}`,
    } {
        req := httptest.NewRequest("POST", "/shorten",
          strings.NewReader(bad))
        rec := httptest.NewRecorder()
        h.ServeHTTP(rec, req)
        if rec.Code != http.StatusUnprocessableEntity {
            t.Errorf("%s: status = %d, want 422", bad, rec.Code)
        }
    }
```

```
}

func TestUnknownCodeIs404(t *testing.T) {
    s := newTestServer(t)
    rec := httptest.NewRecorder()
    s.routes().ServeHTTP(rec, httptest.NewRequest("GET", "/nope12", nil))
    if rec.Code != http.StatusNotFound {
        t.Errorf("status = %d, want 404", rec.Code)
    }
}

func TestLookupMissing(t *testing.T) {
    s := newTestServer(t)
    _, err := s.store.Lookup(context.Background(), "missing")
    if err == nil {
        t.Fatal("want an error for a missing code")
    }
}
```

```
go test -race -cover ./...
```

WHAT THIS PROGRAM DEMONSTRATES

Walk back through it and name the chapter each decision came from.

Error design (11). `ErrNotFound` and `ErrInvalid` are sentinels, wrapped with `%w`, and the handlers branch on `errors.Is`. No handler reads an error message.

Pointers and nil (8). `Lookup` returns `*Link`, nil on failure, so a caller cannot accidentally use a half-built value.

Goroutines and context (13, 16). The hit counter runs in its own goroutine with a **fresh** context, because `r.Context()` is cancelled the moment the redirect is written. Using the request context there would silently lose most of your statistics under load, and it is the kind of bug that never shows up in testing.

Interfaces (10). `writeJSON` takes `http.ResponseWriter`, and the tests pass an `httptest.ResponseRecorder` instead. Nothing was designed for testability; it is testable because the standard library is built on small interfaces.

The database chapter (22). `defer rows.Close()`, `rows.Err()`, placeholders, `sql.ErrNoRows` translated at the boundary.

Security. `crypto/rand` for unguessable codes, a scheme allowlist against XSS, `MaxBytesReader` against oversized bodies, `DisallowUnknownFields` against typos, and timeouts on the server.

Count the dependencies: one, the SQLite driver.

YOUR TURN

Three extensions, in order of difficulty. Add an expiry column and a `GET /{code}` that returns 410 Gone for expired links. Add rate limiting to `POST /shorten` using a map of IP to a token bucket, protected by a mutex, which is chapter fifteen's material. Then add a custom alias (`{"url":..., "alias":"docs"}`), which forces you to handle a real `UNIQUE` constraint violation rather than the random collision retry, and that is a different error path entirely.

ASSIGNMENT

Finish the project, then push it past the chapter.

Build the shortener as written, in `part3-standard-library/23-shortener`, with the tests passing. Then add three things of your own.

One. Expiry: a link that stops working after a given time, with the expiry checked on read rather than by a sweeper.

Two. Hit counting that does not slow the redirect down. The redirect is the hot path, so the count should not be on it synchronously.

Three. A custom alias (`/go/standup`) that refuses to overwrite an existing one, with the race between two creators handled by the database rather than by a check.

Four. A README with the one command that builds it and the one that runs it. This is the first thing in the repository somebody else could use.

COMMIT AS

```
git commit -m "URL Shortener - Assignment One"
```

24

Configuration, environment and secrets

A program that works on your laptop and nowhere else usually has its configuration hard-coded. This short chapter is how to avoid that, and how not to leak a database password into your git history on the way.

Environment variables, the baseline

```
addr := os.Getenv("ADDR")
if addr == "" {
    addr = ":8080"       // a default, so the program runs with no setup
}
```

`os.Getenv` returns `""` for a variable that is unset, so you cannot tell unset from set-to-empty. When that matters:

```
addr, ok := os.LookupEnv("ADDR")
if !ok {
    // genuinely not set
}
```

> **IF YOU KNOW JAVASCRIPT**
> This is `process.env.ADDR`, with the same string-only limitation. There is no `dotenv` in the standard library, so a `.env` file needs a package, usually `joho/godotenv`. Many Go teams skip it entirely and export variables from a shell script or a compose file, because production almost never uses a `.env` file anyway.

Parse once, into a struct

Scattered `os.Getenv` calls are the thing to avoid. Read everything at startup, validate it, and pass the result down.

```
type Config struct {
    Addr        string
    DatabaseURL string
    LogLevel    string
    Debug       bool
```

```
    Timeout     time.Duration
    MaxWorkers  int
}

func Load() (*Config, error) {
    c := &Config{
        Addr:       getEnv("ADDR", ":8080"),
        LogLevel:   getEnv("LOG_LEVEL", "info"),
        Debug:      getEnvBool("DEBUG", false),
        Timeout:    getEnvDuration("TIMEOUT", 30*time.Second),
        MaxWorkers: getEnvInt("MAX_WORKERS", 4),
    }

    // Required: no sensible default exists.
    c.DatabaseURL = os.Getenv("DATABASE_URL")
    if c.DatabaseURL == "" {
        return nil, errors.New("DATABASE_URL is required")
    }
    if c.MaxWorkers < 1 {
        return nil, fmt.Errorf("MAX_WORKERS must be at least 1, got %d",
            c.MaxWorkers)
    }
    return c, nil
}

func getEnv(key, def string) string {
    if v, ok := os.LookupEnv(key); ok {
        return v
    }
    return def
}

func getEnvInt(key string, def int) int {
    if v, ok := os.LookupEnv(key); ok {
        if n, err := strconv.Atoi(v); err == nil {
            return n
        }
    }
    return def
}

func getEnvBool(key string, def bool) bool {
    if v, ok := os.LookupEnv(key); ok {
        if b, err := strconv.ParseBool(v); err == nil {
            return b
        }
    }
    return def
}
```

```
func getEnvDuration(key string, def time.Duration) time.Duration {
    if v, ok := os.LookupEnv(key); ok {
        if d, err := time.ParseDuration(v); err == nil {
            return d
        }
    }
    return def
}
```

`time.ParseDuration` accepts `"30s"`, `"5m"`, `"1h30m"`, which is far better than an integer whose unit lives in a comment.

WATCH OUT

Those helpers silently fall back to the default when parsing fails, so `MAX_WORKERS=eight` quietly becomes 4. For anything that matters, fail loudly instead:

```
func mustInt(key string, def int) (int, error) {
    v, ok := os.LookupEnv(key)
    if !ok {
        return def, nil
    }
    n, err := strconv.Atoi(v)
    if err != nil {
        return 0, fmt.Errorf("%s=%q: not a number", key, v)
    }
    return n, nil
}
```

A typo in a deployment variable should stop the program at startup, not change its behaviour silently three weeks later.

Fail at startup, not on the first request

```
func main() {
    cfg, err := Load()
    if err != nil {
        log.Fatalf("configuration: %v", err)
    }
    // From here, cfg is valid. Nothing downstream re-checks it.
}
```

This is the single most valuable habit in this chapter. A missing variable should crash the container immediately, so the deploy fails visibly, rather than letting a half-configured process serve errors.

Secrets

Three rules, and the first one is the one people break.

Never commit a secret. Not in a `.env` file, not in a test fixture, not "temporarily". Git keeps it forever, and a repository that goes public later publishes your history too.

```
# .gitignore
.env
.env.local
*.pem
```

Commit a `.env.example` with the **keys** and no values, so a new developer knows what to set.

Never log one. This is easy to do by accident:

```
log.Printf("config: %+v", cfg)    // prints DatabaseURL, password and all
```

The defence is a `String` method that redacts:

```
func (c Config) String() string {
    return fmt.Sprintf("Config{Addr:%s LogLevel:%s DatabaseURL:%s}",
        c.Addr, c.LogLevel, redact(c.DatabaseURL))
}

func redact(s string) string {
    if s == "" {
        return ""
    }
    return "[redacted]"
}
```

Because `fmt` looks for `String()`, this covers `%v`, `%s` and `Println`. It does not cover `%+v` on a struct that embeds yours, so the surer answer is a dedicated type:

```
type Secret string

func (Secret) String() string          { return "[redacted]" }
func (Secret) GoString() string        { return "[redacted]" }
func (s Secret) Reveal() string        { return string(s) }
```

Now the only way to see the value is to call `Reveal()`, which is greppable in review.

Never put one in a URL you log. `postgres://user:pass@host/db` is a secret in its entirety, which is why `redact` above hides the whole string rather than trying to mask the password portion.

IF YOU KNOW JAVASCRIPT
The mechanics are the same as Node: environment variables, a gitignored `.env`, a secret manager in production. The Go-specific part is that a compiled binary feels like a sealed box and is not: strings compiled into it are readable with the `strings` command. Never embed a key in source, even in a binary you only ship internally.

Flags, for tools rather than services

For a CLI, `flag` is in the standard library:

```
func main() {
    var (
        verbose = flag.Bool("v", false, "verbose output")
        out     = flag.String("o", "", "output file (default stdout)")
        limit   = flag.Int("n", 10, "how many results")
    )
    flag.Parse()

    for _, path := range flag.Args() {   // the non-flag arguments
        ...
    }
}
```

`flag` gives you `-h` for free, printing every flag with its default. It does not support subcommands or long-and-short pairs; for those, `spf13/cobra` is the near-universal choice and is what Grit's CLI is built on.

The usual division: **flags for tools, environment for services**. A service configured by flags is awkward in a container, and a CLI configured by environment variables is awkward to use.

PROJECT 22: CONFIGURE THE SHORTENER PROPERLY
Take the URL shortener from the last chapter and give it a real configuration layer, with validation that fails at startup.

```
// config.go
package main

import (
    "errors"
    "fmt"
    "os"
    "strconv"
    "time"
)
```

```
type Config struct {
    Addr         string
    DBPath       string
    CodeLength   int
    ReadTimeout  time.Duration
    WriteTimeout time.Duration
    BaseURL      string
}

func LoadConfig() (*Config, error) {
    c := &Config{
        Addr:    envOr("ADDR", ":8080"),
        DBPath:  envOr("DB_PATH", "links.db"),
        BaseURL: envOr("BASE_URL", "http://localhost:8080"),
    }

    var err error
    if c.CodeLength, err = envInt("CODE_LENGTH", 6); err != nil {
        return nil, err
    }
    c.ReadTimeout, err = envDur("READ_TIMEOUT", 15*time.Second)
    if err != nil {
        return nil, err
    }
    c.WriteTimeout, err = envDur("WRITE_TIMEOUT", 15*time.Second)
    if err != nil {
        return nil, err
    }

    // Validation. Every one of these is a mistake worth catching before
    // the first request rather than after.
    if c.CodeLength < 4 || c.CodeLength > 32 {
        return nil, fmt.Errorf("CODE_LENGTH must be 4 to 32, got %d",
            c.CodeLength)
    }
    if c.DBPath == "" {
        return nil, errors.New("DB_PATH must not be empty")
    }
    if c.ReadTimeout <= 0 || c.WriteTimeout <= 0 {
        return nil, errors.New("timeouts must be positive")
    }
    return c, nil
}

// String keeps the config loggable without leaking. Nothing here
// is secret today, and the method exists so that it stays safe when
// somebody adds an API key next month.
func (c Config) String() string {
    return fmt.Sprintf(
        "Config{Addr:%s DBPath:%s CodeLength:%d BaseURL:%s}",
```

```
        c.Addr, c.DBPath, c.CodeLength, c.BaseURL)
}

func envOr(key, def string) string {
    if v, ok := os.LookupEnv(key); ok && v != "" {
        return v
    }
    return def
}

func envInt(key string, def int) (int, error) {
    v, ok := os.LookupEnv(key)
    if !ok || v == "" {
        return def, nil
    }
    n, err := strconv.Atoi(v)
    if err != nil {
        return 0, fmt.Errorf("%s=%q is not a number", key, v)
    }
    return n, nil
}

func envDur(key string, def time.Duration) (time.Duration, error) {
    v, ok := os.LookupEnv(key)
    if !ok || v == "" {
        return def, nil
    }
    d, err := time.ParseDuration(v)
    if err != nil {
        return 0, fmt.Errorf("%s=%q is not a duration, try 15s", key, v)
    }
    return d, nil
}
```

And in `main`:

```
cfg, err := LoadConfig()
if err != nil {
    fmt.Fprintln(os.Stderr, "configuration:", err)
    os.Exit(1)
}
logger.Printf("starting with %v", cfg)
```

Try it:

```
go run .                                # defaults
ADDR=:9000 CODE_LENGTH=8 go run .       # overrides
CODE_LENGTH=2 go run .                  # exits 1 with a clear message
CODE_LENGTH=eight go run .              # exits 1, names the bad value
```

The last two are the point. A deployment with a typo'd variable now fails immediately and says which variable and what it received, instead of starting up and generating two-character codes that collide constantly.

Write the test too, since config is easy to test and almost nobody does:

```
func TestLoadConfigRejectsShortCodes(t *testing.T) {
    t.Setenv("CODE_LENGTH", "2")      // restored automatically
    if _, err := LoadConfig(); err == nil {
        t.Fatal("want an error for CODE_LENGTH=2")
    }
}
```

`t.Setenv` sets a variable for one test and restores it afterwards, and it refuses to run in a parallel test, because environment variables are process global and that would be a race.

YOUR TURN

Add an `API_KEY` using a `Secret` type with a redacting `String()`, and prove with a test that `fmt.Sprintf("%v", cfg)` does not contain the value. Then add `-config` as a flag pointing at a JSON file, with environment variables taking precedence over the file and the file over the defaults. Getting that precedence order right, and writing a test for each layer, is most of what configuration libraries do for you.

ASSIGNMENT

Make a program refuse to start rather than fail at midnight.

Take your chapter twenty two API and give it real configuration.

One. One `Config` struct, parsed once at startup, with defaults for everything that has a sensible one.

Two. Validation that fails loudly: a missing database URL exits with a message naming the variable, before any connection is attempted.

Three. A `String()` method that redacts secrets, and a startup log line that uses it. Then log the raw struct once to see what you would have leaked.

Four. The same binary started twice with different environments, proving nothing is compiled in.

The website's **Config With Defaults** and **Validating Config at Start-up** exercises are the compressed version.

COMMIT AS

```
git commit -m "Config - Assignment One"
```

25

Building, cross compiling and shipping one binary

This is the chapter that explains why people who deploy Node sound envious when Go comes up. Deployment is `scp` and `./app`, and the rest of this chapter is the detail around that sentence.

The build

```
go build -o app .
./app
```

One file. It contains your code, every dependency, and the Go runtime. The target machine needs nothing: no Go, no package manager, no shared libraries.

> **IF YOU KNOW JAVASCRIPT**
> Deploying a Next.js app means Node of the right major version, a `node_modules` tree, a build step, and a process manager. Deploying this means copying one file. That is the whole difference, and it is the reason Go dominates CLI tools and containers.

Smaller binaries

The default build includes debug symbols and a symbol table.

```
go build -ldflags="-s -w" -o app .
```

`-s` drops the symbol table, `-w` the DWARF debug information. Expect roughly 25 to 30 percent off. The cost is that a stack trace from a panic loses some detail and the binary cannot be inspected with a debugger, so many teams ship unstripped and accept the size.

For genuinely small, `upx` compresses further, at the cost of slower startup and antivirus software treating your binary with suspicion. Rarely worth it.

Embedding the version

A binary that cannot tell you which commit it is costs real time during an incident.

```
package main

var (
    version = "dev"
    commit  = "none"
    date    = "unknown"
)

func main() {
    if len(os.Args) > 1 && os.Args[1] == "version" {
        fmt.Printf("%s (%s) built %s\n", version, commit, date)
        return
    }
    ...
}
```

```
go build -ldflags="\
  -X main.version=1.4.2 \
  -X main.commit=$(git rev-parse --short HEAD) \
  -X main.date=$(date -u +%Y-%m-%dT%H:%M:%SZ)" -o app .
```

`-X` sets a string variable at link time. It only works on `string` variables in the `main` package or a named package path, and only ones that are not constants.

> **IF YOU KNOW JAVASCRIPT**
> Grit does exactly this, which is why `grit version` reports a real version rather than "dev". Its release process bumps three places that must agree: the Go variable, the scaffolder's default, and the documentation site's constant. Chapter 47 covers why a version that disagrees with itself is worse than no version.

Cross compiling

Two environment variables, no toolchain to install.

```
GOOS=linux   GOARCH=amd64 go build -o app-linux-amd64 .
GOOS=linux   GOARCH=arm64 go build -o app-linux-arm64 .
GOOS=darwin  GOARCH=arm64 go build -o app-darwin-arm64 .
GOOS=windows GOARCH=amd64 go build -o app-windows-amd64.exe .
```

From any machine, to any target. This is not an exaggeration: a Windows laptop builds the Linux ARM binary that runs on your Raspberry Pi, with no container and no configuration.

```
go tool dist list      # every supported pair
```

> **WATCH OUT**
> Cross compilation is free **until you use cgo**. Any package binding to a C library (the classic `mattn/go-sqlite3`, some image codecs, some crypto) needs a C cross-compiler for the target, and the simplicity evaporates.
>
> The defence is to prefer pure Go packages. This is why the database chapter used `modernc.org/sqlite` rather than `mattn/go-sqlite3`: the former is Go all the way down, so `GOOS=linux go build` works from anywhere. Check for cgo with:
>
> ```
> CGO_ENABLED=0 go build ./...
> ```
>
> If that fails, something in your tree needs C.

Containers, in two stages

A multi-stage Dockerfile builds in one image and ships in a tiny one.

```
# Build
FROM golang:1.25-alpine AS build
WORKDIR /src

# Copy manifests first: this layer caches unless dependencies change.
COPY go.mod go.sum ./
RUN go mod download

COPY . .
RUN CGO_ENABLED=0 go build -ldflags="-s -w" -o /bin/app .

# Ship
FROM gcr.io/distroless/static-debian12
COPY --from=build /bin/app /app
USER nonroot:nonroot
ENTRYPOINT ["/app"]
```

The final image contains your binary, CA certificates and nothing else: no shell, no package manager, no libc. Typical size is your binary plus about two megabytes.

`CGO_ENABLED=0` is required for the `static` base, because there is no libc in it to dynamically link against.

> **WATCH OUT**
> Copy `go.mod` and `go.sum` and run `go mod download` **before** copying your source. Docker caches layers, and if the source is copied first then every code change re-downloads

every dependency. This one ordering decision is the difference between a ten second rebuild and a two minute one.

Embedding files into the binary

`go:embed` puts files inside the executable, so your single-file deployment can still serve a frontend.

```
import "embed"

//go:embed static/*
var staticFiles embed.FS

//go:embed schema.sql
var schema string

func main() {
    http.Handle("/static/", http.FileServerFS(staticFiles))
    db.Exec(schema)
}
```

The comment is the directive and must sit immediately above the variable with no blank line. `embed.FS` for a tree, `string` or `[]byte` for one file.

This is how a Go program ships a built React app inside itself: compile the frontend, embed `dist/`, and deploy one file that serves both the API and the UI. Grit's single-binary architecture mode does precisely this.

Production flags worth knowing

```
go build -trimpath -ldflags="-s -w" -o app .
```

`-trimpath` removes absolute file paths from the binary, so a panic trace says `main.go:42` rather than `/home/you/projects/secret-name/main.go:42`. It also makes builds reproducible, which matters if you sign releases.

```
GOAMD64=v3 go build -o app .
```

Targets a newer x86 instruction set for a few percent. Only do this if you control the hardware, because the binary will not run on older CPUs.

Automating releases

A tag-triggered workflow that cross compiles and attaches binaries:

```yaml
name: Release
on:
  push:
    tags: ['v*']

jobs:
  release:
    runs-on: ubuntu-latest
    permissions:
      contents: write
    steps:
      - uses: actions/checkout@v4
        with: { fetch-depth: 0 }
      - uses: actions/setup-go@v5
        with: { go-version: '1.25' }
      - run: go test -race ./...
      - uses: goreleaser/goreleaser-action@v6
        with: { args: release --clean }
        env:
          GITHUB_TOKEN: ${{ secrets.GITHUB_TOKEN }}
```

`goreleaser` reads a config file, builds every platform, writes checksums, generates release notes from commits and uploads everything. It is the standard tool, and it is what turns "I tagged v1.2.0" into binaries for six platforms without further thought.

PROJECT 23: SHIP THE SHORTENER

Take the URL shortener and make it genuinely deployable: versioned, containerised and cross compiled.

First, the version:

```go
// version.go
package main

import (
    "fmt"
    "runtime"
)

var (
    version = "dev"
    commit  = "none"
    date    = "unknown"
)

func versionString() string {
    return fmt.Sprintf("shortener %s (%s) built %s with %s",
```

```
        version, commit, date, runtime.Version())
}
```

Wire it into `main` before anything else happens:

```
if len(os.Args) > 1 && os.Args[1] == "version" {
    fmt.Println(versionString())
    return
}
```

A `Makefile`, because the build command is now too long to retype:

```
VERSION ?= $(shell git describe --tags --always --dirty)
COMMIT  := $(shell git rev-parse --short HEAD)
DATE    := $(shell date -u +%Y-%m-%dT%H:%M:%SZ)
LDFLAGS := -s -w \
    -X main.version=$(VERSION) \
    -X main.commit=$(COMMIT) \
    -X main.date=$(DATE)

.PHONY: build test release clean

build:
  go build -trimpath -ldflags="$(LDFLAGS)" -o bin/shortener .

test:
  go test -race -cover ./...

release:
  GOOS=linux   GOARCH=amd64 go build -trimpath \
      -ldflags="$(LDFLAGS)" -o bin/shortener-linux-amd64 .
  GOOS=linux   GOARCH=arm64 go build -trimpath \
      -ldflags="$(LDFLAGS)" -o bin/shortener-linux-arm64 .
  GOOS=darwin  GOARCH=arm64 go build -trimpath \
      -ldflags="$(LDFLAGS)" -o bin/shortener-darwin-arm64 .
  GOOS=windows GOARCH=amd64 go build -trimpath \
      -ldflags="$(LDFLAGS)" -o bin/shortener-windows-amd64.exe .

clean:
  rm -rf bin
```

The `Dockerfile`:

```
FROM golang:1.25-alpine AS build
WORKDIR /src
COPY go.mod go.sum ./
RUN go mod download
COPY . .
ARG VERSION=docker
RUN CGO_ENABLED=0 go build -trimpath \
```

```
    -ldflags="-s -w -X main.version=$VERSION" -o /bin/app .

FROM gcr.io/distroless/static-debian12
COPY --from=build /bin/app /app
USER nonroot:nonroot
EXPOSE 8080
ENV DB_PATH=/data/links.db
ENTRYPOINT ["/app"]
```

Build and check the results:

```
make build && ./bin/shortener version
make release && ls -lh bin/

docker build --build-arg VERSION=1.0.0 -t shortener .
docker images shortener          # a few megabytes, not a few hundred
docker run -p 8080:8080 -v $(pwd)/data:/data shortener
```

Three things to notice. `make release` produces four binaries for four platforms from one machine, in seconds, with no cross-compilation toolchain, because nothing in this project uses cgo. The container has no shell at all, so `docker exec -it ... sh` fails: that is the point of distroless, and it is worth experiencing once so the tradeoff is a decision rather than a surprise. And the whole image is small enough that pulling it is not a deployment consideration.

Compare the image size to a Node service doing the same job. That number is the argument of this chapter.

YOUR TURN

Add `go:embed` for a small HTML page at `/` that posts to `/shorten`, so the single binary serves a usable interface as well as an API. Then add a `/healthz` that reports the version, and confirm with `docker run` that the version baked in by the build argument is what the endpoint reports. That round trip, from build flag to running container to HTTP response, is the whole deployment story in one check.

ASSIGNMENT

Ship the shortener as if somebody else had to run it.

One. Build with the version embedded, and have `-version` print it:

```
go build -ldflags "-X main.version=$(git describe --always)" .
```

Two. Cross compile for Linux, macOS and Windows, and record the three sizes in `RELEASE.md`.

Three. A two-stage `Dockerfile` ending in a minimal image. Record the image size next to the binary sizes.

Four. Embed the static files with `embed`, so one file is genuinely the whole program.

Five. A GitHub Actions workflow that does all of it on a tag and attaches the binaries to the release. Push a tag and let it run. That workflow file, in your repository, is the assignment.

COMMIT AS

```
git commit -m "Shipping - Assignment One"
```

26

The twenty six problems, and what a framework is for

You can now write Go. You have built a CLI, an HTTP service, a database layer and a deployable binary with no framework at all. So the fair question before part four is: **why would you add one?**

The answer is not "to write less code". It is that every project runs into the same list of problems, the list is longer than people admit, and most of it has nothing to do with the product you were hired to build.

Two lists, and they belong to different people

The problems split in two, and the split matters because the two halves are paid for by different people at different times.

The developer's twelve cost you the first two weeks. You feel them immediately and you can name them from memory.

The client's fourteen are questions asked before anybody mentions your actual feature. They decide whether there is a project at all, and they are always promised for later.

The developer's twelve

D1	The copy-paste start. Auth, storage and email lifted from the last repository, bugs included
D2	Wiring the monorepo: workspaces, ports, env files, Docker, before anything runs
D3	The same type written twice, or four times: Go, TypeScript, the validation schema, the mobile client
D4	The same UI for every table. Twenty tables, twenty slightly different list pages
D5	Testing performance is its own project, so it waits for a quiet week that does not come
D6	No window into the database without installing one more tool

D7	The API nobody wrote down. Hand-written OpenAPI is accurate for one afternoon
D8	Import and export are always a ticket, and always a week rather than an afternoon
D9	Nothing to develop against: empty tables, and mail you cannot read
D10	Changing the schema is the scary part: the migration that half ran, with no way back
D11	The frontend is somebody else's job, and there is nobody else
D12	Generating it with a model costs money every time, and none of it compounds

Most of these you have met in this book. You wrote the error boundary in chapter eleven, the middleware in twenty one, the config validation in twenty four. None of it was hard. All of it was time, and none of it was the product.

> **IF YOU KNOW JAVASCRIPT**
> D11 deserves a sentence, because it is the reason many readers picked this book up. You are fluent in React and now also in Go, which makes you the rare person who can do both. The cost is that you are now the person who does both, for every project, forever. A generated admin panel is not about disliking frontend work; it is about not rebuilding the same table for the twentieth time.

The client's fourteen

These arrive as questions, usually in the first meeting, before anybody has described the actual product.

B1	Can different people see different things?
B2	What happens if we get hacked?
B3	Does it work without internet?
B4	How will we know when it breaks?
B5	Is any of this backed up?
B6	What if it gets busy?
B7	Who changed this?
B8	Our users photograph things on their phones
B9	Can two of our clients use one system?
B10	Does the screen update by itself?
B11	Can we switch this on for one customer only?

B12	What if two people edit the same record?
B13	Does it talk to our other systems?
B14	Someone has asked us to delete their data

The honest answer to every one of them is "yes, in about three weeks", and the client hears that as a no.

Notice what they have in common: each is real work with **no demo attached**. You cannot show a stakeholder an audit log. So each gets promised for phase two, and offline, audit and observability are the three that never arrive.

Why these survive a decade of excellent libraries

Go's ecosystem is superb at solving one problem each. A router, a queue, a migration tool, an image decoder, all excellent. So why is the list still here?

Libraries solve one problem each. Nobody hands you the shape. A router solves routing. Nothing solves "what does an application made of these look like", so the shape is hand-carved on every project, slightly differently.

A template rots the day you clone it. A boilerplate repository freezes the bugs of the day you started. The fix you make on project five never reaches projects one to four, and you cannot tell which repository has which. This is D1 with extra steps.

The client's fourteen are always phase two. Expensive, invisible, always cut. Which is a precise definition of what belongs in a framework.

> **WATCH OUT**
> D12 is the newest and it is worth stating carefully, because the obvious version of the argument is wrong.
>
> An agent will write this scaffold for you. A Grit project measures about 74,500 lines, which is roughly three quarters of a million output tokens: at current frontier pricing, tens of dollars. That is cheap for two weeks of work and anybody who tells you the cost alone is the objection is overselling.
>
> The objection is the other three properties. You pay it **again** on the next project. The model writes something **different** each time, so the auth it gives you in March and in June are different code with different bugs. And because they are different, a fix you make in one project has nowhere to go.
>
> That is D1, the copy-paste start, with a bill attached.

The question a framework has to answer

So the question is not **which library**. It is:

Can the shape of an application be generated, owned outright by the team that generated it, and still improve after they own it?

Three clauses, and they fight each other.

Generated is easy. Plenty of tools generate code.

Generated and owned is harder, and is where most scaffolders stop: they hand you a repository and walk away.

Generated, owned, and still improving is the actual problem, because the moment code is yours, nobody can fix it but you. That third clause is what `grit upgrade` exists for, and chapter 29 is about how the line between "yours" and "the framework's" is drawn.

What this means for the rest of the book

Part four explains Grit's answer to that question: what `grit new` produces, how the architecture is laid out, why the **resource** is the unit of generation, and how the generated code stays plain Go that you can read with everything you learned in parts one to three.

Part five builds real applications with it.

Two promises about how this is written. Every claim about what Grit does comes from the codebase rather than from marketing, and where it does not close a problem, the book says so: B6, "what if it gets busy", is the honest example, because a framework can hand you a cache, read replicas and a job queue, and it cannot tell you which of them your bottleneck needs.

> **IF YOU KNOW JAVASCRIPT**
> You could also solve all twenty six yourself. You now know enough to. The question is whether you want to solve them on this project, and then again on the next one, and whether the version you write on project five can reach projects one to four.
>
> If your application is genuinely unusual, a handful of endpoints with an unusual data model and no admin surface, then the honest advice is to keep writing what you wrote in part three. A framework earns its place on CRUD-shaped products with an admin panel, several entities and a client who asks the fourteen questions. Most business software is exactly that, which is why these frameworks exist.

> **YOUR TURN**
> Before the next chapter, go through both tables and mark each problem as one of three things: **solved** on your current project, **promised** for later, or **not yet asked about**. Be honest about the middle column.

Then count the “promised” ones. That number is what a framework is offering to buy from you, and whether the trade is worth it depends almost entirely on how large it is.

ASSIGNMENT

Hold the twenty six problems against something you actually want to build.

Create `part4-grit/26-problems/PROBLEMS.md`.

One. Describe the application in five sentences. A real one, with a user who is not you.

Two. Go through D1 to D12 and mark each one as applies, does not apply, or not yet. For every one that applies, write the sentence you would say to a client about what it costs you.

Three. Do the same for B1 to B14, which are the ones you get asked about rather than the ones you feel.

Four. Count them. Then open the `NOTES.md` you wrote in chapter one and compare: the thing that pushed you to Go is usually two or three of these, and the rest are the ones you were going to meet anyway.

COMMIT AS

```
git commit -m "The Problems - Assignment One"
```

27

grit new: what lands, and the shape of a project

One command, and the answers to most of chapter twenty six are already in your directory. This chapter is a tour of what arrives, so that nothing in part five is a surprise.

Installing

```
curl -fsSL https://gritframework.dev/install.sh | sh    # macOS, Linux
# Windows PowerShell
irm https://gritframework.dev/install.ps1 | iex
go install github.com/MUKE-coder/grit/v3/cmd/grit@latest
```

```
grit version
```

Creating a project

```
grit new clinic
```

It asks a few questions, or you can answer them up front:

```
grit new clinic --api              # API only, no frontend
grit new clinic --architecture triple
```

The **architecture** decides how many frontends you get. It is a word this framework uses in a specific way, and it is worth pinning down now because it is not the same thing as an architecture style:

`triple`	Web app, admin panel and API, as three applications
`double`	Web app and API, with admin as role-protected routes in the web app
`single`	One binary serving an embedded SPA
`api`	The Go API alone
`mobile`	API plus an Expo application

> **WATCH OUT**
> In Grit, **architecture** means that choice and nothing else. The arrangement of the code inside, the modular monolith with its layers, is called the architecture **style**, and the two are documented separately for exactly this reason. When someone asks "which architecture does Grit use", find out which question they are asking.

What arrives

```
clinic/
├── grit.json                 the project's own description
├── docker-compose.yml        Postgres, Redis, MinIO, Mailhog
├── docker-compose.prod.yml
├── apps/
│   ├── api/                  the Go backend
│   ├── web/                  the public Next.js app
│   └── admin/                the admin panel
├── packages/
│   └── shared/               Zod schemas and TypeScript types
└── tests/
```

The Go side is where this book's attention goes:

```
apps/api/
├── go.mod
├── Dockerfile
├── cmd/
│   ├── server/               the API
│   ├── migrate/              schema changes
│   ├── seed/                 realistic data
│   ├── backup/               and
│   └── restore/              the half that matters
└── internal/                 about fifty packages
```

Five commands rather than one. `cmd/restore` existing beside `cmd/backup` is B5 from the last chapter answered honestly: a backup you have never restored is a rumour, so the restore is a first-class command rather than a runbook.

The internal packages

Around fifty, and the names tell you what has already been decided for you. Grouped by what they answer:

Your layers	`models`, `services`, `handlers`, `routes`, `policies`
Identity	`authz`, `password`, `totp`, `access`, `crypto`

Data	`database`, `migrate`, `paginate`, `concurrency`, `money`, `ids`
Work	`jobs`, `cron`, `outbox`, `saga`, `workflow`, `events`
Outside	`storage`, `files`, `media`, `mail`, `sms`, `webhooks`, `safefetch`
Operations	`audit`, `appendonly`, `backup`, `health`, `tracing`, `flags`
Compliance	`erasure`, `gdpr`-adjacent pieces, `settings`
Delivery	`respond`, `sanitize`, `imports`, `export`, `pdf`, `realtime`, `sync`

You will not read all of these. The point of listing them is that each one is a thing you did not write, and most of them correspond to a lettered problem from chapter twenty six.

Running it

```
cd clinic
grit migrate        # create the schema
grit seed           # realistic data to develop against
grit start          # everything, together
```

`grit start` brings up the API, the frontends and the infrastructure at once, which is D2 gone. What is now listening:

`:8080`	The Go API
`:3000`	The web app
`:3001`	The admin panel
`/studio`	A database browser: tables, queries, the ERD
`/docs`	The API reference, generated from the registered routes
`/pulse`	Latency, query counts, profiles
`/sentinel`	Errors, with the request attached

The second group is the interesting one. `/studio` is D6, `/docs` is D7 and `/pulse` is D5, and all three are paths that were already there rather than tools you installed.

> **IF YOU KNOW JAVASCRIPT**
> `/docs` deserves a note because it is the kind of thing that is usually a lie. It is assembled from the routes actually registered at runtime, not from comments or a hand-written specification. It cannot drift, because there is no second description to drift from. That is the only way an API document stays true, and it is why Grit generates one rather than asking you to maintain one.

grit.json, and the seeding that matters

`grit.json` records what the project is: its architecture, its module toggles, its name. The CLI reads it so that later commands know the shape of the thing they are editing.

The seed is worth insisting on, because it is the habit people skip:

```
grit seed Patient --count 50000
```

Fifty thousand rows is where you discover whether the list page works. At ten rows the sort is instant, pagination has one page, and an N+1 costs ten queries instead of ten thousand. That is D9, and the seeder is built as the pipeline from chapter fourteen: rows built in chunks, written in batches, with the number in flight bounded by a channel, so a million rows runs in about a minute with the heap flat.

Modules you can switch off

Eleven subsystems are toggles rather than permanent fixtures:

```
MODULE_AI=false
MODULE_AUDIT=false
MODULE_BACKUP=false
MODULE_CRON=false
MODULE_FILES=false
MODULE_FLAGS=false
MODULE_JOBS=false
MODULE_MAIL=false
MODULE_REALTIME=false
MODULE_TWOFACTOR=false
MODULE_WEBHOOKS=false
```

Set one to `false` in `.env` and that subsystem does not start. Delete its package and the rest still builds. This is the "batteries included, individually removable" claim, and it is checkable: the packages are separate and the seams are real.

> **WATCH OUT**
> The default project is the **hardened** one. Auth, sessions, the audit chain, backups and the upload pipeline are present before you write a line, rather than being plugins you remember to enable.
>
> That is deliberate and it is worth stating as a principle: an opt-in security feature is a security feature most projects do not have. The cost is a bigger project on day one, which is a real cost and the trade this framework makes on purpose.

It is still just Go

Open `apps/api/internal/services/user.go`. It is a struct with a `*gorm.DB` field and methods on it. Open a handler: it reads a request, calls one service method and writes a response. There is no framework type in your code and nothing to learn before you can read it.

```
// The shape of every generated service.
type ContactService struct {
    DB *gorm.DB
}
```

That is the whole dependency. If Grit disappeared tonight, this project compiles tomorrow, because there is nothing of Grit inside it. Everything you learned in parts one to three applies directly, which was the reason for putting those parts first.

PROJECT 24: MAKE A PROJECT AND LOOK AROUND

No code to write. Make one and read it, because the rest of part four assumes you have seen these files.

```
grit new bookshop --api
cd bookshop
grit migrate
grit seed
grit start
```

Then answer these by reading, not by searching the documentation.

One. Open `apps/api/internal/handlers/user.go` and find where it calls the service. How many lines of that handler are about HTTP, and how many are about users? That ratio is the thin-handler rule from chapter eleven, enforced by the generator.

Two. Open `apps/api/internal/services/user.go` and find `paginate.Config`. It lists which columns a client may search, sort and filter by. Why is that a whitelist rather than passing the client's column name through? The answer is in chapter twenty two, under placeholders.

Three. Visit `/studio` and look at the tables. Then visit `/docs` and find the route for listing users. Compare the parameters it documents with the `paginate.Config` you just read: they are the same list, because one is derived from the other.

Four. Count the entries in `apps/api/go.mod`. Then count what a comparable Express project needs in `package.json`. Neither number is a virtue on its own, and the comparison is worth seeing.

Five. Set `MODULE_MAIL=false` in `.env`, restart, and confirm the API still starts. Then look at what disappeared from the startup log.

YOUR TURN

Run `grit new` twice with the same name into different directories and compare the two trees byte for byte. They should be identical: the generator is deterministic, which is the property chapter twenty six said a model cannot give you. Then run `grit version` and note it, because the next chapters describe this version and the framework moves quickly.

ASSIGNMENT

Create the project you described in the last chapter, and read what arrived.

```
grit new fieldbook
```

One. Run it, open the admin panel, log in, and confirm the three apps are up. Record the commands that did it in `part4-grit/27-new/RUN.md`.

Two. Walk `apps/api/internal/` and write one line per package saying what it is for. Guess first, then check. The guesses you get wrong are the conventions worth knowing.

Three. Count what landed:

```
git -C fieldbook ls-files | wc -l
```

Put the number in `RUN.md` next to the time it took.

Four. Switch one module off in `grit.json`, restart, and say what stopped working. Commit the project itself: it is the thing the rest of Part IV edits.

COMMIT AS

```
git commit -m "Grit New - Assignment One"
```

28

The architecture: a modular monolith, layer by layer

You have the project. This chapter is the map: how the code is arranged, which layer is allowed to call which, and why those rules exist. Get this and everything in part five is predictable.

The short answer

A modular monolith with a layered API and an in-process event bus. One binary, one database, about fifty internal packages.

That is the headline and it is slightly misleading, because no real system is one style. A Grit project runs six at once, each where it earns its keep:

Modular monolith	The whole API: one deployment, one transaction boundary
Layered (n-tier)	routes, handlers, services, models
Event-driven	`internal/events` : a write emits once, subscribers react
Transactional outbox	`internal/outbox` : durable delivery of those events
Orchestration	`internal/saga` : multi-step work that can half fail
Microkernel	The framework itself: `grit plugin add`

> **IF YOU KNOW JAVASCRIPT**
> Coming from Next.js, the mental shift is that there is one process rather than a function per route. State can live in memory, a connection pool is warm, a background worker runs, and nothing is cold-started. The cost is that you deploy and scale one thing, which for most products is a relief rather than a limitation.

The request path

Follow one request all the way down. This is the shape every generated resource has.

```
HTTP request
  -> routes       which URL maps to which handler, and what guards it
```

```
  -> middleware   auth, CORS, rate limit, recover, request id
  -> handler      read the request, call one service method,
                  write the answer
  -> service      the rules, and every database call
  -> model        the struct, its GORM tags, its hooks
  -> database
```

Each layer talks only to the one below it. A handler never touches the database; a model never calls a service.

The rule the layering exists for

Handlers are thin **on purpose**, and the reason is not tidiness.

```go
// A generated handler, in essence.
func (h *ContactHandler) List(c *gin.Context) {
    opts := paginate.FromQuery(c, contactListConfig)
    result, err := h.service().List(h.ctx(c), opts)
    if err != nil {
        respond.Error(c, err)
        return
    }
    respond.Paginated(c, result)
}
```

Read the request, call one method, write the answer. Every rule lives one layer down, in the service.

The payoff: a rule enforced in the service is enforced **everywhere**. The background job uses it. The CSV import uses it. The CLI command uses it. The test uses it. A rule written in a handler holds until the first time something other than a browser does the same thing, and then it silently does not.

```go
// Every service method takes a context, not a *gin.Context.
func (s *ContactService) List(ctx context.Context,
    opts paginate.Options) (*paginate.Result[models.Contact], error)
```

That signature is the whole architecture in one line. `context.Context` rather than `*gin.Context` means the service has no idea it is being called over HTTP, so a job can call it with `context.Background()`.

> **WATCH OUT**
> The context carries more than cancellation here. It holds the organisation the multi-tenant plugin resolved and the actor that per-record ownership is scoped by. That is why passing the request's context down matters rather than creating a fresh one: a

`context.Background()` in the middle of a request loses the caller's identity, and the query comes back unscoped.

When a background job genuinely needs to act as nobody, there is an explicit way to say so rather than an accident.

Where the rules live, concretely

Open any generated service and you find three things above the methods:

```
// What a client may search, sort and filter by. Whitelisted, because each
// name ends up in SQL.
var contactListConfig = paginate.Config{
    Searchable: []string{"name", "phone"},
    Sortable:   map[string]bool{"id": true, "created_at": true,
        "name": true, "phone": true},
    Filterable: map[string]bool{"id": true, "name": true,
        "phone": true, "group_id": true},
}

// Every column Patch and Bulk may write. id, the timestamps and
// the version are the framework's, and are dropped.
var writableContact = map[string]bool{
    "name": true, "phone": true, "group_id": true,
}
```

These are the answer to chapter twenty two's warning that a column name cannot be a placeholder. The client sends `?sort=name`, and `name` is checked against a map the generator wrote, rather than being interpolated into SQL.

IF YOU KNOW JAVASCRIPT

This is the difference between an ORM protecting you and a convention protecting you. Prisma builds the query so the column is validated by its schema. Here the query is yours, so the whitelist is explicit and sitting at the top of the file where a reviewer sees it.

The event bus

A write emits once, and the systems that care subscribe.

```
events.Emit(c, events.Event{
    Name: "invoices.paid", Resource: "invoices", ID: inv.ID, After: inv,
})
```

The audit log, outbound webhooks and realtime clients are **subscribers**, not calls the handler has to remember. Adding a fourth consumer does not mean editing every handler.

Three delivery modes, and choosing correctly matters:

`Sync`	Runs inline, before the response. For the audit record, which must be written before the caller is told the write succeeded. A slow Sync subscriber is a slow endpoint
`Async`	Runs on a worker pool after the handler returns. For anything with a network call: a webhook that takes four seconds must not make the API take four seconds
`Durable`	Through the transactional outbox: at least once, retried, surviving a restart. For work another module depends on

`Durable` is the one worth understanding, because it is the difference between an event system and a reliable one. The event is written **in the same database transaction as the data**, so a crash between the commit and the HTTP call loses nothing. A relay then delivers it, claiming messages with a lease so another replica takes over if one dies.

> **IF YOU KNOW JAVASCRIPT**
> The naive version is a `POST` at the end of the handler. It works in the demo and loses events in production: the commit succeeded, the HTTP call did not, and nothing records that anything was owed. The outbox is the standard fix and it is already wired, which is B13 from chapter twenty six.

What it deliberately is not

Four, each with a reason rather than a dismissal:

Not microservices. One binary by design. A split is worth paying for when teams need to release independently, and until then it buys network failures and a deployment pipeline in exchange for nothing.

Not CQRS or event sourcing. Both earn their complexity in a hot spot, not across a codebase. The read path is already separate enough to point at another store when one query needs it.

Not hexagonal. The generated service talks to GORM directly rather than through a port. Less indirection to read, at the cost of swapping the ORM being real work. That is a trade, stated as one.

Not serverless. Jobs, a scheduler, realtime connections and a warm pool all assume a process that stays up.

Organised by layer, not by feature

A fair question, since package-by-feature is what many Go teams land on: why is it `models/`, `services/`, `handlers/` rather than `invoice/`, `customer/`?

What feature folders buy is real: a feature is a folder, you can delete one by deleting a directory, and the root stops growing. At forty resources Grit's `services/` is forty files and the directory stops telling you which three form a subsystem.

What the layered layout buys back is the **entity graph**. A business application has thirty entities pointing at each other: invoice to customer to organisation and back. **Go forbids import cycles**, so with feature folders the moment `customer` refers to `invoice` you need an event or an interface dance to express what is really a foreign key.

Grit's thirty-odd models are one flat `package models`. Any model can reference any other with no ceremony, which is what lets `client:belongs_to` be a single word in a generate command rather than a design conversation.

> **WATCH OUT**
> The import-cycle rule still bites, in the other direction. `models` cannot import `services`, so a hook on a model calls the lower-level package directly rather than the service that wraps it. If you add a `BeforeCreate` that needs a sequence number, call the sequence package, not the service.
>
> This is the kind of constraint that is obvious once stated and costs an hour when discovered by trial.

Can you refactor to something else?

Yes, and the useful part of the answer is this: **what decides whether a refactor is a weekend or a quarter is almost never the code. It is the data.**

Moving a package is easy. Moving a table out from under three foreign keys, two joins and an admin screen that lists both sides of a relationship is the real work, and no amount of clean code makes it cheaper.

CQRS on one endpoint	hours	The read path is already a separate whitelist. Point one query at a read replica
Extract one module	weeks	The service is the seam. The cost is the foreign keys you must cut
Fully hexagonal	ongoing	Mechanical, and the generator keeps writing the un-ported version
Microservices	months	Only with an organisational reason. "It will scale" is not one

Event sourcing	a rewrite	State stops being a row. Not a refactor
Collapse services in	weeks	Easier than the reverse, and often the right direction

PROJECT 25: TRACE A REQUEST END TO END

No new code. Follow one request through every layer of the project you made last chapter, with the profiler open, and the architecture stops being a diagram.

```
cd bookshop
grit start
```

One. Open `apps/api/internal/routes/` and find where the user routes are registered. Note which middleware guards them and in what order. The order is the composition from chapter twenty one: outermost runs first.

Two. Open the handler that route points at. Count its lines, then count how many are about HTTP rather than about users. Follow the one service call.

Three. In the service, find `paginate.Config` and the writable map. Make a request with `?sort=nonsense` and see what happens. Then make one with a column that **is** in the whitelist and watch it work. You have just tested the defence from chapter twenty two.

Four. Open `/pulse` and make the same request again. Note the query count for one request. Then create a resource with a `belongs_to` relationship, list it, and watch the query count: if it jumps by the number of rows, you are looking at an N+1, and the "Issued from" column names the file and line that caused it.

Five. Make a write, then open `/studio` and look at the audit table. The handler never called the audit log; it emitted an event, and audit was a subscriber. That indirection is the event bus doing its job.

YOUR TURN

Find a service method that takes `ctx context.Context` and call it from a `cmd/` command with `context.Background()`, bypassing HTTP entirely. It should work, because the service never knew it was being called over HTTP. That is the thin-handler rule paying off, and proving it once is worth more than reading the argument for it.

ASSIGNMENT

Trace one request all the way down and write what you saw.

In your generated project, pick `GET` on any resource and follow it.

One. `ARCHITECTURE.md` listing every file the request touches in order, from the route to the query, with one line each on what that layer decided.

Two. Break the rule on purpose: put a business rule in the handler. Then write the sentence that says why it is wrong, which is the test from the chapter: would it still have to be true if a background job called this?

Three. Subscribe to one event on the bus and log it. Trigger it through the admin panel and show the log line.

Four. Answer the refactoring question for your own application in a paragraph: if this had to become two services in two years, which seam would you cut, and what makes that seam cuttable today?

COMMIT AS

```
git commit -m "Architecture - Assignment One"
```

29

The resource, and why it is the unit of generation

This is the idea the framework rests on. If you disagree with it, the rest of part four is wrong, so it gets a chapter of its own.

The claim

A resource is the only thing that spans the whole stack, so it is the thing worth generating. Not a file. Not a layer. The invoice, from the database column to the form field.

Consider what "add invoices to this application" actually requires:

Database	a table, columns, indexes, a foreign key
Go	a model, a service with the rules, a handler, routes
Contract	a validation schema, a TypeScript type
Frontend	a data hook, a list page, a form, a detail view

Thirteen or so files. A scaffolder that generates **a model** has automated the easy part, because the model was never the expensive bit. The expense is that one concept, "invoice", has to be spelled correctly in thirteen places, and kept correct as it changes.

> **IF YOU KNOW JAVASCRIPT**
> This is D3 from chapter twenty six, and you have felt it. The Go struct, the Zod schema, the TypeScript type and the mobile client are four descriptions of one thing, kept in step by memory. Nothing fails to compile when they drift. You find out in the browser, from a user, as a blank column or a wrong total.

One command

```
grit generate resource Invoice \
  --fields "number:string, total:money, due:date, client:belongs_to"
```

Four field types. One of them is a relationship. That is the entire input.

What it writes, by name:

```
apps/api/internal/models/invoice.go
apps/api/internal/services/invoice.go
apps/api/internal/services/invoice_import.go
apps/api/internal/handlers/invoice.go
apps/api/internal/handlers/invoice_import.go
apps/api/internal/handlers/invoice_test.go
apps/api/internal/policies/invoice.go
apps/api/internal/factory/invoice.go
packages/shared/schemas/invoice.ts
packages/shared/types/invoice.ts
apps/admin/app/_dashboard/resources/invoice/index.tsx
```

Plus the routes registered, the model added to the migration list, and the resource added to the admin's navigation.

> **WATCH OUT**
> Note `index.tsx` in the resource folder. The generator writes it once and then **never regenerates it**, because that is where your custom cells and forms go. The distinction between "regenerated" and "written once" runs through the whole system, and chapter thirty one is about where that line sits.

Why one description is the point

The thirteen files are not the achievement. The achievement is that they come from **one description**, so they cannot disagree.

Add a field later:

```
grit generate field Invoice paid_at:datetime
```

That reaches the model, the migration, the Zod schema, the TypeScript type, the admin table and the admin form in one step. There is no version of this where the backend has `paid_at` and the frontend does not, because there is no second place to update.

That is the real answer to D3. Not "less typing": **no drift**.

A field type is a decision made once

Thirty one field types ship. The important ones are not the obvious ones.

```
string  text  richtext  int  uint  float  bool  toggle  select  radio
check  datetime  date  money  slug  belongs_to  one_to_one  many_to_many
string_array  file  files  email  url  domain  tel  country  color
percent  rating  time  json
```

Take `money`. Writing `total:money` decides a column type, a Go type, a Zod rule, an input mask and how the number is formatted in a table cell. Say `float` instead and you have shipped a rounding bug that will surface in a total months later, found by an accountant.

```
// What money generates: an integer, in minor units.
Total money.Amount    // stored as cents, never a float64
```

This is the lesson from chapter nineteen, where you implemented `MarshalJSON` for exactly this. The difference is that here it is a **type** rather than advice, so the correct decision is the default rather than something you must remember.

Take `belongs_to`. One word, and you get:

- a `ClientID` foreign key column, with a constraint
- a `Client` field on the struct, preloaded on read
- a nested type in the generated TypeScript
- a searchable dropdown in the admin form that shows the client's **name** rather than a UUID

> **IF YOU KNOW JAVASCRIPT**
> In a hand-written stack each of those is a separate decision someone makes slightly differently. The dropdown showing a UUID instead of a name is the canonical example: it is nobody's bug, everybody has shipped it, and it happens because the frontend had no way to know which column is the human label. Here the generator knows, because `belongs_to` told it.

It is still your code afterwards

Open the generated service. It is a struct with a `*gorm.DB` and methods, with comments explaining **why** rather than what:

```
// ContactService owns every database read and write for contacts.
//
// The handler reads the request, calls one of these and writes the
// answer. A job, a command or a test calls the same methods with no
// request at all, so a rule enforced here is enforced everywhere.
type ContactService struct {
    DB *gorm.DB
}
```

One dependency, and it is GORM. You can edit any of it. The generated code is the starting point, not a sealed artifact.

WATCH OUT

Re-running `grit generate resource` on an existing resource does **not** silently overwrite your edits. Every generated file is hashed into a manifest when written, and a file whose hash has moved is reported rather than replaced.

This behaviour exists because the opposite happened: an early version rewrote the model, service and handler on a re-run, printing a green tick for each, and everything hand-written in them was gone. No prompt, no backup, and the documentation had promised that the service was where your business logic goes. A newcomer lost a `BeforeSave` rule and a service method in their first week.

The lesson generalises past this framework: a generator that overwrites without asking is a generator you cannot trust with a file you have edited.

What it does not generate

Worth saying plainly, because the demo makes it look like magic.

It generates the **shape**. It does not generate the rule that an invoice cannot be billed twice, the report the client asks for every month, or the pricing logic that is the reason they hired you. That is still yours, it is still the hard part, and it is still where the value is.

What has gone is the two weeks before you got to any of it.

PROJECT 26: GENERATE A RESOURCE AND READ EVERY FILE

In the project from chapter twenty seven:

```
grit generate resource Book \
  --fields "title:string, isbn:string, price:money, published:date"
grit migrate
grit seed Book --count 500
grit start
```

Then open every file it named, in this order, and answer one question about each.

`models/book.go`. Find the `money` field. What Go type is it, and what does the GORM tag say the column is? Why is that not a `float64`?

`services/book.go`. Find `paginate.Config`. Which columns can a client sort by, and which cannot? Now try `?sort=price` and `?sort=secret` against the running API.

`handlers/book.go`. Count the lines that are about HTTP. Find the one call into the service per endpoint.

`packages/shared/schemas/book.ts`. Find the Zod rule for `price` and compare it to the Go validation. They came from the same word in your command.

`packages/shared/types/book.ts`. Compare the TypeScript type to the Go struct field by field.

The admin panel at `:3001`. Find the Books page. Sort by title, search, open the form. Notice the price input: it is a money input, not a number input, because of one word in your command.

Now add a field and watch the whole chain move:

```
grit generate field Book pages:int
grit migrate
```

Reload the admin. The column is in the table and the field is in the form, and you edited nothing. Check `types/book.ts`: it is there too.

Finally, prove the manifest. Add a method to `services/book.go` by hand, then re-run the original `grit generate resource Book` command. It should refuse to overwrite rather than quietly eating your method, and tell you why.

YOUR TURN

Generate a resource with a deliberately wrong field type, `price:float` instead of `price:money`, and look at what changes in the admin form and the Zod schema. Then compute `0.1 + 0.2` in the Go REPL of your choice, or just `fmt.Println(0.1 + 0.2)`, and decide for yourself whether a `money` type is pedantry or the single most valuable field type in the list.

ASSIGNMENT

Generate the domain you described in chapter twenty six.

One. Three resources, with their fields chosen rather than defaulted:

```
grit generate resource Site --fields "name:string, code:slug"
```

Two. Before running the command, write down what you expect it to create. Then run it with the diff open and check. The list you did not predict is the list worth reading.

Three. Edit one generated service by hand, re-run the generator for that resource, and show what happened to your edit. Record it in `NOTES.md`, because trusting a generator you have not tested this on is how edits get lost.

Four. Commit the whole diff as one commit, so the history shows what one command produced.

COMMIT AS

```
git commit -m "Resources - Assignment One"
```

30

Field types: what each one decides

Thirty one field types ship. This chapter is not a reference, it is an argument about why a field type is the right place to put a decision.

The argument

When you write `total:money`, you are not choosing a storage format. You are making five decisions at once:

Database	an integer column in minor units, never a float
Go	a `money.Amount`, with arithmetic that cannot drift
Validation	a Zod rule that accepts `12.34` and rejects `12.345`
Form	a money input with a currency prefix and two decimal places
Table	right aligned, formatted for the locale

Say `float` instead and all five change, each in a direction that is slightly wrong, and the one that costs you is the first. `0.1 + 0.2` is not `0.3` in any language with IEEE floats, and an invoice total that is out by a penny is found by an accountant rather than by a test.

> **IF YOU KNOW JAVASCRIPT**
> You already wrote this fix by hand in chapter nineteen, implementing `MarshalJSON` on a `Money int64`. That was the right instinct. The difference here is that the correct decision is the **default**, applied in five places consistently, rather than something each developer remembers or does not.

The full list

```
string  text  richtext  int  uint  float  bool  toggle  select  radio
check  datetime  date  money  slug  belongs_to  one_to_one  many_to_many
string_array  file  files  email  url  domain  tel  country  color
percent  rating  time  json
```

Most are unsurprising. The ones worth knowing properly are below.

Text in three sizes

`string`, `text` and `richtext` differ in the column type and the input:

`string`	A varchar, a single line input. Names, codes, references
`text`	A text column, a textarea. Descriptions, notes
`richtext`	A text column holding HTML, and an editor in the form

`richtext` is the one with a hidden cost: it stores HTML that a user supplied, so it has to be sanitised on the way in or you have shipped stored cross-site scripting. The generated service does that, which is the reason to use the field type rather than a `text` plus an editor you wire yourself.

Choices, in three shapes

`select`, `radio` and `check` all constrain a value to a set, and differ only in the control:

```
grit generate resource Ticket \
  --fields "status:select:open=Open|closed=Closed|held=On hold"
```

The values before the `=` go in the database; the labels after it go on the screen. This produces a constraint, a Go constant set, a Zod enum and a dropdown, from one string.

> **WATCH OUT**
> Use `select` for one choice from many, `radio` when there are three or fewer and the user should see them all at once, and `check` for several choices at once, which stores a `string_array`.
>
> The trap is choosing `string` and validating in your head. A free `status` column collects `open`, `Open`, `OPEN` and `opened` within a year, and no report ever agrees again.

Relationships

Three, and they are the field types that save the most work.

`belongs_to` is one word that produces a foreign key column with a constraint, a preloaded struct field, a nested TypeScript type, and a searchable dropdown in the form that shows the related record's **label column** rather than a UUID.

```
--fields "client:belongs_to"
```

```
ClientID uint    `json:"client_id"`
Client   *Client `json:"client"`     // preloaded on read
```

`one_to_one` is the same with a uniqueness constraint. `many_to_many` creates the join table and a multi-select picker.

> **IF YOU KNOW JAVASCRIPT**
> The dropdown showing a name rather than an ID is the detail worth noticing, because it is the canonical hand-written bug. It happens because the frontend has no way to know which column is the human label, so it falls back to the primary key. The generator knows, because `belongs_to` told it to look.

Identity and files

`slug` derives a URL-safe string from another field and keeps it unique, which is three decisions people usually get wrong separately: how to slugify, when to regenerate, and what to do about collisions.

`file` and `files` store an upload reference rather than bytes in a column, and wire the upload pipeline from chapter forty two: resize, re-encode, strip EXIF, generate a thumbnail. Writing `photo:file` is how B8 gets answered.

Validated strings

`email`, `url`, `domain`, `tel` and `country` are all strings with a format. They look cosmetic and are not: each produces a matching rule in Go **and** in Zod, so the browser and the API agree about what is valid.

> **WATCH OUT**
> The point is the agreement, not the validation. Validating only in the frontend is a user-experience feature, not a security control, because anybody can post directly to your API. Validating only in the backend means the user discovers the problem after a round trip.
>
> One field type produces both, from one definition, which is D3 again.

Numbers with meaning

`int`, `uint` and `float` are the raw ones. `percent`, `rating` and `money` are numbers that carry their own rules: a range, a step, a format. Prefer the specific one, because the constraint travels with it.

`json` stores arbitrary structure. Use it for genuinely unstructured data such as a webhook payload, and resist using it to avoid designing a schema, because nothing can query it well and nothing validates it.

Time, in three precisions

`date` is a calendar day with no time and no zone: a birthday, an invoice due date. `datetime` is an instant, stored UTC. `time` is a clock time with no date, for opening hours.

> **WATCH OUT**
> Using `datetime` for a birthday is a real bug, not a style preference. Store it as an instant and somebody in a different timezone sees the day before. The rule: **if the answer should be the same everywhere in the world, it is a `date`.**

Choosing well, in one table

Money	`money`, never `float`
A fixed set	`select`, never `string`
A calendar day	`date`, never `datetime`
A relationship	`belongs_to`, never an `int` you join by hand
An upload	`file`, never a `string` holding a path
An email	`email`, so both sides validate it
Unstructured	`json`, and only when it really is

Every row of that table is a decision that is cheap now and expensive in six months, which is the whole case for field types being a type system rather than a convenience.

> **PROJECT 27: ONE RESOURCE WITH EVERY INTERESTING TYPE**
> Generate a resource that exercises the types that carry decisions, then look at what each produced.

```
grit generate resource Property \
  --fields "title:string, slug:slug, description:richtext, \
price:money, bedrooms:int, rating:rating, available_from:date, \
viewing_time:time, status:select:draft=Draft|live=Live|sold=Sold, \
contact_email:email, website:url, photos:files, owner:belongs_to"
grit migrate
grit seed Property --count 200
grit start
```

Then check each claim rather than taking it on trust.

The money column. Open `models/property.go` and find `price`. What Go type, and what does the GORM tag say? Then open the admin form and type `12.345` into it. Then post `{"price": 12.345}` directly to the API with curl. Both should refuse, and that is the agreement this chapter is about.

The date against the time. Find `available_from` and `viewing_time` in the generated TypeScript. They are different types, because one is a day and one is a clock reading.

The select. Find the status constants in Go and the enum in the Zod schema. Then try posting `{"status": "pending"}` and read the error.

The relationship. Open the Property form and find the owner dropdown. It shows names. Now look at `services/property.go` and find the preload that makes that possible without an N+1.

The slug. Create a property called "Garden Flat", then another with the same name, and look at the two slugs.

The files. Upload two photos and then look at what landed in storage: the original is not what is served.

YOUR TURN

Generate the same resource twice, once with `price:money` and once with `price:float`, into throwaway projects. Diff the two generated models, Zod schemas and admin forms. Count how many files differ from that one word.

Then, in the float version, set a price of `0.1`, add `0.2` to it in a Go test, and assert it equals `0.3`. Watch it fail. That failing test is the entire argument of this chapter in four lines.

ASSIGNMENT

Use the type system the generator actually has.

One. Generate one resource using at least twelve different field types, including a relationship, a money, an enum, a slug and a file.

Two. Open the admin form and record, in a table in `FIELDS.md`, what each type produced: the input, the validation, the column type and the list cell.

Three. Find one that chose wrong for your domain, change it, migrate, and note what moved with it. A field type is a decision made once, so the cost of changing it is worth measuring yourself.

Four. Pick the two types you had never considered before reading the list, and say where you would have used a plain string instead.

COMMIT AS

```
git commit -m "Field Types - Assignment One"
```

31

Models, migrations and seeding at scale

D10 from chapter twenty six was “changing the schema is the scary part: the migration that half ran, with no way back”. This chapter is how that is answered, and it is more interesting than it sounds, because the usual answer is a lie.

The model

A generated model is a struct with tags:

```
type Invoice struct {
    ID        uint          `gorm:"primarykey" json:"id"`
    Number    string        `gorm:"size:50;uniqueIndex" json:"number"`
    Total     money.Amount `gorm:"not null" json:"total"`
    Due       jsontime.Date `json:"due"`
    ClientID  uint          `gorm:"index" json:"client_id"`
    Client    *Client       `json:"client,omitempty"`
    Version   int           `gorm:"default:1" json:"version"`
    CreatedAt time.Time     `json:"created_at"`
    UpdatedAt time.Time     `json:"updated_at"`
    DeletedAt gorm.DeletedAt `gorm:"index" json:"-"`
}
```

Three of those fields are the framework’s rather than yours.

`Version` is optimistic locking, and chapter thirty three explains the 409 it produces. `DeletedAt` is soft delete: a `DELETE` sets the timestamp and every query excludes it automatically, which is how the Trash page works. `CreatedAt` and `UpdatedAt` are maintained by GORM.

> **WATCH OUT**
> Soft delete has a consequence people meet late: a `uniqueIndex` on a column still applies to deleted rows. Delete a user with the email `ada@example.com` and you cannot create another with that email, because the old row is still there.
>
> The usual fix is a composite unique index including `deleted_at`, so the same value can exist once among live rows and many times among deleted ones. It is worth knowing before a user reports it.

Migrations, and the honest problem

```
grit migrate
```

It reads your models and makes the database match. No migration files to write.

That convenience has a well-known cost and this is where most frameworks are quiet: **GORM's AutoMigrate has no down scripts**. It can add a column. It cannot tell you what it added last Tuesday, and it cannot undo it.

So the answer here is not a pretend rollback. It is a record:

```
grit migrate           # apply, and record what this run changed
grit migrate --status  # what previous runs did
grit migrate --down    # undo the last run, using that record
```

Each run writes down what it actually added. `--down` reads that record and drops exactly those things. It is not a general rollback, it is an undo of **recorded** runs, and the distinction matters: a change made by someone typing SQL into a console was never recorded, so it cannot be undone this way.

> **IF YOU KNOW JAVASCRIPT**
> Prisma and Rails generate a migration file per change, with an explicit `up` and `down`, which is more ceremony and gives you a reviewable artifact. This trades the file for automatic detection plus a recorded history.
>
> Neither is free. The file-based approach costs a step per change and gives you a precise diff in code review. This approach costs you reviewability and gives you speed. Know which you are using, and never assume a framework that offers `--down` can undo something it did not do.

The refusal that protects you

```
grit migrate --down
```

Dropping a column destroys data. The command asks, and a non-interactive run needs `--yes`, deliberately, so a CI pipeline cannot quietly delete a column because somebody renamed a struct field.

A rename is the case to understand. Changing `Name` to `FullName` in a struct is not a rename to AutoMigrate: it is an unknown column appearing and a known one no longer being mentioned. The safe sequence is to add the new column, backfill it, then remove the old one in a later change, which is the same expand-and-contract dance every schema tool requires.

Seeding, and why it is not optional

D9 was "nothing to develop against".

```
grit seed                        # every seeder
grit seed Invoice --count 50000  # one, at scale
```

The generator writes a factory per resource that produces realistic values for each field type: a real-looking name for `string`, a valid address for `email`, an amount in range for `money`.

At ten rows everything works. The sort is instant, pagination has one page, and an N+1 costs ten queries instead of ten thousand. Every list-page bug you have is invisible until the data arrives, which is why the habit is to seed **before** you look at a screen rather than after a user complains.

A million rows, with the heap flat

The seeder is the pipeline from chapter fourteen, in production:

```
build rows in chunks  ->  bounded channel  ->  batch writers  ->  database
```

Measured on a laptop: **999,800 rows in 59 seconds, with the heap flat at 10 MB the whole time**.

The number that matters is the third one. Sixteen thousand rows a second is a fine figure and it is not the point. The point is that memory is a property of the **pipeline** rather than of how many rows you asked for: the channel holds a fixed number of batches in flight, so ten thousand rows and ten million look the same to the heap.

> **WATCH OUT**
> The naive versions both fail, in opposite directions. One row per `INSERT` keeps memory perfect and takes an afternoon, because you pay query parsing, a transaction and a round trip a million times. Building all million rows into a slice first is fast until the process is holding a million structs plus the driver's copy, and then it swaps or is killed.
>
> **Batching the writes is the win, not the goroutines.** Going from one insert per row to one insert per batch moves the number by an order of magnitude. Parallelism on top is worth perhaps five times more. Parallelism without batching is worth almost nothing, because a million round trips is a million round trips however many goroutines are waiting on them.

The seeder is also **resumable**: it counts what is in the table and inserts only the difference. Run it twice and the second run does nothing. Kill it halfway through a million rows and run it again, and it continues.

Backups, and the half that counts

```
grit backup           # write an archive
grit restore <file>   # replay one
```

`cmd/restore` existing beside `cmd/backup` is the point. B5 asked "is any of this backed up", and the honest reading of that question is "has anyone ever restored one".

A backup you have never restored is a rumour. The archive carries a row count per table, so a restore can be checked rather than assumed, and the restore is a command rather than a runbook page nobody has followed.

PROJECT 28: BREAK A MIGRATION ON PURPOSE

The point of this project is to experience the failure modes while they are cheap.

```
grit new migrations-lab --api
cd migrations-lab
grit generate resource Product \
  --fields "name:string, price:money, sku:string"
grit migrate
grit seed Product --count 1000
```

One: watch a run get recorded.

```
grit migrate --status
```

Note what the last run says it changed.

Two: add a column and undo it.

```
grit generate field Product weight:float
grit migrate
grit migrate --status      # a new run, with the new column
grit migrate --down        # and gone again
```

Check in `/studio` that the column really left.

Three: the rename trap. Edit `models/product.go` by hand and change `SKU` to `Code`. Run `grit migrate --status` after migrating and look carefully: a column was added, and the old one is still there with the data in it. Nothing was renamed, because nothing could know that was your intention.

Now do it properly: add the new field, write a tiny `cmd/` program that copies the values across, and only then remove the old one. That is expand-and-contract, and doing it once by hand is worth more than reading about it.

Four: the scale that matters.

```
time grit seed Product --count 200000
```

Watch the progress line report rows per second and the heap. Then open the admin Products page, sort by price and search. If anything is slow, open `/pulse` and look at the query count: this is the moment that a missing index announces itself, and it is the entire argument for seeding before you demo.

Five: prove the restore.

```
grit backup
grit seed Product --count 10          # change the data
grit restore <the archive>
```

Then count the rows. Restoring once, in a project where it does not matter, is how you find out your backup process works before the night it has to.

YOUR TURN

Add a `uniqueIndex` to the product SKU, create a product, soft delete it, and try to create another with the same SKU. Read the error, then fix it with a composite index including `deleted_at` and confirm both cases now work: one live row with that SKU, and any number of deleted ones.

That is a bug most applications ship, discovered by a user who deleted something and could not re-add it.

ASSIGNMENT

Make the data real, then prove you could get it back.

One. Add three fields to an existing model with `grit generate field`, then migrate. Record what the migration refused to do and why.

Two. Seed a hundred thousand rows and watch the memory while it runs:

```
grit seed Reading --count 100000
```

Put the peak in `DATA.md`.

Three. Now the half that counts. Take a backup, drop a table, restore it, and confirm the rows are back. Write the exact commands in `DATA.md`.

Four. Write one paragraph on what you would still lose if the machine died between the backup and the drop. Every honest backup story has that paragraph.

COMMIT AS

```
git commit -m "Migrations - Assignment One"
```

32

Services: where the rules live, and why not the handler

The thin-handler rule has come up three times now. This chapter is the one that justifies it properly and shows what a generated service actually contains.

The rule, and the reason

Every database read and write for a resource lives in that resource's service. The handler reads the request, calls one method, and writes the answer.

The reason is not tidiness. It is this:

```
func (s *InvoiceService) Create(ctx context.Context,
    in CreateInvoice) (*models.Invoice, error)
```

That signature takes a `context.Context`, not a `*gin.Context`. The service has no idea it is being called over HTTP. So the same method serves:

- the HTTP route
- a background job
- a CLI command in `cmd/`
- the CSV importer
- a test, with no server running

A rule enforced here is enforced in all five. A rule written in the handler holds until the first time something other than a browser does the same thing, and then it silently does not. That is the whole argument, and it is why "put the validation in the handler, it is right there" is the most expensive shortcut in a web codebase.

> **IF YOU KNOW JAVASCRIPT**
> In an Express app the equivalent discipline is a service module that never imports `req` or `res`. Most teams intend this and drift, because the handler is where the data already is. Go makes the drift visible: the moment a service needs `*gin.Context` it must import Gin, and that import is a review comment waiting to happen.

What a generated service contains

Four things, in this order.

The struct, with one dependency.

```
type ContactService struct {
    DB *gorm.DB
}
```

The list configuration, which is a security boundary:

```
// What a client may search, sort and filter by. Whitelisted, because each
// name ends up in SQL.
var contactListConfig = paginate.Config{
    Searchable: []string{"name", "phone"},
    Sortable: map[string]bool{
        "id": true, "created_at": true, "name": true, "phone": true,
    },
    Filterable: map[string]bool{
        "id": true, "name": true, "phone": true, "group_id": true,
    },
}
```

The writable set, which is the other one:

```
// Every column Patch and Bulk may write. id, the timestamps and the
// version are the framework's, and are dropped.
var writableContact = map[string]bool{
    "name": true, "phone": true, "group_id": true,
}
```

The methods: `List`, `GetByID`, `Create`, `Update`, `Patch`, `Delete`, `Bulk`, `BulkCreate`, `BulkEdit`, `Export`.

The two whitelists are the chapter

Chapter twenty two said a value can be a placeholder and a column name cannot. These maps are the consequence.

A client sends `?sort=name&filter[group_id]=3`. Those strings end up in SQL as **identifiers**, which no placeholder can protect. So each is checked against a map the generator wrote:

```
// Roughly what paginate does with it.
if !cfg.Sortable[requested] {
    requested = "created_at"      // fall back, never interpolate
}
```

`writableContact` is the same defence for writes, and it closes a different hole. A `PATCH` that merged the client's JSON straight into the model would let anybody set `id`, `created_at`, `version`, or on a user record, `role`. That is mass assignment, and it is how a normal user becomes an administrator.

> **WATCH OUT**
> When you add a field by hand rather than with `grit generate field`, **you must add it to these maps yourself**. The symptom of forgetting is not an error: sorting by the new column silently falls back, and patching it silently does nothing.
>
> This is the main reason to use `grit generate field` rather than editing the struct: the command updates the model, the migration, the two maps, the Zod schema, the type and the admin form together.

Ownership travels in the context

```
func (s *ContactService) db(ctx context.Context) *gorm.DB {
    return s.DB.WithContext(ctx)
}
```

That looks trivial and is not. The context carries the organisation the multitenant plugin resolved and the actor that per-record ownership is scoped by, and the scoping is applied through the context rather than by each method remembering to add a `WHERE`. So a service method that uses `s.db(ctx)` is automatically scoped, and one that reaches for `s.DB` directly is not.

> **WATCH OUT**
> This is the strongest argument for threading the request's context all the way down rather than creating a fresh one in the middle. A `context.Background()` halfway through a request does not merely lose cancellation, it loses the caller's identity, and the query comes back unscoped: every organisation's rows, not just this one's.
>
> When a background job genuinely must act as nobody, there is an explicit way to say so. The explicitness is the point: an unscoped query should be a sentence someone wrote on purpose.

Errors, by design

The service returns the sentinels and typed errors from chapter eleven:

```
func (s *ContactService) GetByID(ctx context.Context,
    id uint) (*models.Contact, error) {
    var c models.Contact
```

```
    err := s.db(ctx).First(&c, id).Error
    if errors.Is(err, gorm.ErrRecordNotFound) {
        return nil, fmt.Errorf("contact %d: %w", id, respond.ErrNotFound)
    }
    if err != nil {
        return nil, fmt.Errorf("getting contact %d: %w", id, err)
    }
    return &c, nil
}
```

Note what it does **not** do: it does not know about status codes, and it does not log. It returns, and the boundary decides. That is chapter eleven's "handle an error once" as a structural rule rather than advice.

Adding your own method

This is the part people worry about. The generated service is a starting point, and adding to it is ordinary Go:

```
// InvoiceService methods you add live alongside the generated ones.
func (s *InvoiceService) MarkPaid(ctx context.Context,
    id uint, paidAt time.Time) (*models.Invoice, error) {

    inv, err := s.GetByID(ctx, id)      // reuse the generated method
    if err != nil {
        return nil, err
    }
    if inv.PaidAt != nil {
        return nil, fmt.Errorf("invoice %d: %w", id, ErrAlreadyPaid)
    }

    inv.PaidAt = &paidAt
    if err := s.db(ctx).Save(inv).Error; err != nil {
        return nil, fmt.Errorf("marking invoice %d paid: %w", id, err)
    }

    events.Emit(ctx, events.Event{
        Name: "invoices.paid", Resource: "invoices",
        ID: inv.ID, After: inv,
    })
    return inv, nil
}
```

Three things worth copying from that. It reuses `GetByID` rather than writing another query. It returns a sentinel for the business rule, so the handler can map it to 409. And it emits an event rather than calling the audit log, the webhook sender and the realtime pusher by hand.

WATCH OUT
Put it in the **same file** as the generated methods, not a separate one.

The manifest from chapter twenty nine hashes generated files, so an edited file is reported on a re-run rather than silently overwritten. That protection only applies to the file it knows about. A method in `invoice_custom.go` is invisible to that machinery, and the next person to regenerate will not be warned about anything.

Where a rule does not belong in a service

Two exceptions worth naming.

A rule about the shape of a request belongs in the validation layer: "email must look like an email" is a Zod rule and a binding tag, not a service method.

A rule that spans resources may belong in a saga rather than in either service, because neither owns the sequence. Marking an invoice paid, adjusting stock and sending a receipt is three services and a failure mode in the middle, which is what `internal/saga` exists for.

PROJECT 29: ADD A BUSINESS RULE PROPERLY
Take the bookshop from chapter twenty seven and add a real rule: a book cannot be discounted below its cost price.

```
grit generate field Book cost:money
grit generate field Book discount_percent:percent
grit migrate
```

Then add the method to `services/book.go`, below the generated ones:

```
var ErrBelowCost = errors.New("discount would price below cost")

func (s *BookService) ApplyDiscount(ctx context.Context,
    id uint, percent int) (*models.Book, error) {

    if percent < 0 || percent > 90 {
        return nil, fmt.Errorf("discount %d: %w", percent,
            respond.ErrValidation)
    }

    b, err := s.GetByID(ctx, id)
    if err != nil {
        return nil, err
    }

    final := b.Price.MinusPercent(percent)
    if final.Cents() < b.Cost.Cents() {
```

```
        return nil, fmt.Errorf("book %d: %w", id, ErrBelowCost)
    }

    b.DiscountPercent = percent
    if err := s.db(ctx).Save(b).Error; err != nil {
        return nil, fmt.Errorf("discounting book %d: %w", id, err)
    }
    return b, nil
}
```

Now prove the rule holds in more than one place, which is the point of the chapter. Write a test that calls it directly, with no HTTP:

```
func TestApplyDiscountRefusesBelowCost(t *testing.T) {
    s := newTestBookService(t)
    b, err := s.Create(context.Background(), CreateBook{
        Title: "Go", Price: money.FromCents(2000),
        Cost: money.FromCents(1500),
    })
    if err != nil {
        t.Fatal(err)
    }

    _, err = s.ApplyDiscount(context.Background(), b.ID, 50)
    if err == nil {
        t.Fatal("want an error for a discount below cost")
    } else if !errors.Is(err, ErrBelowCost) {
        t.Errorf("got %v, want ErrBelowCost", err)
    }

    _, err = s.ApplyDiscount(context.Background(), b.ID, 10)
    if err != nil {
        t.Errorf("a 10 percent discount should be allowed: %v", err)
    }
}
```

That test runs with `context.Background()` and no server, which only works because the rule lives in the service. Move the check into a handler and try to write the same test: you now need a request, a recorder and a router, and the CSV importer bypasses your rule entirely.

Finally, write a four-line `cmd/discount/main.go` that calls the same method from the command line, and watch the rule apply there too, for free.

YOUR TURN

Add the discount field to the admin by hand instead of with `grit generate field`: edit only the model. Then try to sort the Books table by it and patch it through the API. Both

fail silently, because the two maps in the service do not mention it. Fix them, and you will not forget what those whitelists are for again.

ASSIGNMENT

Put a rule in the right place and prove it holds from two directions.

One. Add a method to a generated service that enforces something real: stock cannot go below zero, an invoice cannot be paid twice, a booking cannot overlap. Return a typed error, not a string.

Two. Call it from a handler. Then call the same method from a second caller: a CLI command, a seeder or a job. Both must be refused the same way, and that is the whole argument of the chapter.

Three. Add the new sortable and filterable columns to both whitelists, then deliberately leave one out and watch the admin column quietly do nothing.

Four. A test for the rule that does not go through HTTP at all.

COMMIT AS

```
git commit -m "Services - Assignment One"
```

33

Handlers and the HTTP contract

A handler's job is small and the contract it serves is not. This chapter is about the envelope every endpoint returns, why it is the same shape everywhere, and the two status codes that are worth understanding properly.

A handler, in full

```
func (h *ContactHandler) List(c *gin.Context) {
    opts := paginate.FromQuery(c, contactListConfig)
    result, err := h.service().List(h.ctx(c), opts)
    if err != nil {
        respond.Fail(c, err)
        return
    }
    respond.Paginated(c, result)
}
```

Four lines of work. Parse the query against the whitelist, call one service method, hand any error to the boundary, write the answer.

The `h.ctx(c)` is chapter thirty two's point: it converts the Gin context into a `context.Context` carrying the actor and the organisation, which is what makes the service's queries scoped.

One envelope, everywhere

Every successful response has the same shape.

A single item:

```
{
  "data": { "id": 1, "name": "Ada" },
  "message": "Contact created successfully"
}
```

A list:

```
{
  "data": [ ... ],
```

```
  "meta": { "total": 100, "page": 1, "page_size": 20, "pages": 5 }
}
```

And every failure:

```
{
  "error": {
    "code": "VALIDATION_ERROR",
    "message": "Email is required",
    "details": { "email": "This field is required" }
  }
}
```

> **IF YOU KNOW JAVASCRIPT**
> The value of a fixed envelope is on the **client** side. One `fetch` wrapper knows how to find the data, one error handler knows how to find the message, and `meta` is always where pagination lives. Without it, every endpoint is a small negotiation and the frontend accumulates special cases.
>
> This is why the generated React hooks are so short: they can assume the shape.

Status codes, and the ones that matter

200	OK
201	Created. A `POST` that made something returns this, with the thing
400	The request was malformed. Bad JSON, a missing parameter
401	Not authenticated. You have no identity
403	Authenticated, and not allowed. You have one, it is insufficient
404	No such thing, or none you may see
409	Conflict. Covered below
422	Well formed, and the values are wrong. The validation case
429	Rate limited
500	We broke

Two distinctions people get wrong.

400 against 422. A 400 means the request could not be understood: invalid JSON, a parameter that is not a number. A 422 means it was understood perfectly and the values are unacceptable: an empty name, an email without an `@`. The frontend treats them differently, because only one of them can point at a field.

401 against 403. A 401 says "I do not know who you are", and the client should log in. A 403 says "I know exactly who you are and the answer is no", and logging in again will not help. Returning 401 for a permissions failure sends users into a login loop.

404 as a permissions answer

A deliberate choice worth stating: when you ask for a record you are not allowed to see, you get **404, not 403**.

A 403 confirms the record exists. That is an information leak: an attacker can enumerate IDs and learn which invoices, users or documents are real, even without reading them. Answering 404 for both "absent" and "not yours" gives nothing away.

> **WATCH OUT**
> This surprises developers during integration, because a record they can see in the database returns 404 through the API. The cause is almost always ownership scoping doing its job: the query is filtered by the actor in the context, so the row is genuinely not in the result set.
>
> The fix is not to loosen the scope. It is to check who the request is authenticated as.

The 409, and a number that must not be a string

Optimistic locking is the `Version` column from chapter thirty one. A client sends the version it is editing, and the server refuses if it has moved:

```
If-Match: 8
```

```
{
  "error": {
    "code": "VERSION_CONFLICT",
    "message": "this invoice changed while you were editing it",
    "current_version": 9
  }
}
```

That response is doing something more useful than reporting a failure: it is telling the interface **enough to act**. With the current version the client can offer to reload, show the difference, or retry. A bare "conflict" leaves the frontend with nowhere to go but an alert box.

> **WATCH OUT**
> `"current_version": 9`, not `"9"`.

It shipped as a quoted string once. Every client that compared versions was comparing text, so `"10" < "9"` was true and the comparison was silently wrong in exactly the cases that matter.

The general lesson: a number that crosses a wire as a string will be compared as a string by somebody, and the failure is silent. This is the same family of bug as `omitempty` on a bool from chapter nineteen.

respond, and the 119 codes

The `respond` package is the boundary from chapter eleven, made concrete.

```
respond.OK(c, data)
respond.Created(c, data, "Contact created successfully")
respond.Paginated(c, result)

respond.NotFound(c, "contact")
respond.Forbidden(c, "not your contact")
respond.Conflict(c, err)
respond.Fail(c, err)              // works it out from the error
```

`respond.Fail` is the one to use by default. It walks the error chain with `errors.Is` and `errors.As`, finds the sentinel or typed error, and chooses the status and code. The handler does not decide; the error already knows.

There are 119 distinct error codes, which sounds excessive until you consider what a frontend does with them. `EMAIL_EXISTS` can focus the email field. `ACCOUNT_LOCKED` can show a different screen from `INVALID_CREDENTIALS`. `CSRF_INVALID` can retry after refreshing the token. A single `VALIDATION_ERROR` for all of them makes every one of those a string match on the message.

IF YOU KNOW JAVASCRIPT

A `Coded` interface lets an error carry its own wire shape:

```
// Implement it on a sentinel when the caller needs to act on it.
type Coded interface {
    Code() string
    Status() int
}
```

So a package can define an error that knows it is a 409 called `STOCK_INSUFFICIENT`, and `respond.Fail` needs no case for it. That is chapter ten's consumer-declared interface in the framework's own code.

Validation, in two places that agree

A generated handler binds the request into a struct with tags:

```
type CreateContact struct {
    Name  string `json:"name"  binding:"required,min=2,max=100"`
    Email string `json:"email" binding:"required,email"`
    Phone string `json:"phone" binding:"omitempty,e164"`
}
```

A binding failure becomes a 422 with per-field details, which is where the `details` object in the envelope comes from.

The matching Zod schema in `packages/shared` was generated from the same field definitions, so the browser refuses the same values for the same reasons. Neither is trusted: the frontend validation is for the user, and the backend validation is the control.

Writing your own handler

Adding an endpoint is ordinary Gin, and it should stay thin:

```
func (h *InvoiceHandler) MarkPaid(c *gin.Context) {
    id, err := strconv.ParseUint(c.Param("id"), 10, 64)
    if err != nil {
        respond.BadRequest(c, "invalid id")
        return
    }

    inv, err := h.service().MarkPaid(h.ctx(c), uint(id), time.Now())
    if err != nil {
        respond.Fail(c, err)      // ErrAlreadyPaid becomes a 409
        return
    }
    respond.OK(c, inv)
}
```

Then register it in the routes file, next to the generated ones.

> **WATCH OUT**
> Resist the urge to put the rule here because the data is already to hand. The check for "already paid" belongs in the service, as it was written in chapter thirty two, so that the CSV importer and the CLI obey it too.
>
> The test for whether a line belongs in a handler: **would it still need to be true if this were called by a background job?** If yes, it belongs one layer down.

> **PROJECT 30: EXERCISE THE WHOLE CONTRACT**

Use the bookshop and drive every branch of the envelope from the command line, so the contract stops being documentation.

```
grit start
```

The happy paths.

```
curl -s localhost:8080/api/v1/books | jq '.meta'
curl -s -X POST localhost:8080/api/v1/books \
  -H 'Content-Type: application/json' \
  -d '{"title":"Go","price":2000}' | jq
```

Note the 201 and the `message`, and that `meta` on the list has four keys.

400 against 422. Send broken JSON, then valid JSON with a bad value:

```
curl -s -X POST localhost:8080/api/v1/books \
  -d '{"title":' | jq '.error.code'
curl -s -X POST localhost:8080/api/v1/books \
  -H 'Content-Type: application/json' -d '{"title":""}' | jq '.error'
```

The second should carry `details` naming the field. That object is what lets a form highlight the right input rather than showing a banner.

401 against 403. Call a protected route with no token, then with a token for a user who lacks the role. Two different codes, and the frontend should do two different things.

404 for someone else's record. Create a book as one user, then fetch it as another. You get 404, not 403, and now you know why.

The 409. Fetch a book and note its `version`. Update it twice, the second time sending the stale version:

```
curl -s -X PATCH localhost:8080/api/v1/books/1 \
  -H 'If-Match: 1' -H 'Content-Type: application/json' \
  -d '{"title":"First"}' | jq '.data.version'

curl -s -X PATCH localhost:8080/api/v1/books/1 \
  -H 'If-Match: 1' -H 'Content-Type: application/json' \
  -d '{"title":"Second"}' | jq
```

The second returns 409 with `current_version`. Check the type in the raw JSON: it must be `9`, not `"9"`.

The whitelist. Try `?sort=title`, then `?sort=secret_column`. One works and one falls back silently, which is the behaviour chapter thirty two warned about.

YOUR TURN

Write a handler test with `httptest` that asserts the 409 body contains a numeric `current_version`, using a JSON decode into a struct with `CurrentVersion int`. If the server ever regresses to a string, the decode fails and the test catches it. That is a four line test guarding a bug that shipped once and was invisible.

ASSIGNMENT

Drive every branch of the contract and keep the evidence.

In `API.md`, record the request and the response for each of these against your own project.

One. A 201 with its `message`, and a list with all four `meta` keys.

Two. A 400 and a 422, with the 422 carrying `details` naming the field. Say in one line what a form would do differently with each.

Three. A 401 and a 403, from the same endpoint, by changing only who is asking.

Four. A 404 for a record that exists but is not yours, confirmed in `/studio` so you know it is scoping rather than absence.

Five. A 409 from a stale version, and a test asserting `current_version` decodes into an `int`. That test guards a bug that shipped once.

COMMIT AS

```
git commit -m "Handlers - Assignment One"
```

34

The generated admin panel

D4 was “the same UI for every table, twenty tables, twenty slightly different list pages”. D11 was “the frontend is somebody else’s job, and there is nobody else”. This chapter is the answer to both, and the part of the framework that is easiest to undersell and easiest to oversell.

What you get without asking

Generate a resource and the admin gains a working screen: a table with pagination, sorting, search and filters, a form with the right input for every field type, a detail view, bulk actions, Excel export and CSV import.

```
grit generate resource Invoice \
  --fields "number:string, total:money, due:date, client:belongs_to"
```

Reload the admin and Invoices is in the sidebar, listing rows.

The form is the part worth looking at. `total` is a money input with a currency prefix and two decimals. `due` is a date picker rather than a text box. `client` is a searchable dropdown showing client **names**. None of that was configured; it follows from the field types, which is chapter thirty’s argument arriving where you can see it.

How it fits together

```
apps/admin/
├── app/
│   └── _dashboard/
│       └── resources/
│           └── invoice/
│               └── index.tsx     written once, never regenerated
└── components/                   the shared table, form and inputs
```

The generator writes `index.tsx` **once**. After that it is yours, and a re-generate will not touch it. Everything underneath, the table component, the form builder, the inputs, is framework code shared by every resource.

WATCH OUT

That single file is the seam, and it is worth understanding before you fight it.

Customising a column, adding a computed cell, changing how a row renders: all of it happens in `index.tsx`, and it survives regeneration. Changing how **every** table behaves happens in the shared components, and that is framework code which `grit upgrade` will want to update.

The rule: per-resource changes in the resource file, cross-cutting changes in the shared components and expect to merge them on upgrade.

Customising a resource

The file is a definition rather than a page, which is why it is short:

```
export default defineResource({
  name: "invoice",
  label: "Invoices",
  columns: [
    { key: "number", label: "Number", sortable: true },
    { key: "client.name", label: "Client" },
    { key: "total", label: "Total", align: "right" },
    { key: "due", label: "Due", sortable: true },
  ],
})
```

To change a cell, give the column a renderer:

```
{
  key: "due",
  label: "Due",
  sortable: true,
  render: (row) => {
    const overdue = new Date(row.due) < new Date() && !row.paid_at
    return (
      <span className={overdue ? "text-red-500 font-medium" : ""}>
        {formatDate(row.due)}
      </span>
    )
  },
}
```

This is ordinary React. Everything you know applies, which is the point: the generated screen is a starting point written in the same language you would have used.

IF YOU KNOW JAVASCRIPT

If you have used Django admin or Laravel Filament, the shape is familiar: a resource definition rather than a page. The difference from Django is that this is React you can read and change, not a template system with its own idioms, and the difference from a hand-built panel is that twenty resources cost twenty definitions rather than twenty pages.

The list page, honestly

What the table does for you:

Pagination	Server side, using the `meta` from the last chapter
Sorting	Only on columns the service whitelisted
Search	Across the `Searchable` columns, server side
Filters	Per column, from the `Filterable` map
Selection	With bulk delete, edit and export
Import	CSV, with per-row errors rather than all-or-nothing
Export	Excel, honouring the current filters

Notice that four of those are driven by the service's configuration from chapter thirty two. The frontend does not decide what is sortable; it asks, and the server is the authority. That is why a column you added by hand without updating the maps appears in the table and refuses to sort.

Where it stops

This is the part worth being plain about, because an admin panel demo always looks like it does more than it does.

It is an operator interface, not your product. It is excellent for the screens staff use: managing records, fixing data, running an import. It is not a customer-facing experience, and trying to make it one fights the grain.

A genuinely custom screen is a page, not a resource. A dashboard that joins four models with a chart and a date range is not a table of one resource. Write it as a normal Next.js page using the generated hooks, and put it in the sidebar by hand.

The shared components are opinionated. They are dark-themed, dense, built for operators. Reskinning them entirely is possible and is a real project, and you will be merging your changes against upgrades afterwards.

WATCH OUT

The failure mode to avoid is bending the resource definition into something it is not. If you find yourself writing a `render` for every column, hiding the table and nesting a custom layout inside it, you have written a page inside a resource file.

Write the page. The generated hooks and types are available to it, and you lose nothing except the table you were not using.

The hooks underneath

Each resource generates typed React Query hooks:

```
const { data, isLoading } = useInvoices({ page: 1, search: "ACME" })
const { mutate: create } = useCreateInvoice()
const { mutate: update } = useUpdateInvoice()
```

These are the same hooks the generated table uses, and they are available to any page you write yourself. The types come from `packages/shared`, which came from the same field definitions as the Go model, so the autocomplete in your editor is describing the actual API.

That chain is D3 closed end to end: change a field, run one command, and the Go struct, the validation, the TypeScript type and this hook all move together.

PROJECT 31: CUSTOMISE A RESOURCE SCREEN

Take the bookshop and make the Books screen genuinely useful, which means touching exactly one file.

```
grit generate field Book stock:int
grit generate field Book cover:file
grit migrate
grit seed Book --count 300
grit start
```

Open `apps/admin/app/_dashboard/resources/book/index.tsx`.

One: a computed cell. Add a column that shows stock as a badge: red at zero, amber under ten, plain otherwise.

```
{
  key: "stock",
  label: "Stock",
  sortable: true,
  render: (row) => {
    const tone = row.stock === 0 ? "bg-red-500/15 text-red-400"
      : row.stock < 10 ? "bg-amber-500/15 text-amber-400"
      : "text-muted-foreground"
```

```
      return <span className={`rounded px-2 py-0.5 ${tone}`}>
        {row.stock}</span>
    },
  }
```

Two: a thumbnail. Render `cover` as an image rather than a filename.

Three: prove the whitelist is the authority. Make `stock` sortable in the definition, then try to sort by it. If you added the field with `grit generate field` it works. Now add a column for a field you added to the model by hand, mark it sortable, and watch it do nothing, because `services/book.go` never heard of it.

Four: prove the seam. Re-run `grit generate resource Book --fields "..."` with the original fields. Your `index.tsx` edits survive, because that file is written once. Check the model and service: those are the files the manifest reports on.

Five: write a real page. Add `apps/admin/app/_dashboard/reports/page.tsx` that uses `useBooks()` to show total stock value by category, with a chart. Notice you are writing ordinary React against typed hooks, and that this was never going to be a resource definition.

YOUR TURN

Count the lines in your customised `index.tsx`. Then estimate what the same screen would have cost hand-written: a table with server-side pagination, sorting, search, filters, selection, a modal form with eleven input types, validation wired to the API's error details, CSV import with per-row errors and Excel export.

That difference is D4 and D11 priced. It is also the fairest way to judge whether this part of the framework is worth it for your project, because if your application has three tables the answer might genuinely be no.

ASSIGNMENT

Customise one screen, then write the page the screen could not be.

One. In one resource's `index.tsx`, add a computed cell, reorder the columns, and render one field as something other than text.

Two. Re-run the generator for that resource and confirm your edits survived. Note which files the manifest reported instead.

Three. Write a real page under the dashboard that uses the generated hooks to show something no table could: a chart, a comparison, a daily summary.

Four. In `NOTES.md`, price it honestly. Count the lines you wrote, list what the table does for free, and say whether this part of the framework earns its place in your application. If your app has three tables, "no" is a respectable answer.

COMMIT AS

```
git commit -m "Admin Panel - Assignment One"
```

35

Authentication, roles and per-record ownership

B1 was "can different people see different things", and it is the question clients ask first. The answer has three layers that people routinely confuse, so this chapter separates them before showing what is generated.

Three questions, not one

Authentication	Who are you? Produces an identity, or a 401
Authorization	May people like you do this? Roles and permissions, or a 403
Ownership	May **you** do it to **this record**? Per-row, or a 404

Most teams build the first, assume the second is the whole of the problem, and discover the third when a salesperson can open another salesperson's accounts by changing a number in the URL.

> **IF YOU KNOW JAVASCRIPT**
> That third one has a name: **insecure direct object reference**, and it is perennially near the top of the OWASP list. It is not an exotic attack. It is changing `/invoices/41` to `/invoices/42`, and an application that checks only the role will serve it happily, because the role was right.

What arrives

A new project already has: registration, login, logout, password reset, email verification, refresh tokens, server-side sessions, account lockout, optional two-factor, and OAuth and SAML behind flags. Three roles exist by default, `ADMIN`, `EDITOR` and `USER`, and you can add more.

None of this is a plugin you enable. B2's answer is present before you write a line, for the reason chapter twenty seven gave: an opt-in security feature is a security feature most projects do not have.

Sessions, not just tokens

A JWT alone cannot be revoked: it is valid until it expires, so "log this device out" is not expressible. Every refresh token here is backed by a `sessions` row, stored as a SHA-256 hash rather than the token itself.

That buys four things a bare JWT cannot:

- revoke one device, from the account page
- detect replay: a refresh token used twice means it was stolen, and the whole session chain is killed
- idle and absolute timeouts
- a password change signs out every device

> **WATCH OUT**
> Storing only the hash matters. A database read, a leaked backup or an over-broad log should not hand someone a usable token. The same reasoning as password hashing, applied to the thing that is just as good as a password.
>
> The refresh cookie is also scoped to `/api/auth`, so it is not sent to every endpoint. The consequence to know: a handler outside that path cannot see the refresh cookie, so it cannot identify the **session**, only the user.

Four layers on the sign-in form

The login endpoint is the most attacked route in any application, and it carries four defences:

1	A rate limit on the route
2	A lockout on the account after repeated failures
3	A growing delay on repeated failure
4	A locked account answers exactly like a wrong password

The fourth is the one that gets missed, and it is the interesting one.

If a locked account says "this account is locked" and an unknown one says "no such user", you have built an oracle for discovering who has an account. Combined with a breach list, that is a targeted attack. The polite, helpful error message **is** the vulnerability.

> **WATCH OUT**
> The same trap hides in the timing. If the locked path returns instantly while the real check spends eighty milliseconds hashing a password, an attacker reads the clock instead of the message and learns the same thing.

> So the delay has to be on both paths. This is the kind of detail that is obvious once stated and absent from almost every hand-rolled login, and it is a decent argument for not hand-rolling one.

Roles, and adding your own

```
grit add role MODERATOR
```

That updates seven places: the Go constants, the Zod schema, the TypeScript union, the sidebar visibility, the form options and the seeded permissions. Doing it by hand means finding all seven, and the one you miss is usually the frontend, where the role exists but nothing shows for it.

Protecting a route:

```
admin := api.Group("/admin")
admin.Use(middleware.RequireRole("ADMIN", "EDITOR"))
```

And in a service, where it also holds for jobs and imports:

```
if !authz.Can(ctx, "invoices.delete") {
    return respond.ErrForbidden
}
```

Ownership is the one that bites

Roles say “editors may delete invoices”. Ownership says “this editor may delete **this** invoice”.

```
grit generate resource Invoice --owned-by user \
  --fields "number:string, total:money"
```

That adds a `UserID` column and, more importantly, scopes every generated query by the actor in the context. Listing returns only yours. Fetching someone else’s returns 404, not 403, for the reason chapter thirty three gave.

The scoping lives in the service rather than the handler, so it applies to the CSV import, the export, the background job and the CLI, which is the whole argument of chapter thirty two meeting the problem it was written for.

> WATCH OUT

This is why threading the request's context matters so much. A `context.Background()` created halfway through a request has no actor, so the scope cannot be applied, and the query returns **everybody's** rows.

The failure is silent and it is a data breach rather than an error. When a background job genuinely must see everything, there is an explicit way to say so, and the explicitness is the point: an unscoped query should be a sentence someone wrote deliberately.

Multitenancy is the same idea one level up

B9 asked "can two of our clients use one system". Ownership scopes to a user; multitenancy scopes to an organisation, through the same context.

The property to hold onto: **isolation is not a feature you add, it is something every query must have**. One forgotten `WHERE` in one place, in a background job or an export rather than a route, and one client sees another's data.

That is why it travels in the context rather than being a parameter each method remembers to apply. A parameter can be forgotten at one call site. A context that carries the organisation is applied by the layer that builds the query.

What is not solved for you

Two honest limits.

Permission design is yours. The framework gives roles, permissions and per-record ownership. Deciding that a regional manager sees their region, and what happens when someone moves region, is domain modelling and nothing generates it.

Approval workflows are not authorization. "An invoice over ten thousand needs a second approver" is a business rule that lives in a service, possibly a saga. It looks like permissions and it is not, and modelling it as a role produces a mess.

PROJECT 32: PROVE THE ISOLATION HOLDS

The only way to trust ownership scoping is to try to break it.

```
grit new tenancy-lab --api
cd tenancy-lab
grit generate resource Note --owned-by user \
  --fields "title:string, body:text"
grit migrate
grit start
```

One: two users. Register Ada and Grace through the API and keep both tokens.

Two: a note each. Create one note as each user. Note the IDs.

Three: the attack. As Grace, fetch Ada's note by ID.

```
curl -s -H "Authorization: Bearer $GRACE" \
  localhost:8080/api/v1/notes/1 | jq
```

You should get 404. Not 403, and not the note. Check `/studio` to confirm the row really is there, so you know the 404 is scoping rather than absence.

Four: the list. `GET /notes` as each user returns one note each, and `meta.total` says 1 rather than 2. A count that leaks the true total is a subtler version of the same bug, so it is worth checking.

Five: the back door. Try the same through every other surface. Export the notes as Grace and see whose appear. Run the CSV import as Grace with a row referencing Ada's note. If the scope only lived in the handler, one of these would leak; it does not, because it lives in the service.

Six: break it deliberately. In `services/note.go`, change one method from `s.db(ctx)` to `s.DB` directly. Rebuild, and repeat step three. You will now get Ada's note.

That one character is the difference between an application and an incident, and having caused it on purpose once is the best way to recognise it in review.

YOUR TURN

Write the step-three test as a real Go test, with two users and an assertion that the second gets `ErrNotFound`. Then put it in CI.

An isolation test is the single highest-value test in a multi-user application, because the bug it catches is silent, is not a crash, and is the one you cannot apologise your way out of. Grit's own repository keeps one for exactly this reason.

ASSIGNMENT

Try to break your own isolation, then stop yourself from ever regressing it.

One. Generate a resource with `--owned-by user`, and create one record as each of two users.

Two. As the second user, request the first user's record by ID. You want a 404. Confirm in `/studio` that the row is there.

Three. Try the same through every other door: the list total, the export, the import, a background job. One of them is where a hand-written application would have leaked.

Four. Break it on purpose, in one character, and watch the data appear. Then put it back.

Five. Write the isolation test as a Go test and add it to CI. In a multi-user application it is the highest-value test you will ever write, because the bug it catches is silent.

COMMIT AS

```
git commit -m "Auth - Assignment One"
```

36

The shared package: one definition, four outputs

D3 was "the same type exists four times". A `Blog` is a Go struct, a database column list, a TypeScript interface and a validation schema, and nothing makes them agree except the person who remembers to change all four.

This chapter is about the place where three of those four stop being separate things, and about the one command that keeps them that way.

What is in it

```
packages/shared/
├── package.json
├── types/          TypeScript interfaces, one file per model
├── schemas/        Zod schemas, one file per model
├── constants/      API routes and shared constants
├── themes.ts       the palette both frontends read
└── brand.config.ts the product name, logo and copy
```

It is a workspace package rather than a published one, so there is no build step and no version to bump. The frontends import it by subpath:

```
"exports": {
  "./schemas":   "./schemas/index.ts",
  "./types":     "./types/index.ts",
  "./constants": "./constants/index.ts",
  "./brand":     "./brand.config.ts",
  "./themes":    "./themes.ts"
}
```

```
import { CreateBlogSchema } from "@repo/shared/schemas"
import type { Blog } from "@repo/shared/types"
```

Everything that renders imports from here: the web app, the admin panel, and the desktop and mobile clients if the project has them. One definition, four consumers.

A generated pair, in full

Given this model:

```
type Blog struct {
    ID          string     `json:"id"          gorm:"primaryKey"`
    Title       string     `json:"title"       binding:"required"`
    Slug        string     `json:"slug"`
    Content     string     `json:"content"     gorm:"type:text"`
    Image       *string    `json:"image"`
    Published   bool       `json:"published"`
    PublishedAt *time.Time `json:"published_at"`
    CreatedAt   time.Time  `json:"created_at"`
    UpdatedAt   time.Time  `json:"updated_at"`
}
```

you get `types/blog.ts`:

```
export interface Blog {
  id: string;
  title: string;
  slug: string;
  content: string;
  image: string | null;
  published: boolean;
  published_at: string | null;
  created_at: string;
  updated_at: string;
}
```

and `schemas/blog.ts`, which is three schemas rather than one:

```
export const BlogSchema = z.object({
  id: z.string(),
  title: z.string(),
  slug: z.string(),
  content: z.string(),
  image: z.string().nullable(),
  published: z.boolean(),
  published_at: z.string().nullable(),
  created_at: z.string(),
  updated_at: z.string(),
});

export const CreateBlogSchema = z.object({
  title: z.string().min(1, "Title is required"),
  content: z.string().optional(),
  image: z.string().optional(),
  published: z.boolean().optional(),
```

```
});

export const UpdateBlogSchema = CreateBlogSchema.partial();
```

Why three, and what each one is for

`BlogSchema`	The row as the API returns it. Every field, including the ones the server assigns
`CreateBlogSchema`	What a `POST` accepts. No `id`, no timestamps, and `required` where the Go binding tag said so
`UpdateBlogSchema`	The create schema, `.partial()`. A `PATCH` sends the fields that changed and nothing else

`id`, `created_at`, `updated_at` and `deleted_at` are dropped from the create schema automatically. They are assigned by the server, so a form that offers them is a form that can lie.

> **IF YOU KNOW JAVASCRIPT**
> This is the shape you would have built by hand in a Next.js project, and the part that goes wrong by hand is `UpdateSchema`.
>
> People reuse the create schema for editing, so every `PATCH` has to send every field, so an edit form that was missing one input quietly blanks that column. `.partial()` is the fix, and having it generated means you cannot forget it on the eleventh resource.

nullable against optional, which is not pedantry

`image` is `z.string().nullable()` in the row schema and `z.string().optional()` in the create schema, and the difference is the difference between two states that look the same in a form:

- **nullable**: the server will send `"image": null`. The key is there, the value is empty.
- **optional**: the client may leave the key out entirely, and the server applies its default.

A Go pointer field (`*string`, `*time.Time`) becomes nullable, because a pointer is how Go says "this may be absent". A non-required field becomes optional on create. Get these the wrong way round by hand and you get a form that refuses to submit because a column the user cannot see is `null`.

grit sync, and exactly what it does

```
grit sync
```

It parses every file in `apps/api/internal/models/` with Go's own parser, not a regular expression, and for each struct it writes:

`types/<name>.ts`	The interface, overwritten
`schemas/<name>.ts`	The three schemas, overwritten
the admin resource	New fields added to the generated section only

The name is kebab-cased from the struct: `BlogPost` becomes `blog-post.ts`.

`User` is skipped. It has hand-written schemas covering login, registration, password reset and profile updates, which are not derivable from the struct, and regenerating them would throw away work the framework cannot reproduce.

The mapping, for the types you will actually hit:

Go	TypeScript	Zod
`string`	`string`	`z.string()`
`int`, `int64`	`number`	`z.number().int()`
`uint`	`number`	`z.number().int().nonnegative()`
`float64`	`number`	`z.number()`
`bool`	`boolean`	`z.boolean()`
`time.Time`	`string`	`z.string()`
`*time.Time`	`string | null`	`z.string().nullable()`

Times cross as strings, because JSON has no date type. Chapter nineteen covers the format; the thing to hold onto here is that the TypeScript type says `string` and means RFC 3339, and the parsing is the frontend's job.

WATCH OUT

`grit sync` overwrites those two files completely. There is no merge and no manifest conflict: the generated type and the generated schema are outputs, and an output is rewritten.

So a refinement you add inside `schemas/blog.ts`, a `.refine()` for a cross-field rule or a stricter message, is gone the next time somebody adds a column. Put it in a file beside it and compose:

```
// schemas/blog-form.ts, which sync does not touch
export const BlogFormSchema = CreateBlogSchema.extend({
  slug: z.string().regex(/^[a-z0-9-]+$/, "lowercase and dashes"),
}).refine((v) => !v.published || v.content.length > 0, {
  message: "a published post needs content",
```

```
  path: ["content"],
})
```

The rule for the whole framework, stated once: **edit the inputs, extend the outputs**. The model is an input. The shared schema is an output.

The admin resource is updated, not overwritten

The third thing `sync` touches is the admin resource definition, and it is treated differently because it is yours.

The generated file carries markers around the section the generator owns. New model fields are added inside those markers, to the columns and the form. Anything you customised, a renamed label, a `render` function, a reordered column, sits outside them and survives.

So the usual loop after changing a model is two commands:

```
grit migrate
grit sync
```

and the admin has the new field in the table and the form without you opening the file.

Where the chain stops, honestly

`sync` keeps three things in step: the type, the schema, and the admin resource. It does not update the generated **mobile** forms or the **desktop** columns, because those have no marker scheme yet.

It says so rather than pretending:

```
⚠ 2 field(s) never reached apps/mobile:
  reading_notes, verified_at
```

That warning exists because the failure without it is miserable to diagnose: a correct API, a correct type, a correct admin screen, and a mobile create form frozen at the three fields the resource was generated with. The field is in the response and absent from the form, and nothing is broken enough to show an error.

WATCH OUT
Treat that warning as a task, not as noise. The fix is to open the named screen and add the field by hand, which is five minutes, or to regenerate the screen if you have not customised it.

> A framework that quietly half-synced would be worse than one that does not sync at all, because you would trust the half.

Using it on the frontend

The schemas are not decoration. They are the validation layer of every generated form:

```
const form = useForm<CreateBlogInput>({
  resolver: zodResolver(CreateBlogSchema),
})
```

The inferred input type comes from the same schema:

```
export type CreateBlogInput = z.infer<typeof CreateBlogSchema>;
```

so renaming a field in the Go model and running `grit sync` makes the editor underline every place the old name is used, before anything runs. That is the actual payoff of D3: not that the four definitions exist in one place, but that disagreeing with them is a compile error rather than a support ticket.

The error codes travel the same way. `types/errors.ts` is generated from the Go error catalogue, so a frontend can match on a code rather than on a message:

```
if (apiErrorCode(err) === "EMAIL_EXISTS") {
  form.setError("email", { message: "That email is taken" })
}
```

Constants, themes and brand

Three smaller files, each solving a small duplication that gets annoying at the third app:

- `constants/` holds the API route paths, so a URL typo is a type error rather than a 404 at runtime.
- `themes.ts` holds the palette, read by the admin, the web app and the desktop client, so a theme change lands everywhere at once.
- `brand.config.ts` holds the product name, logo and marketing copy, which is why a generated project shows **your** name rather than Grit's.

> **PROJECT 33: BREAK THE CHAIN AND WATCH IT FAIL**
> The point of this project is to feel what the single definition is worth, by removing it.

```
grit generate resource Reading \
  --fields "label:string, value:float, taken_at:datetime"
grit migrate
```

One: the normal loop. Add a field to the Go model by hand:

```
Verified bool `json:"verified"`
```

Run `grit migrate` then `grit sync`. Open `packages/shared/types/reading.ts`, the schema, and the admin resource file. Three files changed from one edit.

Two: skip the sync. Add a second field and run only `grit migrate`. The column exists, the API returns it, and the TypeScript type does not have it. Try to read it in a page and watch the compiler stop you. That error is the framework working, not failing.

Three: the overwrite. Add a `.refine()` inside the generated `schemas/reading.ts`, then run `grit sync` again. It is gone. Now move it to a sibling file as the gotcha above shows, and run sync a third time.

Four: the drift warning. If the project has a mobile or desktop client, add a third field and read what sync prints. Open the named screen and confirm the field really is missing.

Five: the payoff. Rename `label` to `name` in the Go model, migrate, sync, and then run `pnpm typecheck`. Every place that used the old name is now an error with a file and a line. Count them, and ask how you would have found them in a project where the type lived in two languages.

YOUR TURN

Open `packages/shared/schemas/index.ts` and read the exports. Every generated resource adds three schemas and two inferred types to that barrel.

Then ask the question that decides whether this package earns its place: if your application has one frontend and will never have another, is the shared package still worth it? The honest answer is that it is worth it the moment there is a second consumer of the type, and that a mobile app, a desktop client and an admin panel are each a second consumer.

ASSIGNMENT

Make the single definition prove itself on your own project.

One. Add two fields to an existing model, run `grit migrate` then `grit sync`, and commit the diff. In `NOTES.md`, list every file that changed from those two edits and mark which were generated and which were updated in place.

Two. Write a cross-field rule that the generator cannot infer: a published post needs content, an end date must follow a start date, a discount cannot exceed the total. Put it in a sibling schema file, use it in a form, and then run `grit sync` and show that it survived.

Three. Rename a field in the Go model. Migrate, sync, and run the TypeScript check. Paste the compiler output into `NOTES.md`: that list is the thing D3 buys you, and it is worth seeing written out.

Four. One paragraph on the honest limit: which clients did `sync` not update, what did it tell you, and what would you have to do by hand?

COMMIT AS

```
git commit -m "Shared Package - Assignment One"
```

37

Project: a task tracker, end to end

Part IV explained the pieces. This chapter uses them, in order, to build something a small team could actually run: projects, tasks with a status and an assignee, comments, a dashboard, and a web page that is not the admin panel.

Everything here is a command you type and a file you open. Nothing is skipped for space, and the parts that are genuinely tedious are called tedious rather than hidden.

What we are building

A tracker with three models and one rule that matters:

`Project`	A name, a colour, an owner. Archived rather than deleted
`Task`	Belongs to a project, has a status, an assignee and a due date
`Comment`	Belongs to a task, written by a user

The rule: **a task cannot be closed while it has open subtasks**, which is the kind of thing that is trivial to state, easy to enforce in one place, and impossible to enforce if it lives in a form.

Step one: the project

```
grit new tracker
cd tracker
grit start
```

Three applications come up: the API on `:8080`, the web app on `:3000` and the admin on `:3001`. Log in with the seeded admin account printed in the output, and confirm the dashboard renders before generating anything. A broken starting point is worth five minutes to rule out.

> **WATCH OUT**
> If the admin loads and the lists are empty while the API logs show nothing, the usual cause is that migrations have not run. `grit migrate` once, then reload.

The second most common cause is a port already in use from an earlier project, which shows up as a login that posts and does nothing because `:8080` belongs to somebody else. `grit doctor` reports both.

Step two: the resources

Generate the parent first, because the child refers to it:

```
grit generate resource Project \
  --fields "name:string, colour:color, archived:bool" \
  --owned-by user
```

Then the task, with the relationship and a real status:

```
grit generate resource Task --fields "\
title:string, \
notes:richtext, \
status:select:todo=To do|doing=In progress|done=Done, \
priority:select:low=Low|normal=Normal|high=High, \
due:date, \
project:belongs_to:Project, \
assignee:belongs_to:User"
```

And the comment:

```
grit generate resource Comment \
  --fields "body:text, task:belongs_to:Task"
```

Then once:

```
grit migrate
```

Reload the admin. Three new sections in the sidebar, each with a working list and form. The task form has two dropdowns with labels rather than raw values, a date picker, a rich text editor, and two searchable relationship pickers.

IF YOU KNOW JAVASCRIPT

Count what you did not do. No table component, no form library wiring, no `<select>` with hand-written options, no fetch hook, no TypeScript interface, no Zod schema, no migration file, no route registration.

That is chapter twenty nine's claim arriving. The reason it is worth distrusting until you have done it is that most tools make the same claim and stop at the list page.

Step three: the rule, in the only place it can live

Open `apps/api/internal/services/task.go` and add a method. This is chapter thirty two, applied:

```
var ErrHasOpenSubtasks = errors.New(
    "task has open subtasks")

func (s *TaskService) Close(ctx context.Context, id uint) (
    *models.Task, error) {

    task, err := s.GetByID(ctx, id)
    if err != nil {
        return nil, err
    }

    var open int64
    if err := s.db(ctx).Model(&models.Task{}).
        Where("parent_id = ? AND status != ?", id, "done").
        Count(&open).Error; err != nil {
        return nil, fmt.Errorf("counting subtasks: %w", err)
    }
    if open > 0 {
        return nil, ErrHasOpenSubtasks
    }

    task.Status = "done"
    if err := s.db(ctx).Save(task).Error; err != nil {
        return nil, fmt.Errorf("closing task %d: %w", id, err)
    }
    return task, nil
}
```

The subtask relationship needs a column, which is one command rather than a migration file:

```
grit generate field Task parent:belongs_to:Task
grit migrate
```

A self-referencing `belongs_to` is legal and useful: it is how a task gets subtasks without a second model.

> **WATCH OUT**
> `s.db(ctx)` rather than `s.DB`. The first is scoped by the actor in the context, the second is not.

> On a `--owned-by user` resource that one difference decides whether the subtask count includes other people's tasks. It will look correct in your own testing, because you are the only user, which is exactly why this is the bug that reaches production.

Step four: the endpoint

```
func (h *TaskHandler) Close(c *gin.Context) {
    id, err := strconv.ParseUint(c.Param("id"), 10, 64)
    if err != nil {
        respond.BadRequest(c, "invalid id")
        return
    }

    task, err := h.service().Close(h.ctx(c), uint(id))
    if err != nil {
        respond.Fail(c, err)
        return
    }
    respond.OK(c, task)
}
```

Register it next to the generated routes:

```
tasks.POST("/:id/close", h.Task.Close)
```

And teach the boundary what the new error means, so the frontend gets a 409 rather than a 500:

```
case errors.Is(err, services.ErrHasOpenSubtasks):
    respond.Conflict(c, err)
```

Check it is wired:

```
grit routes
```

That prints every registered route with its method, handler and middleware group, which is the fastest way to catch a route registered on the wrong group and therefore unauthenticated.

Step five: data worth looking at

An empty application tells you nothing about whether it works:

```
grit seed Project --count 8
grit seed Task --count 400
grit seed Comment --count 1200
```

Four hundred tasks is where the list page stops being a demo. Sort by due date, filter by status, search a title, and page through. Pagination is server side, so the browser is holding twenty rows no matter how many exist.

> **YOUR TURN**
> Seed a hundred thousand tasks and try the same screen. It stays quick, because the query is doing the work rather than the client.
>
> Then sort by a column the service did not whitelist and watch nothing happen. That is chapter thirty two's whitelist, and it is also the reason the first sort you add by hand appears to be ignored.

Step six: the admin screen, customised

The generated list is correct and plain. Two changes make it usable, and both live in `apps/admin/app/_dashboard/resources/task/index.tsx`.

A status badge rather than a word:

```
{
  key: "status",
  label: "Status",
  sortable: true,
  render: (row) => {
    const tone = {
      todo:  "bg-slate-500/15 text-slate-300",
      doing: "bg-amber-500/15 text-amber-400",
      done:  "bg-emerald-500/15 text-emerald-400",
    }[row.status]
    return <span className={`rounded px-2 py-0.5 ${tone}`}>
      {row.status}</span>
  },
}
```

And an overdue due date, because a date that is late should look late:

```
{
  key: "due",
  label: "Due",
  sortable: true,
  render: (row) => {
    const late = row.status !== "done" &&
      new Date(row.due) < new Date()
```

```
    return <span className={late ? "text-red-400" : ""}>
      {formatDate(row.due)}</span>
  },
}
```

Re-run the generator for `Task` afterwards and confirm these survive. They are in the file the generator writes once.

Step seven: a page the admin cannot be

An operator console is not a product. The team wants a board, so write one as an ordinary Next.js page using the generated hooks:

```
export default function BoardPage() {
  const { data, isLoading } = useTasks({ page_size: 200 })
  if (isLoading) return <BoardSkeleton />

  const columns = ["todo", "doing", "done"] as const
  return (
    <div className="grid gap-4 md:grid-cols-3">
      {columns.map((status) => (
        <Column
          key={status}
          title={LABELS[status]}
          tasks={data.data.filter((t) => t.status === status)}
        />
      ))}
    </div>
  )
}
```

Moving a card calls the same API the admin does:

```
const { mutate } = useUpdateTask()
mutate({ id: task.id, status: "doing" })
```

Nothing here is framework-specific. It is React against typed hooks, and the types came from the Go model through chapter thirty six's shared package, so renaming `due` in the model makes this page fail to compile rather than fail in front of a user.

> **WATCH OUT**
> Resist putting the "cannot close with open subtasks" check in this page. It is one line of JavaScript and it is wrong, because the board is not the only caller.
>
> Call the endpoint, show the 409's message, and let the rule live once. The test is always chapter twenty eight's: would it still have to be true if a nightly job did it?

Step eight: a form for people without accounts

Bug reports should not require an admin login. One command writes the page:

```
grit expose form Task --to apps/web/app/report/page.tsx
```

That emits a Next.js page with one input per field, submitting through the authenticated create hook. For a genuinely public form, create a form share in the admin under System, then:

```
grit expose form Task --to apps/web/app/report/page.tsx \
  --public-share --token 9CkLh7gJZ...
```

which posts to the public submission endpoint instead, with no account required.

> **WATCH OUT**
> A public write endpoint is a spam target the hour it is discovered. The share token is revocable in the admin, and the form share is rate limited, but the decision about what a stranger may create is yours.
>
> The safer shape for a bug reporter: a public form that creates a `Report`, and a human who turns a report into a `Task`. One extra model, and no stranger ever writes to the table the team works from.

Step nine: the things that make it a product

Four small commands, each closing something a client will ask about:

```
grit add role MANAGER
grit backup
grit doctor
grit routes
```

A role that the sidebar, the forms and the seeded permissions all know about. A backup you have actually taken. A health check across the project. A route list you have read, looking for anything registered outside the authenticated group.

What it cost

Count honestly, including the parts nobody counts:

Three resources, migrated, in the admin	four commands
The business rule and its endpoint	about 40 lines
Two customised columns	about 25 lines
The board page	about 80 lines

Everything else: auth, lists, forms, validation, types, hooks	generated

Compare that with chapter forty eight, where one resource with no admin panel and no frontend took about six hundred lines. That ratio is the entire argument of this book, and now you have both numbers from your own machine rather than from a marketing page.

PROJECT 34: FINISH THE TRACKER

Take the tracker from working to usable. Four additions, each of which uses a different part of Part IV.

One: activity. Generate a `Activity` resource with `--append-only` and write a row whenever a task changes status. Append-only means no edits and no deletes, which is what an audit trail is.

Two: my work. Add a page listing the signed-in user's open tasks across projects, ordered by due date. Use the generated hook with a filter rather than fetching everything and filtering in the browser, and explain in a comment why that distinction matters at ten thousand rows.

Three: a dashboard widget. Add a stat card showing overdue tasks, and a chart of completions per week. Both read from the API, not from the browser's copy.

Four: the rule, tested. A Go test that creates a task with one open subtask and asserts `Close` returns `ErrHasOpenSubtasks`. Then a second test that closes the subtask first and asserts success. Run them in CI.

Then delete something on purpose: `grit remove resource Comment`, confirm the project still builds, and read what the command removed. A generator that cannot take its own work back is a generator you cannot experiment with.

YOUR TURN

Open the tracker in a browser as two different users in two windows: an admin and an ordinary member. Walk the same screens.

The differences you see are chapter thirty five working. The ones you expected and do not see are permissions you have not configured yet, and it is much better to find them here than in a client demo.

ASSIGNMENT

Build the tracker in your own repository, under `part5-building/37-tracker`, and push it as a working project rather than a set of notes.

One. The three resources, the subtask relationship, the `Close` rule, its endpoint and its 409. The rule must be enforced from a Go test that never touches HTTP.

Two. The board page, with the status badge and the overdue styling in the admin resource file.

Three. Seed four hundred tasks and record, in `NOTES.md`, the time to first paint on the list page and what changes at a hundred thousand rows.

Four. `grit routes` output committed as `ROUTES.txt`, with one line naming any route you would not want public.

Five. The honest comparison. Count the lines you wrote yourself, put them next to the chapter forty eight number, and write a paragraph on which parts of the generated code you would keep if you had to maintain this for five years.

COMMIT AS

```
git commit -m "Task Tracker - Assignment One"
```

38

Relationships, and the forms they generate

A model on its own is a spreadsheet. The moment a task belongs to a project and carries tags, you have the thing that makes applications hard: two tables that must agree, a form that has to offer the right options, a query that must not run once per row, and a JSON shape the frontend can rely on.

Grit has four relationship shapes. This chapter is what each one writes, and where each one bites.

The four shapes, and the question each answers

`belongs_to`	This row points at one other row. A task has a project
`one_to_one`	The same, but only one row may point at each target. A user has one profile
`many_to_many`	Both sides have many. A task has tags, a tag has tasks
`--items`	A child that exists only inside its parent. An invoice has line items

The fourth is not a field type, and the difference is worth naming early: an `--items` child has no life of its own. You do not browse line items; you edit an invoice. Everything else is a relation between two things that each exist.

belongs_to, and everything it writes

```
grit generate resource Task \
  --fields "title:string, project:belongs_to:Project"
```

The related model is the third part, and it is optional when the field name already names it: `project:belongs_to` finds `Project`. Spell it out when the field is called something else, which is the usual case for people: `author:belongs_to:User`.

In the model you get two things, not one:

```
type Task struct {
    ID        uint     `json:"id" gorm:"primaryKey"`
    Title     string   `json:"title"`
    ProjectID uint     `json:"project_id" gorm:"index"`
    Project   *Project `gorm:"foreignKey:ProjectID"
                        json:"project,omitempty"`
}
```

`ProjectID` is the column the database knows about, indexed because every listing filters on it. `Project` is the association GORM fills in when asked, and it is a **pointer with `omitempty`**.

> **WATCH OUT**
> That pointer is not a style choice, and the version before it shipped a convincing bug.
>
> As a value, a relation that was never preloaded marshalled as a complete zero-value object: `"project": {"id": 0, "name": "", "colour": ""}`. To a client that reads as a real project whose every field happens to be blank, and the generated TypeScript agreed, because it said the object was always there.
>
> As a pointer with `omitempty`, an absent relation is absent. The frontend can tell "not loaded" from "loaded and empty", which are different facts and used to be the same one.

The service preloads it, so a list of tasks arrives with its projects:

```
db.Preload("Project").Find(&tasks)
```

That is chapter twenty two's N+1 solved by default: one query for tasks, one for the projects they reference, rather than one per row. You get it without asking, which is also why it is worth knowing it is there when a list of twelve columns suddenly reads from five tables.

The admin dropdown, and the label it shows

The form field for a `belongs_to` is a searchable dropdown, and the thing that makes it usable is what it displays. It is not the ID, and it is not a field called whatever came first. The generator looks for a label column on the related model, in this order:

```
name, title, label, subject, number, code, reference, email
```

The first one that exists wins. A `Project` shows its name, an `Invoice` shows its number, and a `User` with none of those falls back to its email, which is the reason the fallback list ends there rather than at `code`.

IF YOU KNOW JAVASCRIPT
Every admin panel you have built by hand has had this bug: a select showing `[object Object]`, or a list of IDs, or a dropdown that loads six thousand rows to find one. The generated one searches server side and shows a human label. It is a small thing that is tedious to get right once per relation, and there are usually twenty relations.

one_to_one is belongs_to plus one index

```
grit generate resource Profile \
  --fields "bio:text, user:one_to_one:User"
```

The generated code differs from `belongs_to` in exactly one place: the foreign key gets a unique index rather than an ordinary one.

That single line is what makes the name true. Without it, "one profile per user" is a comment, because the database would happily accept a second row pointing at the same user, and the second one usually arrives from a retried request rather than from a user doing something strange.

A relation pointing at itself is a tree

```
grit generate field Category parent:belongs_to:Category
```

Electronics above Cameras above Lenses, in one table. Two details differ from an ordinary relation, and both were found the hard way.

The association must be a pointer. Go rejects a struct that contains itself by value: `invalid recursive type`. A generated project with a self-reference did not compile at all before this was handled.

The foreign key must be nullable. GORM creates a real foreign key constraint, and a root category has no parent, so the column has to hold something the constraint accepts. SQL has exactly one such value, and it is `NULL`.

WATCH OUT
An earlier version used the empty string for "no parent", matching what every other `belongs_to` field does. It passed every test and failed on the first real project:

```
ERROR: insert or update on table "categories" violates
foreign key constraint "fk_categories_children"
```

The tests ran on SQLite, which does not enforce foreign keys unless you ask it to. Postgres does. The generated tests now switch foreign keys on for exactly this reason,

and the general lesson is larger than trees: **a test database that is more permissive than production is a test database that hides this class of bug entirely**.

For a real hierarchy, the flag does more than the field:

```
grit generate resource Category --fields "name:string" --tree
```

That gives the self-referential parent plus a materialized path, a depth, a sibling order, the tree queries and a move endpoint. The difference matters when you need "everything under Electronics" as one query rather than as recursion in application code.

many_to_many, and the table you did not name

```
grit generate resource Task \
  --fields "title:string, tags:many_to_many:Tag"
```

The related model is required here. There is nothing to infer from, because `tags` is a plural of the field, not of the model.

The model gets a slice and no foreign key, because the keys live in a third table:

```
Tags []Tag `gorm:"many2many:tasks_tags" json:"tags"`
```

The join table is named from the owning model and the field: `tasks_tags`. GORM creates and maintains it, and you do not get a model for it.

WATCH OUT

The moment you need a column **on the relationship itself**, a quantity, a role, a date added, `many_to_many` is the wrong shape and no flag fixes it.

"A user belongs to many projects" is many-to-many. "A user belongs to many projects **as owner or as member**" is a `Membership` model with two `belongs_to` fields and a `role`, generated like any other resource.

Reaching for the join table as a place to put data is the single most common modelling mistake in this chapter, and it is expensive because the migration that fixes it has to move rows.

In the admin this renders as a multi-select picker, and in JSON it is an array, which means a create or update sends IDs:

```
{ "title": "Fix the leak", "tag_ids": [3, 7] }
```

Line items: a child inside its parent

```
grit generate resource Invoice \
  --fields "number:string, client:belongs_to:Client" \
  --items "InvoiceItem:description:string,qty:int,unit:money"
```

The child is generated fully, and the parent's form gets a line-items table: add a row, remove a row, totals that update. That is a component people build by hand for a week.

The part worth understanding is the update, because it is not what you would write first:

```
// Inside the same transaction as the parent update
tx.Where("invoice_id = ?", item.ID).
    Delete(&models.InvoiceItem{})
tx.Create(&rows)
```

The children are deleted and recreated rather than diffed.

> **WATCH OUT**
> Delete-and-recreate is the right call for line items and the wrong call for anything with its own identity. New IDs on every save means anything pointing at a line item now points at nothing.
>
> So the test for `--items` is: **would anybody ever link to this row?** An invoice line, no. A comment on a task, yes, and a comment is therefore a resource with a `belongs_to`, not an item.
>
> Note also that the delete and the insert are inside one transaction. Without that, a failure between them leaves an invoice with no lines, which is a document that silently became worth nothing.

What relationships cost on a list page

Three consequences that surprise people, all of them from chapter thirty two's whitelists:

You cannot sort by a relation. The sortable list contains columns, and `project.name` is not a column of `tasks`. Sorting by a related field needs a join the generated query does not make. Sort by `project_id` if order matters, or denormalise the name onto the row if it matters a lot.

You filter by the key, not the label. `?project_id=4` works. `?project=Apollo` does not, unless you add it.

Search does not cross the relation. Searching tasks searches the task's own searchable columns. Finding "every task whose project is named Apollo" is a query you write.

IF YOU KNOW JAVASCRIPT

These are not framework limitations so much as the honest shape of SQL showing through. An ORM that silently joined four tables to satisfy a sort parameter would be faster to demonstrate and slower in production, and you would find out at the worst moment.

The framework's choice is to make the fast thing automatic and the expensive thing explicit.

Choosing well

Two questions settle most modelling arguments.

Is the other side a real thing, or a word? A status is a word: `select`, not a relation. A supplier is a thing: a relation. The giveaway is whether anybody would ever want to edit it, list it, or attach anything else to it.

Does the link itself carry data? If yes, it is a model, not a `many_to_many`.

PROJECT 35: MODEL SOMETHING GENUINELY AWKWARD

Build a small school timetable, which has every shape in this chapter and one real trap.

```
grit new timetable
cd timetable
```

One: the obvious ones.

```
grit generate resource Teacher --fields "name:string, email:email"
grit generate resource Room --fields "code:string, seats:int"
grit generate resource Course \
  --fields "title:string, teacher:belongs_to:Teacher"
```

Open the Course form and confirm the teacher dropdown shows names, not IDs.

Two: the trap. Model "a student attends many courses" as a `many_to_many`. Then try to answer: when did they enrol, and did they pass?

You cannot, so replace it with an `Enrolment` resource that belongs to both and carries `enrolled_on:date` and `grade:select`. Write the two commands and the migration, and note in `NOTES.md` what you would have had to do if the first version had been live for a year.

Three: the tree. Give `Course` a `prerequisite:belongs_to:Course`. Create a root with no prerequisite and confirm the column stores `NULL`. Then run the same thing against Postgres rather than SQLite:

```
docker compose up -d db
```

Four: line items. Generate a `Timetable` with `--items "Slot:day:select:mon=Mon|tue=Tue,period:int,room:belongs_to:Room"`. Add three slots, save, and watch the IDs change. Then write one sentence on why that is acceptable here and would not be for enrolments.

Five: the cost. Load the Course list with 500 rows and look at the API log. Count the queries. Then remove the `Preload` from the service, reload, and count again. That difference is the N+1, measured.

YOUR TURN

Take the data model of something you have already built, and write down each relationship as one of the four shapes in this chapter.

The ones you cannot classify are the interesting ones. In my experience they are almost always a join table that grew a column, which is the modelling mistake this chapter exists to name.

ASSIGNMENT

In `part5-building/38-relationships`, build a small library system with every shape, and prove the behaviour rather than describing it.

One. `Author`, `Book` with `author:belongs_to:Author`, `Member`, and a `Loan` that belongs to both a book and a member and carries `due:date`. Explain in one line why `Loan` is a model and not a `many_to_many`.

Two. Add `categories:many_to_many:Category` to `Book`. Find the join table in `/studio`, record its name, and paste one row into `NOTES.md`.

Three. A self-referential `Category` with a parent. Create a root, confirm the stored value is `NULL` and not an empty string, and say what would break on Postgres if it were.

Four. The JSON evidence. Fetch one book with its author preloaded, and one endpoint where it is not. Paste both responses: one has an `author` object, the other has no `author` key at all. That difference is the pointer and the `omitempty`, and it is the one thing in this chapter a frontend developer will thank you for.

Five. Try to sort the book list by author name through the API. Record what happens, then make it work, and write down what you had to do.

COMMIT AS

```
git commit -m "Relationships - Assignment One"
```

39

Project: a publication, from draft to public page

Every project in Part V so far has been behind a login. This one is not, and that changes almost everything: the audience is the internet, the content is written by people rather than generated, and the pages have to be fast for somebody who has never signed in and never will.

We are building a publication: posts with a rich text body, images, a draft and published workflow, a public API and a web front end that renders it.

The model you already have

A new project ships with a `Blog` model, and it is worth reading before replacing it, because it is a worked example of several chapters at once:

```
type Blog struct {
    ID          string     `gorm:"primarykey;size:36"`
    Title       string     `gorm:"size:255;not null"
                           binding:"required"`
    Slug        string     `gorm:"size:255;uniqueIndex"`
    Content     string     `gorm:"type:text" sanitize:"html"`
    Image       string     `gorm:"size:500"`
    Excerpt     string     `gorm:"size:500"`
    Published   bool       `gorm:"default:false"`
    PublishedAt *time.Time
    Version     int        `gorm:"not null;default:1"`
    DeletedAt   gorm.DeletedAt `gorm:"index" json:"-"`
}
```

Five decisions in one struct:

`uniqueIndex` on slug	The URL is a key, and the database enforces it
`sanitize:"html"`	The body is cleaned on every write. See below
`Published` plus `PublishedAt`	State and the moment it changed
`Version`	Optimistic locking, from chapter thirty three

`DeletedAt`	Soft delete, so an unpublish is never a loss

The slug is generated in a `BeforeCreate` hook, which is the right place for it: a title is typed once and the slug must exist whichever surface created the row, including a seeder and an import.

The one that is not optional: sanitising HTML

A rich text editor produces HTML, and HTML from a user is an attack surface with a text box attached. The tag says what to do about it:

```
Content string `gorm:"type:text" sanitize:"html"`
```

Every write that has a model passes through the sanitiser, so a `<script>` in a draft does not survive being saved. Not on render, on **write**, which matters because the content will be read by more things than your web page: an RSS feed, a mobile app, a newsletter.

> **WATCH OUT**
> Sanitising on render rather than on write is the mistake that keeps coming back, because it works perfectly until the second reader exists. The database holds the attack, and every new consumer has to remember to defuse it.
>
> Sanitise once, at the boundary where the data arrives. Then the thing in the database is safe, and the number of places that have to be careful is one rather than one per feature.

Draft and published, as a service rule

Publishing is not a column edit. It is a transition with consequences:

```
func (s *BlogService) Publish(ctx context.Context,
    id string) (*models.Blog, error) {

    post, err := s.GetByID(ctx, id)
    if err != nil {
        return nil, err
    }
    if strings.TrimSpace(post.Content) == "" {
        return nil, ErrEmptyPost
    }
    if post.Published {
        return post, nil  // idempotent on purpose
    }

    now := time.Now().UTC()
    post.Published = true
    post.PublishedAt = &now
```

```
    if err := s.db(ctx).Save(post).Error; err != nil {
        return nil, fmt.Errorf("publishing %s: %w", id, err)
    }
    events.Emit(ctx, events.BlogPublished{ID: post.ID})
    return post, nil
}
```

Three things in that function are worth copying into whatever you build next.

It refuses an empty post. The rule lives here, so the admin, the API and a scheduled publish all obey it.

It is idempotent. Publishing something already published returns it rather than erroring. Retries happen, double clicks happen, and a second publish should not move `PublishedAt`.

It emits an event rather than doing the downstream work inline. Chapter forty four is about the bus; what matters here is that "tell the newsletter" is not the publishing function's business.

The public surface

The generated endpoints are behind authentication, which is right for an admin and useless for readers. A public-facing resource asks for a second, narrower surface:

```
grit generate resource Article \
  --fields "title:string, slug:slug, body:richtext, \
cover:file, published:bool" --public
```

That adds a list and a get-by-slug under `/api/v1/public/`, mounted outside the auth middleware. The protected routes are untouched, because the admin still calls those.

Two properties of that surface are the whole design.

The response is an allowlist, not the model. A public response is its own struct, built from the fields that are publishable by default: names, slugs, descriptions, images. Anything whose name contains `cost`, `margin`, `profit`, `internal`, `note`, `secret`, `token`, `supplier` or `wholesale` is excluded whatever its type, and foreign keys and stock counts are left out too.

A column added next month is private until somebody says otherwise. That is the opposite default to the admin surface, and it is the right one when the audience is the internet: the alternative, publishing everything and trusting the developer to remove what should not be there, puts a `cost_price` column on the web the first time somebody adds one in a hurry.

To publish a field that was excluded, you edit one struct, deliberately, in a diff somebody can review.

> **WATCH OUT**
> The public group is guarded by an API key, which is not secrecy: a publishable key ships inside your app where anybody can read it.
>
> What it buys is identification, a rate limit bucket per key, per-endpoint and per-origin narrowing, and the ability to turn one client off without a deploy. Treat it as a name badge rather than a password.

The cache, and the one rule about it

Responses on the public group are cached, and nowhere else. The cache key is the URL and nothing else:

```
publicAPI.Use(middleware.CacheResponse(svc.Cache, ttl))
```

On a public endpoint that is exactly right. Every caller gets the same answer, so one cached copy serves all of them and an article page stops hitting the database on every visit.

> **WATCH OUT**
> **Never mount a writing endpoint on the public group, and never move this cache onto an authenticated one.**
>
> The second is the dangerous one: with the URL as the key, `GET /api/v1/invoices` cached for one user is served to the next user who asks. One line of middleware in the wrong group is a data breach that passes every test, because tests usually run one user at a time.
>
> `--public` is read-only by design. A public **write**, a comment form, a newsletter signup, is a hand-written endpoint with its own rate limit and its own validation, and it does not belong on a cached group.

Rendering it in the web app

The public list, in a server component:

```
async function getArticles() {
  const res = await fetch(`${API}/api/v1/public/articles`, {
    headers: { "X-API-Key": process.env.PUBLIC_API_KEY! },
    next: { revalidate: 60 },
  })
  if (!res.ok) throw new Error("articles unavailable")
```

```
  return (await res.json()) as { data: Article[]; meta: Meta }
}
```

And the article page, by slug rather than ID, which is why the public endpoint looks up by slug:

```
export default async function ArticlePage(
  { params }: { params: { slug: string } },
) {
  const article = await getArticle(params.slug)
  return (
    <article className="prose prose-invert mx-auto">
      <h1>{article.title}</h1>
      <div dangerouslySetInnerHTML={{ __html: article.body }} />
    </article>
  )
}
```

> **IF YOU KNOW JAVASCRIPT**
> `dangerouslySetInnerHTML` is doing what its name says, and it is acceptable here for one reason only: the HTML was sanitised on write.
>
> If you are ever tempted to render untrusted HTML that has not been through a sanitiser, the answer is not a careful regular expression. It is a library, on the way in.

Metadata comes from the same fetch, so the social preview and the page agree:

```
export async function generateMetadata({ params }) {
  const a = await getArticle(params.slug)
  return {
    title: a.title,
    description: a.excerpt,
    openGraph: { images: [a.cover] },
  }
}
```

The things a publication needs that nothing generates

Four small pieces, each about twenty lines, none of them framework specific:

A sitemap. `app/sitemap.ts` fetching the public list and returning the URLs with their `published_at` as `lastModified`.

An RSS feed. A route handler returning XML. Readers still use them, and so do aggregators that will send you traffic.

Related posts. The public surface can expose a related endpoint; otherwise it is a query by tag with the current article excluded.

A reading time. Word count divided by two hundred, computed on write and stored, because computing it on every render is work repeated for no reason.

PROJECT 36: SHIP THE PUBLICATION

Build it properly, including the parts that are only visible when something goes wrong.

```
  grit new press && cd press
  grit generate resource Article \
    --fields "title:string, slug:slug, body:richtext, \
cover:file, excerpt:text, published:bool, \
author:belongs_to:User, tags:many_to_many:Tag" --public
  grit generate resource Tag --fields "name:string, slug:slug" --public
  grit migrate && grit seed Article --count 60
```

One: the workflow. Add `Publish` and `Unpublish` to the service with the rules above, plus endpoints. Publishing twice must not move `published_at`.

Two: the public surface, inspected. Fetch `/api/v1/public/articles` and compare the JSON with the model. List in `NOTES.md` every field that did not come out, and say for each whether you agree with the default.

Three: drafts must not leak. Create a draft, note its slug, and request it from the public endpoint. You should get a 404. Then check the list's `meta.total` and confirm the draft is not counted. A total that leaks the number of unpublished drafts is a smaller version of the same bug.

Four: the cache, observed. Request the list twice and compare response times. Then publish a new article and watch how long it takes to appear. Note the TTL and decide whether sixty seconds is right for your publication.

Five: the attack. Paste this into a draft body through the admin editor, save, and then read the stored row in `/studio`:

```
<img src=x onerror="alert(document.cookie)">
```

Record exactly what the database holds. That is the sanitiser, and seeing it work is more convincing than reading that it exists.

Six: the front end. A list page, an article page by slug, metadata, a sitemap and an RSS feed. Run Lighthouse on the article page and keep the score.

YOUR TURN

Turn the API off and reload your article page.

Whatever happens next is what a reader sees on your worst day. A blank page with a stack trace, a spinner forever, or a cached page with a quiet notice are three different products, and only one of them is a decision rather than an accident.

ASSIGNMENT

In `part5-building/39-publication`, build the publication and prove the boundary between private and public.

One. The `Article` resource with `--public`, the publish and unpublish rules in the service, and tests for both, including the idempotent publish.

Two. `PUBLIC.md` containing: the full model field list, the fields the public response actually returns, and one line each on why the difference is correct or what you changed.

Three. Evidence that a draft is invisible publicly: the 404, and the `meta.total` with and without drafts.

Four. The XSS test from Project 36, with the stored row pasted in. If your sanitiser let something through, that is the finding, and it is worth more than a passing test.

Five. A front end with a list, an article page, metadata, a sitemap and an RSS feed, plus one paragraph on what you would cache and for how long if this publication had a hundred thousand readers a day.

COMMIT AS

```
git commit -m "Publication - Assignment One"
```

40

Money, dates, and the types that prevent bugs

There is a class of bug you cannot test your way out of, because the test and the code share the same wrong assumption. A total that is a cent off. A birthday that is a day early for half your users. A currency conversion that is a hundred times too large and looks plausible.

The defence is not care. It is a type that cannot hold the wrong thing. This chapter is the two the framework ships, and the general move behind both.

Why a float is the wrong type for money

```
fmt.Println(0.1 + 0.2)        // 0.30000000000000004
fmt.Println(1.1 * 3)          // 3.3000000000000003
```

This is not a Go problem. It is binary floating point, and JavaScript does the same thing, which is why you have seen `toFixed(2)` sprinkled through every invoice page you have ever worked on.

> **IF YOU KNOW JAVASCRIPT**
> `toFixed(2)` is a **display** fix applied to a **storage** problem. The error is already in the number; rounding it at the end hides one cent per operation and accumulates in the direction nobody notices until a month-end total is off by four pounds and somebody has to explain it.
>
> The fix is the same in both languages: store integers of the smallest unit, and keep the decimal point as a rendering concern.

Money: an integer and a currency

```
type Money struct {
    Amount   int64  `json:"amount"`
    Currency string `json:"currency" gorm:"size:3"`
}
```

`Amount` is in **minor units**: 2800 is twenty eight dollars. The currency rides with it, because an amount without one is not a quantity of anything.

A `money` field embeds this as two columns with the field name as prefix:

```
grit generate resource Invoice --fields "total:money"
```

gives `total_amount` and `total_currency`.

> **WATCH OUT**
> Two columns rather than one string, for a reason you feel at the first report: a column holding `"19.99 USD"` is not something you can `SUM`, index or compare in SQL.
>
> The amount stays an integer the database understands, and the currency is queryable, so "total revenue by currency this quarter" is one query rather than a loop in application code.

The operations, and which ones can fail

```
money.New(2800, "USD")          // from minor units
money.FromMajor(28.00, "USD")   // from a figure a person typed

m.Major()                       // 28.00, for display only
m.String()                      // "$28.00"

sum, err := a.Add(b)            // errors on mixed currencies
diff, err := a.Sub(b)
line := unit.MulInt(qty)        // a line total
tax := line.MulFloat(0.2)       // a percentage
parts := total.Allocate(3)      // a split that loses nothing
eur, err := usd.Convert("EUR", "0.9213")
```

Three of these are worth stopping on.

`FromMajor` is the one place a float is allowed near money, because a person typed `28.00` into a form. It is converted immediately rather than carried, and `Major()` goes the other way for display. Feeding `Major()` back into arithmetic is how the cent you were protecting gets lost.

`Add` returns an error, and the error is a currency mismatch.

> **WATCH OUT**
> You will be tempted to ignore that error, because your application has one currency. Do not.

The day it has two, an ignored error is not a crash and not a validation message. It is a total that is silently wrong, computed from amounts that were never comparable, and it will be wrong in reports that have already been sent.

This is the clearest case in the book for chapter four's rule: the error exists because the operation genuinely cannot be defined, and `_` is a decision to produce nonsense.

`Allocate` is the one people write by hand and get wrong. Splitting 10.00 three ways gives 3.34, 3.33, 3.33. Three lots of 3.33 loses a cent, and in accounting a lost cent is not a rounding detail, it is a reconciliation that fails. The remainder goes to the earliest parts, which is the convention accounting systems expect.

Not every currency has two decimal places

```
var exponents = map[string]int{
    "BIF": 0, "CLP": 0, "DJF": 0, "GNF": 0, "ISK": 0, "JPY": 0,
    "KMF": 0, "KRW": 0, "PYG": 0, "RWF": 0, "UGX": 0, "VND": 0,
    "XAF": 0, "XOF": 0, "XPF": 0,
    "BHD": 3, "IQD": 3, "JOD": 3, "KWD": 3, "LYD": 3, "OMR": 3,
    "TND": 3,
}
```

Yen and Ugandan shillings have no minor unit. Kuwaiti dinars have three. Every other currency is assumed to have two, which is true of most of ISO 4217.

WATCH OUT

A hardcoded `amount / 100` anywhere in your frontend is a bug waiting for its first international customer. Dividing a UGX amount by a hundred turns fifty thousand shillings into five hundred, on the invoice, in the email, in the report.

Format through the shared helper, which knows the exponent. This is a one-line habit that saves an embarrassing afternoon.

Conversion handles it too, which is the subtle part:

```
usd := money.New(1000, "USD")        // 10.00
jpy, _ := usd.Convert("JPY", "148.2") // 1482 JPY
```

Multiplying the amount by the rate and relabelling the currency would give 148200: a hundred times too much, silently, because USD has two decimals and JPY has none.

The rate is a **decimal string** rather than a float, and that is deliberate: 1.005 is 1.0049999999999989 in binary, and an exchange rate multiplies the largest amounts in the system, which is exactly where that error stops being invisible. The string is parsed as an exact fraction, multiplied once, and rounded once.

Dates: the type that exists because of one error message

A date field used to be a `*time.Time`, and `time.Time` accepts RFC 3339 and nothing else. The admin's date picker sends `"2001-08-06"`. So every date field failed to save with this:

```
parsing time "2001-08-06" as "2006-01-02T15:04:05Z07:00":
cannot parse "" as "T"
```

The fix was not to make the browser send a timezone it does not have. It was two wire types that accept what a browser actually sends:

```
type Date struct{ time.Time }      // "2001-08-06"
type DateTime struct{ time.Time }  // and the browser's variants
```

The accepted layouts, widest first:

```
time.RFC3339Nano
time.RFC3339
"2006-01-02T15:04:05"   // datetime-local, with seconds
"2006-01-02T15:04"      // datetime-local, without
"2006-01-02"            // a date picker
```

The middle two are what `<input type="datetime-local">` produces: no zone, and often no seconds. Neither is RFC 3339, and both are what you get from a real browser.

> **IF YOU KNOW JAVASCRIPT**
> The JavaScript habit that causes the worst version of this is building a `Date` from a date string and sending that.
>
> `new Date("2001-08-06")` is parsed as UTC midnight, so a user west of Greenwich sees the fifth of August. A date of birth that is a day early for everybody in the Americas is a bug that survives for years, because it is correct in the office where it was written.
>
> A date is not a timestamp. Send the string.

date against datetime, and how to choose

`date`	A day that is the same day everywhere: a birthday, an invoice date, a public holiday, a due date
`datetime`	A moment on the clock: a login, an appointment, a reading from a sensor

The test: **would two people in different countries disagree about what day it is?** A birthday is the sixth of August in Kampala and in Lima. A meeting at 14:00 is not the same moment in both.

Store timestamps in UTC and render them local. Store dates as dates, with no zone at all, because a zone on a birthday is a question nobody asked.

WATCH OUT
Date sorting failed silently once, and the way it failed is worth knowing because it will happen to you with some other column.

The sortable-column check tested the Go type for `string`, `int` or `uint`. A date is `*jsontime.Date`, so it matched none of them and was never added to the whitelist. The pagination layer **drops** a sort it does not recognise rather than rejecting it, so the request succeeded, the rows came back in whatever order the database felt like, and an invoice list could not be sorted by its issue date.

Nothing errored. Nothing logged. The general lesson: **a whitelist that silently ignores what it does not know needs a test that asserts the whitelist's contents**, not just a test that the endpoint returns 200.

The same move, in smaller places

Money and time are the big two. The field type list has several more of the same idea, where choosing the right one closes a bug rather than saving typing:

`email`, `url`, `tel`	Validated on write, so the column holds something sendable
`percent`	Stored as a number with a known meaning, not "0.2 or 20?"
`slug`	Lowercased, dashed, unique. A URL is a key
`select`	A closed set, in the database and in the type
`uint`	A count that cannot be negative, enforced by the type

IF YOU KNOW JAVASCRIPT
This is the thing functional programmers mean by "make illegal states unrepresentable", arrived at from the opposite direction: not from theory, but from reading support tickets.

Every type in that table exists because somebody shipped a string, and then shipped a validation, and then shipped a second validation in a second place that disagreed with the first.

PROJECT 37: AN INVOICE THAT ADDS UP
Build the arithmetic properly, then try to break it.

```
  grit new billing && cd billing
  grit generate resource Invoice \
    --fields "number:string, issued:date, due:date, \
client:belongs_to:Client, status:select:draft=Draft|sent=Sent|paid=Paid" \
    --items "InvoiceItem:description:string,qty:int,unit:money"
  grit generate resource Client --fields "name:string, email:email"
  grit migrate
```

One: line totals. Add a `Total()` method on the invoice that multiplies each line with `MulInt` and sums with `Add`, handling the error. Write a test with three lines whose total would be wrong in floats, for example three lines of 0.1 each.

Two: tax. Add twenty percent with `MulFloat`, rounded once. Assert the exact minor-unit value in a test rather than comparing formatted strings.

Three: the split. A payment plan over three instalments using `Allocate`. Assert that the instalments sum back to exactly the total. This is the test that catches a hand-rolled division.

Four: the mismatch. Add a line in a different currency and watch `Add` return an error. Then ignore the error on purpose, print the total, and look at the number you get. That number is what an ignored error buys you.

Five: the dates. Save an invoice through the admin with a due date, then read the raw row in `/studio`. Confirm it is a date and not a timestamp at midnight in somebody's zone. Then sort the invoice list by `issued` and confirm it actually sorts.

Six: the exponents. Create one invoice in JPY and one in KWD. Check both render correctly in the admin, in the list and in the detail view. If either is wrong by a factor of a hundred, you have found the hardcoded division.

YOUR TURN

In a scratch file, add `0.1 + 0.2` in Go, in JavaScript and in your database with a `NUMERIC` column, and print all three to twenty decimal places.

The third one is right, which is the argument for `NUMERIC` in the database. The first two are the reason the application should not be doing the arithmetic in a float on the way there.

ASSIGNMENT

In `part5-building/40-money`, build the invoicing slice and write the tests that prove the arithmetic.

One. The invoice with line items and a `Total()` that handles the currency error rather than discarding it.

Two. Four tests: a float-unsafe total, tax rounded once, an `Allocate` that sums back exactly, and a mixed-currency addition that fails.

Three. `MONEY.md` recording what 10.00 USD converts to in JPY at 148.2, and what you would have got by multiplying and relabelling. Show both numbers.

Four. A date field saved from the admin, with the stored row pasted in, and one paragraph on why it is a date and not a timestamp.

Five. Find one place in code you have already written where money is a float or a date is a timestamp. Describe the specific bug it would produce and which user would see it first. If you genuinely cannot find one, say so: that is a good sign and a rarer one than you would think.

COMMIT AS

```
git commit -m "Money and Time - Assignment One"
```

41

Project: an invoicing application

An invoice is not a record. It is a document somebody can be held to, and that changes what the software has to guarantee: the number must be unique and gap-free, the document must not change after it is sent, the arithmetic must be exact, and every change must leave a trace.

Chapter forty did the arithmetic. This chapter is the rest, and most of it is about refusing to do things.

What makes this harder than CRUD

Numbering	INV-202610-0003 must be unique, sequential and gap-free, under concurrent creates
Immutability	A sent invoice is evidence. You correct it with another document, not with an edit
State	Draft, sent, part paid, paid, void, and the transitions that are not allowed
Money	Totals, tax, partial payments, in a currency that is not always yours
Trace	Who changed what, when, kept in something nobody can quietly rewrite

Numbering, which is a concurrency problem in disguise

The naive version is `SELECT MAX(number) + 1`, and it is wrong the first time two people create an invoice in the same second. Both read the same maximum, both write the same number, and one of them wins depending on your unique index.

```
grit generate sequence Invoice --prefix INV \
  --reset monthly --width 4
```

That creates a counter package backed by a table, and a typed wrapper:

```
number, err := services.NextInvoiceNumber(db, time.Now())
// INV-202610-0001
```

The table is one row per sequence and period:

```
type Counter struct {
    Name      string `gorm:"primaryKey;size:50"`
    Bucket    string `gorm:"primaryKey;size:20"`
    NextValue uint64 `gorm:"not null;default:0"`
}
```

`Bucket` is the period key: `202610` for a monthly reset, `2026` for yearly, empty for never. The increment runs inside a transaction holding a row-level lock:

```
tx.Clauses(clause.Locking{Strength: "UPDATE"}).
    Where("name = ? AND bucket = ?", cfg.Name, bucket).
    First(&c)
```

so concurrent callers serialise on that row rather than racing.

WATCH OUT

Call the sequence from the model's `BeforeCreate`, and call `sequence.Next` **directly** rather than the generated `services.Next...` wrapper.

Services already import models. A model importing services is an import cycle, and Go refuses to compile it. The error names two packages and not the line you are thinking about, so it is worth knowing before you meet it:

```
import cycle not allowed
```

The wrapper exists for handlers and jobs. The hook, which lives in the model package, uses the generic function.

IF YOU KNOW JAVASCRIPT

The honest limit, stated in the package's own comment: one counter row serialises its writers, so beyond roughly a hundred invoices a second on a single counter you need a different design.

That number is far above what an invoicing application does and far below what an event stream does. Knowing which side of it you are on is the useful part. Most business documents are rare events; telemetry is not.

Immutability, and why an edit is the wrong verb

A draft is yours to change. A sent invoice is not, because a copy exists outside your database, in somebody's inbox and somebody else's accounts.

So the model has two regimes:

```
func (s *InvoiceService) Update(ctx context.Context,
    id uint, in UpdateInvoice) (*models.Invoice, error) {

    inv, err := s.GetByID(ctx, id)
    if err != nil {
        return nil, err
    }
    if inv.Status != "draft" {
        return nil, ErrNotEditable
    }
    ...
}
```

and a correction is a second document:

```
func (s *InvoiceService) CreditNote(ctx context.Context,
    id uint, reason string) (*models.CreditNote, error)
```

> **WATCH OUT**
> The pressure to allow "just a small edit" on a sent invoice will come from the client, and it will sound reasonable, because the mistake is small.
>
> It is still wrong, and the argument that works is not technical: the number on the document in their customer's hand no longer matches the number in the system, and the first person to notice is an auditor. Offer a credit note and a reissue, which is what accountants already expect.

The ledger of what actually happened goes in a resource nobody can rewrite:

```
grit generate resource LedgerEntry --append-only \
  --fields "invoice:belongs_to:Invoice, kind:string, \
amount:money, note:text"
```

`--append-only` means no update and no delete routes, a GORM guard, and a database trigger installed by `grit migrate`. Three layers, because the one that matters is the one that still holds when somebody writes a quick script against the database directly.

The state machine, written as methods

Statuses in a dropdown are a suggestion. Statuses as methods are a rule:

```
func (s *InvoiceService) Send(ctx context.Context, id uint) (
    *models.Invoice, error) {

    inv, err := s.GetByID(ctx, id)
    if err != nil {
```

```
        return nil, err
    }
    switch inv.Status {
    case "sent", "paid":
        return inv, nil          // idempotent
    case "void":
        return nil, ErrVoided
    }
    if len(inv.Items) == 0 {
        return nil, ErrNoLines
    }

    now := time.Now().UTC()
    inv.Status, inv.SentAt = "sent", &now

    if err := s.db(ctx).Save(inv).Error; err != nil {
        return nil, fmt.Errorf("sending %d: %w", id, err)
    }
    events.Emit(ctx, events.InvoiceSent{ID: inv.ID})
    return inv, nil
}
```

The transitions worth refusing, in any document workflow:

Send an empty invoice	A document with no lines is a mistake, not a zero bill
Void a paid invoice	Money moved. Refund it, do not pretend
Pay a draft	The customer has not been asked yet
Edit after sending	Covered above

Payments, and the status you should not store twice

A payment is its own record, because partial payments exist and because a payment has a date and a method that the invoice does not:

```
grit generate resource Payment \
  --fields "invoice:belongs_to:Invoice, amount:money, \
paid_on:date, method:select:bank=Bank|card=Card|cash=Cash"
```

The balance is arithmetic over payments, not a column somebody updates:

```
func (inv *Invoice) Balance() (money.Money, error) {
    total, err := inv.Total()
    if err != nil {
        return money.Money{}, err
    }
    for _, p := range inv.Payments {
```

```
        total, err = total.Sub(p.Amount)
        if err != nil {
            return money.Money{}, err
        }
    }
    return total, nil
}
```

> **WATCH OUT**
> Storing `paid_amount` on the invoice as well as storing payments is two sources of truth, and they will disagree. They always disagree after the first deleted payment, the first failed transaction, or the first manual correction in the database.
>
> Derive it. If the derivation is too slow at scale, that is a **caching** decision made later with measurements, and a cached value is clearly labelled as one. It is not the same as two columns that both claim to be the truth.

The document itself

The admin gives you two ways out of the screen:

Print and PDF. The detail page has a print action, and PDF rendering uses `@react-pdf/renderer` in the browser, lazy loaded so it is not in the first load of every list page.

Excel. The list exports with the current filters applied, through `xlsx`, also lazy loaded.

> **IF YOU KNOW JAVASCRIPT**
> Client-side PDF is right until it is not. It is free, it uses the layout you can see, and it needs no fonts installed on a server.
>
> It stops being right when the PDF must exist without a browser: attached to an automated email, archived nightly, generated by a webhook. At that point the job moves server side and becomes a real project with fonts, page breaks and a rendering budget.
>
> Start with the browser. Move it when a background job needs the document, not before.

Reminders, in one line for now

Overdue invoices should chase themselves:

```
grit generate job SendInvoiceReminders --cron "0 9 * * *"
```

That writes a job that runs at nine every morning. Chapter forty three is about what makes a job safe to run unattended: retries, idempotency, and what happens when it

fails at three in the morning. For now, note the shape: the job calls the same service method a human would, so the rules cannot be bypassed by automation.

PROJECT 38: BUILD THE INVOICING APPLICATION

```
  grit new billing && cd billing
  grit generate resource Client \
    --fields "name:string, email:email, address:text"
  grit generate resource Invoice \
    --fields "number:string, client:belongs_to:Client, \
issued:date, due:date, \
status:select:draft=Draft|sent=Sent|paid=Paid|void=Void" \
    --items "InvoiceItem:description:string,qty:int,unit:money"
  grit generate sequence Invoice --prefix INV --reset monthly
  grit generate resource Payment \
    --fields "invoice:belongs_to:Invoice, amount:money, paid_on:date"
  grit generate resource LedgerEntry --append-only \
    --fields "invoice:belongs_to:Invoice, kind:string, amount:money"
  grit migrate
```

One: numbering under pressure. Wire the sequence into `BeforeCreate`. Then create fifty invoices concurrently:

```
var wg sync.WaitGroup
for i := 0; i < 50; i++ {
    wg.Add(1)
    go func() { defer wg.Done(); createInvoice(t) }()
}
wg.Wait()
```

Assert fifty distinct numbers with no gaps. Run it with `-race`.

Two: the state machine. `Send`, `Pay`, `Void` and `CreditNote` as service methods, each refusing the transitions in the table above, each idempotent where that makes sense. Tests for every refusal, not just the happy paths.

Three: immutability. Prove that updating a sent invoice fails through the API and through the admin. Then try it in `/studio` and notice that you can, which is why the ledger exists.

Four: the balance. Partial payments, with the balance derived. Add a payment in a different currency and confirm the error surfaces as a 422 rather than a wrong number.

Five: the document. Print an invoice to PDF from the detail page and read it as a customer would. Then export the list to Excel with a filter applied and confirm the filter was respected.

Six: the trace. Write a ledger entry on every transition. Then try to delete one through the API, through the admin, and directly with SQL. Record which of the three refused you, and why that is the right number.

YOUR TURN

Create an invoice in October and one in November and look at the numbers. The monthly reset means the counter starts again, which is what most accounting systems expect and which surprises people who assumed numbers are global.

Then decide, for your own domain, whether a reset is right at all. Gap-free within a period and restarting is one convention; a single unbroken sequence forever is another. Both are defensible, and picking one after the first thousand documents is not.

ASSIGNMENT

In `part5-building/41-invoicing`, build the application with its guarantees demonstrated rather than claimed.

One. The resources, the sequence in `BeforeCreate`, and the concurrency test proving distinct gap-free numbers under fifty parallel creates.

Two. The four state transitions as service methods, with a test for every refusal in the table.

Three. `GUARANTEES.md` with one short section each: how numbering survives concurrency, what happens to a sent invoice somebody wants to change, how the balance is derived, and what the append-only ledger does and does not protect you from.

Four. One deliberate failure. Remove the row lock from the sequence, run the concurrency test again, and paste the duplicate numbers into the notes. Then put it back.

Five. A paragraph for a non-technical reader, the one you would send a client, explaining why you cannot simply edit a sent invoice. If you can write that paragraph, you understand the chapter.

COMMIT AS

```
git commit -m "Invoicing - Assignment One"
```

42

File uploads and the image pipeline

D9 was “a user uploads a ten megabyte photo from their phone, and it is served to every visitor at ten megabytes”. It is the most predictable performance problem in web software, it is never the thing anybody plans for, and it has a second half nobody mentions: that photo arrives with the GPS coordinates of the house it was taken in.

This chapter is what a file field does, what happens to an image between the form and the disk, and the two attacks that live in this feature.

A file field, and what it stores

```
grit generate resource Product \
  --fields "name:string, photo:file, gallery:files"
```

`file` is one, `files` is many. The column does not hold a path. It holds a structured reference:

```
{
  "url": "https://cdn.example.com/u/2026/10/x7f3.webp",
  "key": "u/2026/10/x7f3.webp",
  "name": "kitchen-final.jpg",
  "mime": "image/webp",
  "size": 147204,
  "width": 1600,
  "height": 1067,
  "thumbnail_url": ".../x7f3_thumb.webp",
  "format": "webp",
  "optimised": true
}
```

IF YOU KNOW JAVASCRIPT

A bare URL string is what everybody stores first, and it is missing exactly the fields a page needs.

Without `width` and `height` the browser cannot reserve space, so the layout jumps as images load, which is most of a bad Cumulative Layout Shift score. Without `name` you cannot show the user what they uploaded. Without `mime` you are guessing from the

extension. Without `thumbnail_url` a list page downloads full-size images to draw at 60 pixels.

`optimised: false` is the one people miss: it says the pipeline failed and the original was stored untouched, which is a thing you want to be able to find later.

Where the bytes go

S3, R2, Spaces	Production. Any S3-compatible object store
MinIO	The same API, in Docker, for local development
Local disk	`STORAGE_DRIVER=local`. No object store at all

The local driver is the one to use while learning. It needs nothing running, and uploads land in a directory the API serves.

WATCH OUT

With `STORAGE_DRIVER=local`, every stored file's URL is built from `APP_URL`. Change `APP_PORT` without changing `APP_URL` and every upload gets a URL pointing at a port nothing is listening on. The upload succeeds, the file is on disk, and the image is broken, which sends people hunting through the storage code for a bug that is in an environment variable.

On a phone or another machine, the same applies to `localhost`: an image URL that says `localhost` resolves to the phone itself. Mobile clients rewrite the host for this reason, and MinIO has to be published on the LAN address rather than `127.0.0.1`.

What happens to an image on the way in

Every image field runs through a profile. The default one is a set of decisions somebody would otherwise have to research:

```
func DefaultProfile() Profile {
    return Profile{
        Max:        Fit(1600, 1600),
        Quality:    0.82,
        Format:     Auto,
        Renditions: map[string]Size{"thumb": Fill(400, 400)},
        OnError:    StoreOriginal,
        MaxPixels:  50_000_000,
    }
}
```

1600	Covers a 2x retina display at a typical content width. Storing more is paying to keep pixels no browser will draw
0.82	The point at which JPEG is visually indistinguishable from source. Below about 0.75, gradients and skin show artifacts
400	The thumbnail an admin row or a card grid needs at 2x
Auto	Encode to the best format the client can take, usually WebP

Measured on a 5.3 MB phone photo: **147 KB out, including the thumbnail**. That is D9 closed, and it happened without anybody remembering to do it.

Three more things happen in the same pass, and two of them are not about size.

EXIF orientation is applied. A portrait photo from a phone is usually stored landscape with a rotation flag, and a naive decode produces a sideways thumbnail.

Metadata is stripped. Including GPS coordinates. A photograph taken at home and uploaded to a public listing carries the house's location in a field nobody looks at, and stripping it is not a feature request anybody files: it is something a framework should do by default.

Renditions are derived. The thumbnail comes from the decoded original rather than from the resized copy, so it is not a resize of a resize.

Your own profile, when the default is wrong

```
media.Define("product-image", media.Profile{
    Max:        media.Fit(2400, 2400),
    Quality:    0.9,
    Renditions: map[string]media.Size{
        "thumb": media.Fill(400, 400),
        "card":  media.Fit(800, 0),
    },
})
```

A clothing retailer wants more pixels and higher quality than a blog does. A document scan wants no lossy compression at all. The profile is the one place that decision lives, rather than being spread across the upload handler, the list page and whatever the mobile app does.

The chainable API is there when you need something bespoke:

```
img, err := media.Open(file)
thumb, err := img.Cover(150, 150).ToWebP().Encode()
large, err := img.Fit(1600, 0).Quality(80).ToJPEG().Encode()
```

An operation that fails does not panic and does not return its own error. The error travels down the chain and comes out of `Encode` or `Store`, which is what makes the chain readable.

The decompression bomb, which the size limit cannot see

```
MaxPixels: 50_000_000
```

This is the line that stops a specific attack, and it is worth understanding because the obvious defence does not work.

A solid-colour PNG compresses to almost nothing whatever its dimensions. **165 kilobytes on the wire can be 144 megapixels decoded.** Measured, one such upload allocated 224 MB, and ten concurrent uploads would have been 2.2 GB, which is the process gone.

Your file size limit cannot see it, because on disk the file is small. The header is read first, the pixel count is computed before any pixels are allocated, and an image claiming more than the profile allows is refused without being decoded.

> **WATCH OUT**
> Fifty megapixels is chosen to pass a 48 megapixel professional camera frame and refuse the bomb. If you raise it, raise it with a number in mind rather than to make one complaint go away.
>
> The same reasoning applies to the upload cap of 50 MB. Limits that exist for a reason should carry the reason in a comment, or the next person removes them during a demo.

When the pipeline fails

```
OnError: StoreOriginal
```

A transform that fails stores the original untouched and marks the ref `optimised: false`. The alternative, `Reject`, refuses the upload.

Storing a large image is a worse outcome than storing a small one, and both are much better than losing the user's file because your encoder did not like something. The failures are findable afterwards, and a reprocess pass can retry them.

> **WATCH OUT**
> `DiscardOriginal` is deliberately phrased in the negative, and the story is a general lesson about defaults in Go.

It began as `KeepOriginal`, which read better. But profiles are merged with the defaults field by field, and a `bool` has no way to say "unset": every profile that did not mention `KeepOriginal` had it as `false`, so the original was silently discarded while the documentation promised the opposite.

The rule: **in a struct that gets merged with defaults, the zero value must be the recommended behaviour**. Where it cannot be, use a pointer and treat `nil` as unset.

Uploading from the frontend

The admin has a dropzone component, and the generated form uses it for any `file` or `files` field. Behind it is an ordinary multipart POST:

```
const body = new FormData()
body.append("file", file)

const res = await fetch(`${API}/api/v1/uploads`, {
  method: "POST",
  body,                          // no Content-Type header
  credentials: "include",
})
```

WATCH OUT

Do not set `Content-Type` yourself on a multipart request. The browser adds it **with the boundary parameter**, and a hand-written `multipart/form-data` header has no boundary, so the server cannot parse the body.

This is the single most common upload bug in every stack, and on React Native it has a sibling: some upload helpers send an empty body, and the fix is the same `fetch` plus `FormData` with no content type.

For very large files, the direct-to-storage path skips the API: the client asks for a presigned URL, uploads straight to the bucket, and tells the API the key. The image helpers run the same transform afterwards, so a presigned upload is not a way around the pipeline.

Serving them

Three things make the difference between a fast page and a slow one, and none of them is the upload:

Use the thumbnail in lists. A table of fifty rows showing full-size images is fifty megabytes for fifty thumbnails.

Give the browser the dimensions. They are in the ref, so pass them to the image component and the layout stops jumping.

Let the CDN cache it. Objects are content-addressed by key, so a long cache lifetime is safe: a changed image is a new key.

PROJECT 39: MEASURE THE PIPELINE
Numbers, not impressions.

```
grit new gallery && cd gallery
grit generate resource Photo \
  --fields "caption:string, image:file, album:belongs_to:Album"
grit generate resource Album --fields "name:string"
grit migrate && grit start
```

One: the ratio. Upload a real photo straight off a phone. Record the file size before, the stored size after, and the thumbnail size. Compute the ratio and put all three in `NOTES.md`.

Two: the metadata. Before uploading, read the EXIF of your photo:

```
exiftool kitchen.jpg | grep -i "gps\|orientation"
```

Then read the stored file. The GPS fields should be gone and the image should be the right way up.

Three: the bomb. Generate a large solid-colour PNG and try to upload it:

```
  python -c "from PIL import Image; \
Image.new('RGB',(12000,12000),'blue').save('bomb.png')"
```

Record the file size on disk and what the API says. Then raise `MaxPixels`, retry, and watch the memory of the API process while it decodes.

Four: a profile of your own. Define a profile for `Photo.image` with a second rendition at 800 pixels wide, re-upload, and confirm both renditions exist in storage.

Five: the failure path. Upload a file with a `.jpg` extension that is not an image. Confirm the ref says `optimised: false` and that the row still exists, rather than the request failing with a 500.

Six: the page. Render the album with thumbnails and dimensions set, then again without, and compare the layout shift in Lighthouse.

YOUR TURN
Switch `STORAGE_DRIVER` to `local`, upload a file, and find it on disk. Then change `APP_PORT` without changing `APP_URL` and reload the page.

Seeing the broken image once, with the cause known, is worth an hour of debugging it later with the cause unknown.

ASSIGNMENT

In `part5-building/42-uploads`, build the gallery and report the pipeline honestly.

One. The resources, the upload working through the admin and through a `fetch` you wrote yourself.

Two. `PIPELINE.md` with a table of five real photos: original size, stored size, thumbnail size, dimensions in and out, and the format chosen.

Three. The EXIF evidence, before and after, with the GPS fields shown present and then absent.

Four. The decompression bomb, with the file size on disk next to the decoded megapixels, and what the API did. One sentence on why a file size limit alone would not have saved you.

Five. One custom profile with a justification: name the screen that needs the extra pixels, and the trade you are making in storage to get them.

COMMIT AS

```
git commit -m "Uploads - Assignment One"
```

43

Background jobs, cron and transactional email

Some work must not happen inside a request. Resizing twelve images, emailing four hundred people, reconciling yesterday's payments: if the user is waiting for it, you have built a timeout.

This chapter is the queue, the scheduler and the mailer, and most of it is about the two questions that separate a job system from a `go func()`: what happens when it fails, and what happens when it runs twice.

Why not just a goroutine

```
go sendTheEmails(users)   // please do not
```

Three things are wrong with it, and the third is the one that hurts.

It dies with the process. A deploy, a crash, a scale-down, and the work is gone with no record that it existed.

It cannot be retried. The error goes to a log, if anywhere.

It cannot be seen. Nobody can answer "did the invoice reminders run last night", which is the question you will actually be asked.

> **IF YOU KNOW JAVASCRIPT**
> This is the same reason a Node application reaches for BullMQ rather than `setTimeout`. The difference in Go is that the goroutine is so cheap and so easy that the temptation is stronger, and it works in development every single time.
>
> The framework does keep a small escape hatch for genuinely minor work: a detached task with its own context and a hard cap of thirty two concurrent slots, deliberately small, because **anything needing more than thirty two workers is a job**.

A job, generated

```
grit generate job SendInvoiceReminders --cron "0 9 * * *"
```

That writes three things: the job file, a handler registration in the worker's mux, and, with `--cron`, a schedule entry that the admin's cron page lists.

The job file is yours from the moment it is written, and a second `generate` refuses rather than overwrites.

```
const TypeSendInvoiceReminders = "send_invoice_reminders"

type SendInvoiceRemindersPayload struct {
    Day string `json:"day,omitempty"`
}

func handleSendInvoiceReminders(deps WorkerDeps) func(
    context.Context, *asynq.Task) error {

    return func(ctx context.Context, task *asynq.Task) error {
        var p SendInvoiceRemindersPayload
        if len(task.Payload()) > 0 {
            if err := json.Unmarshal(task.Payload(), &p); err != nil {
                return fmt.Errorf("decoding: %v: %w",
                    err, asynq.SkipRetry)
            }
        }
        // deps.DB, deps.Mailer, deps.Storage, deps.Cache
        return nil
    }
}
```

Two conventions in that template are the whole discipline.

The payload is IDs and small values. It is serialised into Redis, and a retry sees exactly what the first attempt saw. Putting a whole struct in the payload means a retry works from a snapshot that may now be stale.

An undecodable payload is wrapped with `asynq.SkipRetry`. A payload that does not parse now will not parse in four minutes either. Retrying a permanent failure is how a queue fills up with work that can never succeed.

Return an error to retry, return nil to stop

```
return nil   // done, or can never succeed
return err   // retry with backoff
```

That is the entire protocol, and the second half of the first line is the part people get wrong. A task that fails for a reason that will never change, a deleted record, a malformed input, a file that is not an image, should return `nil` and log loudly, or wrap with `SkipRetry`. Returning the error buys you the same failure five more times with exponential backoff.

> **WATCH OUT**
> The practical rule: **retry anything that is about the world, do not retry anything that is about the data**.
>
> A network timeout, a rate limit, a database that is briefly down: retry. Invalid JSON, a missing user, an unsupported format: do not. If you cannot tell which one you have, you can log it and return `nil`, and the work stays visible in the logs rather than invisible in a retry loop.

The two kinds of "it ran twice"

A queue guarantees **at least once**, not exactly once. The retry that happens after a worker was killed mid-task is the normal case, not the pathological one.

So every job has to answer: what if this runs again?

Idempotency at the enqueue. The same work queued twice is refused:

```
queue.EnqueueSendInvoiceReminders(ctx, payload,
    jobs.EnqueueOption{
        IdempotencyKey: "reminders:" + day,
        Window:         24 * time.Hour,
    })
```

A second enqueue inside the window returns `ErrDuplicateTask` rather than running. This is how a nightly job that is triggered twice, by a scheduler and by an impatient human, stays a single run.

Idempotency in the work. Even one enqueue can execute twice. So the work itself should be safe to repeat: mark the invoice as reminded in the same transaction that records the send, and skip the ones already marked.

> **WATCH OUT**
> The order matters, and the wrong order is the one that feels natural.
>
> Send the email, then mark it sent: a crash between the two sends it again tomorrow. Mark it sent, then send: a crash between them means nobody is ever reminded.
>
> Neither is free, so choose the failure you prefer. For a reminder, a duplicate is better than a silence. For a payout, a silence is very much better than a duplicate, and the answer is usually a record written first with a status that a later pass reconciles.

Cron, and the five fields

```
grit generate job NightlyReconcile --cron "30 23 * * *"
grit generate job Heartbeat --cron "@every 15m"
grit generate job Cleanup --cron "@daily"
```

The spec is validated when you generate, not at three in the morning:

```
--cron wants five fields (minute hour day-of-month month
day-of-week), e.g. "30 23 * * *" for 23:30 every day
```

Scheduled runs arrive with an empty payload, which is why the template handles that case. The schedule is registered with a retry limit, and every registered task appears on the admin's cron page with its name, its spec and its type, so "did it run last night" is a screen rather than a grep.

> **WATCH OUT**
> Cron runs in the **server's** timezone. A daily report at `0 9 * * *` lands at nine in UTC, which is not nine for anybody you are reporting to.
>
> Set the timezone deliberately and write it down. And for anything that must happen on a date rather than at an hour, a daily job that checks for the date beats a schedule that tries to express it.

Mail, and the drivers

```
MAIL_MAILER = smtp | resend | mailgun | postmark |
              sendgrid | ses | log | failover
```

Three behaviours in the configuration are worth copying into whatever you build next.

An unset driver does something sensible. In development the mail goes to Mailhog from Docker Compose, falling back to the log when Mailhog is not running, so a password reset link is never lost.

A named but misconfigured driver is an error, not a fallback. An application that believes it sends through SES must not quietly write to a log instead.

Production with no mailer configured returns nothing, and the code that sends says so, rather than pretending.

> **YOUR TURN**
> Set `MAIL_MAILER=log`, trigger a password reset, and read the link out of your terminal.
>
> It is the fastest way to work on an auth flow, and it is the setting to reach for whenever an email is in the way of what you are actually testing.

Queued or inline, decided once

Transactional mail goes through one function, and the decision is made there:

```
func dispatchMail(ctx context.Context, mailer *mail.Mailer,
    queue *jobs.Client, key string,
    opts mail.SendOptions) error {

    if queue != nil {
        return queue.EnqueueSendEmail(ctx, opts.To,
            opts.Subject, opts.Template, opts.Data,
            jobs.EnqueueOption{IdempotencyKey: key})
    }
    if mailer == nil {
        return ErrNoMailer
    }
    sendCtx, cancel := context.WithTimeout(
        context.WithoutCancel(ctx), 15*time.Second)
    defer cancel()
    return mailer.Send(sendCtx, opts)
}
```

With Redis, the mail is queued and retried. Without it, the mail is sent inline with a timeout. The caller does not care, which is why a project can start without Redis and gain it later without touching the sending code.

`context.WithoutCancel` is the detail worth stealing. The send is allowed to finish after the request it came from has been answered, but not to run forever: the request's context is cancelled the moment the response is written, and inheriting that cancellation would kill the send mid-flight.

> **WATCH OUT**
> The `key` is the idempotency key, and the thing to pass is something stable for the **business action**, such as `"verify:" + user.ID`.
>
> A client retrying a request, or a proxy replaying it, then does not send the same person the same verification email twice. A key built from a timestamp or a random value deduplicates nothing at all, which is the common mistake.

Seeing it work

Three places, and knowing they exist saves an afternoon:

Admin: Jobs	Queued, active, retrying and dead tasks, with payloads
Admin: Cron	Every registered schedule and when it last ran
Admin: Mail	A preview of each template, rendered, without sending

The mail preview is the one that changes how you work. Email templates are miserable to iterate on through actual sends, and a page that renders them with sample data turns a twenty minute loop into a reload.

PROJECT 40: MAKE THE REMINDERS TRUSTWORTHY
Using the invoicing application from chapter forty one.

```
grit generate job SendInvoiceReminders --cron "0 9 * * *"
```

One: the work. Find invoices that are sent, unpaid and past due, and email each client once. Use the mail dispatch function rather than the mailer directly.

Two: idempotency, twice over. An `IdempotencyKey` of `"reminders:" + day` on the enqueue, and a `reminded_at` column checked and written in the same transaction as the send record. Then run the job three times in a row and prove one email.

Three: the failure. Point the mailer at a broken SMTP host. Watch the retry with backoff in the admin's Jobs page, and find the task in the dead queue when it gives up. Then fix the host and replay it from the admin.

Four: a permanent failure. Feed the job an invoice whose client has no email. It must not retry five times. Decide between `SkipRetry` and returning `nil` with a log, and write one sentence on why.

Five: the clock. Change the cron spec to run in two minutes, deploy nothing, restart the worker, and confirm it fires. Note the timezone the server is in and what that means for `0 9 * * *`.

Six: the preview. Open the mail preview page, find your reminder template, and fix the thing that is wrong with it. There is always something wrong with it, usually the subject line when the client name is empty.

YOUR TURN
Kill the worker process while a job is running, then start it again.

Watch what happens to the task. That is the "at least once" guarantee in front of you, and it is the single best argument for the idempotency work this chapter asks for.

ASSIGNMENT
In `part5-building/43-jobs` , build the reminder job and prove its behaviour under failure.

One. The job, scheduled, with both layers of idempotency, and a test that runs the handler twice and asserts one email.

Two. `JOBS.md` with four short sections: what this job retries, what it refuses to retry, what happens if it runs twice, and what happens if it never runs at all.

Three. Evidence from the admin: a screenshot or a paste of the Jobs page showing a retry, and the dead queue entry after it exhausted its attempts.

Four. The ordering question, answered for your own job. Which do you write first, the send or the mark, and which failure did you choose to live with?

Five. One job of your own that is not mail: a nightly export, a cleanup of expired rows, a reconciliation. Schedule it, run it, and say in one line how you would know tomorrow morning whether it ran.

COMMIT AS

```
git commit -m "Jobs and Mail - Assignment One"
```

44

The event bus, realtime, and webhooks

An application grows a particular kind of mess: one thing happens, and five systems have to hear about it. An order is placed, so write the audit record, push to the open dashboard, send the webhook, queue the receipt, update the cache. Every one of those is a line in the handler, and the sixth is the one somebody forgets.

This chapter is the alternative: the handler says what happened, once, and anything interested listens.

The problem, in the framework's own history

Before the bus existed, a generated handler called the audit logger and nothing else. Webhooks, realtime and background jobs all existed, and **none of them was ever fired by generated code**. Dispatching a webhook meant hand-writing the call in every handler, for every operation, on every resource.

Forgetting one produced no error at all. Just a subscriber that never heard anything, which is the worst failure mode there is: silent, and only noticed by the customer whose integration stopped working.

```
events.Emit(ctx, events.Created{
    Resource: "order", ID: order.ID, Actor: actor,
})
```

One line in the handler. The audit log, outbound webhooks and realtime clients are now subscribers rather than things somebody has to remember to call.

Three deliveries, and choosing between them

```
const (
    Sync Delivery = iota
    Async
    Durable
)
```

`Sync`	Runs inline, before the handler responds. For work that must be part of the request

`Async`	Runs on the bus's worker pool after the response. For anything with a network call
`Durable`	Through the transactional outbox: at least once, retried, surviving restarts

Sync is for the audit record: it needs the request context and should be written before the caller is told the write succeeded. A slow `Sync` subscriber is a slow endpoint, and that is the whole cost model.

Async is for the webhook endpoint that takes four seconds. The API must not take four seconds because somebody else's server is slow.

Durable is for work another module **depends on**. If losing the event would leave two systems disagreeing, it is durable.

> **WATCH OUT**
> The async queue is bounded, and that is deliberate. An unbounded queue turns one slow subscriber into memory exhaustion, which fails later and worse than dropping events and saying so.
>
> When you see events being dropped in the logs, the fix is not a bigger queue. It is finding the subscriber that is slow, and usually moving it to `Durable` where retrying is the design rather than an accident.

The outbox, and the choice it removes

`Durable` delivery exists because of a problem with two halves, and doing the obvious thing gets one of them wrong whichever order you pick.

Publish, then commit. The publish succeeds, the commit fails, and a webhook has announced an order that does not exist. Downstream systems act on it, and there is nothing to reconcile against because the row was never written.

Commit, then publish. The commit succeeds, the process is killed before the publish, and the order exists with nobody told. Nothing is logged anywhere, because from the process's point of view nothing failed.

The outbox removes the choice:

```
BEGIN
  INSERT INTO orders ...
  INSERT INTO outbox_messages ...
COMMIT
```

The message is written in the **same transaction** as the business data, so it commits or rolls back with it. A relay reads committed messages and delivers them afterwards. Either both happened or neither did.

> **IF YOU KNOW JAVASCRIPT**
> This pattern is worth carrying with you whatever you build next, because it is not a framework feature. It is the standard answer to "I need to update my database and tell somebody else, atomically", and it works in any language against any queue.
>
> The name to look up is **transactional outbox**. Once you have seen it, you start noticing the places you got away without it.

The cost is honest and must be stated: delivery is **at least once**, not exactly once. A message can be delivered twice if the process dies between the send and the status update. Consumers have to be idempotent, which is what a dedup key on the receiving end is for, and which is the same discipline chapter forty three asked for.

Realtime: channels, not just users

The hub can address a user, everybody, or a named channel. Channels are the one that makes a real interface possible:

```
socket.send(JSON.stringify({
  action: "subscribe",
  channel: "invoice.42",
}))
```

The server answers `subscribed`, `subscription_error` or `unsubscribed`, each naming the channel, and server code publishes to the name:

```
hub.Publish("invoice.42", payload)
```

Every subscriber on every replica receives it, which is the part that is tedious to build: a second instance of your API has to reach clients connected to the first, so the hub has a backplane.

> **WATCH OUT**
> A channel name is a permission question, and it must be answered server side. `invoice.42` means "whoever may read invoice 42", not "whoever can type that string".
>
> The hub has a registry of authorizers per channel pattern for this reason. A realtime layer where the client picks what it hears is an API with no access control, reachable from a browser console.

Without channels, a page showing one invoice had no way to hear about that invoice without every event for that user being pushed at it, which is a real cost on a busy account.

When WebSockets do not arrive

Some platforms do not pass a WebSocket upgrade through to the application. One managed host strips `Connection: upgrade` before the container, and it is not alone: a corporate proxy, an older load balancer, or an API gateway configured for plain HTTP all do the same.

The failure is unpleasant: the handshake fails, the client's reconnect loop retries forever, and live updates silently never arrive. Nothing errors in a way anybody sees.

So there is a Server-Sent Events fallback. SSE is plain HTTP with a response that never ends, so it survives all of that.

WebSocket	Two-way, lower latency, needs the upgrade to survive the network
SSE	Server to client only, plain HTTP, reconnects by itself

For dashboards, notifications and live lists, which is most of what an application needs, SSE is enough. Reach for WebSockets when the client needs to **send** continuously: a cursor position, a drawing, a chat typing indicator.

Webhooks, outbound

```
grit plugin add webhooks
```

That wires outbound, Standard-Webhooks-signed deliveries with retries, a delivery log and manual replay. Core already **verifies** incoming webhooks; this **sends** outgoing ones, which is the opposite direction and a different problem.

What matters when you are the one sending:

Sign every delivery. The receiver has to know it was you. Standard Webhooks is a published convention for the headers and the signature, which beats inventing your own and documenting it badly.

Include an ID and a timestamp. Together they let a receiver reject replays and deduplicate retries, which they will need because you are delivering at least once.

Retry with backoff, and keep a log. Somebody will ask "did you send it", and the answer has to be a record with a status code, not a shrug.

Make replay a button. Integrations break on the other side. A delivery log with a replay action turns a support conversation into one click.

> **WATCH OUT**
> A webhook subscriber must be `Async` or `Durable`, never `Sync`. Your API's response time should not depend on a receiver you do not control, and some receivers take seconds.

The same goes for the timeout on the delivery itself. An HTTP client with no timeout, pointed at a stranger's server, is a slow leak of connections that ends in an API that will not answer anybody.

PROJECT 41: WIRE ONE EVENT TO FOUR LISTENERS

Using the invoicing application again, because an invoice being paid is a genuinely interesting event.

One: emit. In `MarkPaid`, emit one event with the invoice ID, the amount and the actor. Nothing else changes in that method.

Two: four subscribers.

```
audit     Sync      writes the ledger entry
realtime  Async     publishes to "invoice.<id>"
mail      Async     queues the receipt
webhook   Durable   notifies the accounting integration
```

Each in its own file. None of them is called by the handler.

Three: watch the ordering. Add a deliberate two second sleep to the realtime subscriber and time the endpoint. Then move it to `Sync` and time it again. That difference is the whole reason the delivery modes exist.

Four: the outbox. Make the webhook durable, then kill the API process immediately after a payment, before the delivery. Restart and confirm the webhook still goes out. Then look in `outbox_messages` and read the row.

Five: the rollback. Force the transaction to fail after the outbox write. Confirm there is no message and no delivery: that is the half of the problem publish-then-commit gets wrong.

Six: the browser. Open an invoice page, subscribe to its channel, and mark it paid from a second window. The page should update with no polling. Then try to subscribe to a channel for an invoice your user may not read, and confirm the server refuses.

YOUR TURN

Count the lines in your `MarkPaid` method before and after this chapter.

It should be the same length or shorter, while doing four more things. That is the measure of whether an event bus is paying for itself: the handler describes what happened, and the list of interested parties is a configuration rather than a sequence of calls nobody can safely reorder.

ASSIGNMENT

In `part5-building/44-events`, wire one domain event to several listeners and prove the delivery guarantees.

One. One `Emit`, four subscribers, one per file, with the delivery mode of each chosen and justified in a comment.

Two. Timings: the endpoint with the slow subscriber on `Sync` and on `Async`, both recorded in `EVENTS.md`.

Three. The crash test. Kill the process between commit and delivery, show the outbox row, restart, and show the delivery completing. Paste both states of the row.

Four. The rollback test. A failed transaction leaves no message and no webhook.

Five. A channel authorizer that refuses a subscription the user may not have, with the client's error response pasted in. One paragraph on what an unauthorised channel would expose in your own application.

COMMIT AS

```
git commit -m "Events - Assignment One"
```

45

Observability: Studio, Pulse and Sentinel

B11 was "when something goes wrong, nobody can tell me what happened". The honest version of that problem is that the information existed, in a log file, on a server, in a format nobody could search, during an hour when the person who could read it was asleep.

A generated project mounts three dashboards and a health endpoint. This chapter is what each one answers, and the discipline that makes them useful rather than decorative.

Four questions, four tools

`/studio`	What is actually in the database?
`/pulse`	What is this request doing, and why is it slow?
`/sentinel`	Who is attacking us, and what did we do about it?
`/health`	Is everything this process depends on alive?

They answer different questions, and reaching for the wrong one is how a debugging session takes three hours. "The invoice shows the wrong total" is Studio. "The invoice page takes four seconds" is Pulse.

Studio: the database, as it is

The embedded database browser at `/studio` lists tables, browses rows, follows foreign keys and runs queries. It is the fastest way to answer a question of the form "did that actually save".

The habit worth forming: **before debugging the code, look at the row**. Half of all "the API is broken" reports are a row that is exactly what the code wrote, and the bug is upstream in what was sent.

> **WATCH OUT**
> Studio sees everything, which is the point and also the risk. The scoping that makes a user's query return only their rows is in the service layer, and Studio is underneath it.

So a row you can see in Studio and cannot fetch through the API is usually ownership scoping doing its job rather than a bug. Chapter thirty five's 404 is the same observation from the other side.

Pulse: where the time went

Pulse mounts at `/pulse` and records request traces, database queries, runtime metrics and errors. It is the tool for the question "why is this slow", and it answers it in the only way that helps: by showing which part took the time.

The three views that earn their keep:

The request trace. One request, with its queries listed and timed. An endpoint taking 800ms turns out to be forty queries of 20ms, which is the N+1 from chapter twenty two, now visible rather than suspected.

The slow query list. Sorted by total time rather than by worst case, because a 5ms query run ten thousand times costs more than a 2 second report nobody runs.

Errors with their context. Not just the stack, but the request that caused it.

```
pulse.WithExcludePaths("/studio/*", "/sentinel/*",
    "/docs/*", "/pulse/*")
```

The dashboards exclude each other, which sounds obvious and is a real bug otherwise: an observability tool observing its own polling fills itself with its own traffic.

WATCH OUT

A story worth carrying, because the shape of it is common.

An earlier version of the request-body capture read the body to record it and put back only the first four kilobytes. Every request carrying a `Content-Length` reached the handler truncated. Uploads from a phone and from `curl` failed; uploads from a browser worked, because browsers send chunked bodies with no `Content-Length`.

So the bug report was "uploads fail on mobile", and nothing in the upload code was wrong. **Middleware that reads a request body and puts it back is a place to look** whenever a request behaves differently depending on who sent it.

Pulse can keep its data in SQLite so traces survive a restart, or stay in an in-memory ring buffer for peak throughput. The trade is the usual one: the ring buffer costs nothing and forgets, the file costs writes and remembers.

Sentinel: the attacks you were not watching

Sentinel is the security suite: a web application firewall, rate limiting, an auth shield and anomaly detection, with a dashboard at `/sentinel`.

What it changes is not that you are attacked. It is that you can see it. A login endpoint takes credential stuffing from the day it is public, and without a dashboard the only evidence is a slightly busier log.

Two configuration details are worth knowing before you meet them.

Development gets relaxed limits. A rate limit tight enough for production blocks you while you are testing, and a developer who has been blocked by their own firewall usually turns the whole thing off.

Its storage is its own. Sentinel persists the threat log, blocked IPs and audit trail through its own adapter, not the application's database handle. Left unset it falls back to a local SQLite file, which is ephemeral inside a container, so every redeploy drops the threat log and the blocked-IP list. Point it at the real database in production.

> **WATCH OUT**
> A mount failure is logged rather than fatal. Sentinel refuses to start on a misconfiguration, and in development that should not take the API down with it.
>
> The consequence to watch for: **a security layer that failed to start looks exactly like one that is running, unless you read the log line**. Check the dashboard after a configuration change rather than assuming.

Health: the one endpoint a load balancer reads

```
{
  "status": "degraded",
  "components": {
    "api":      { "state": "ok" },
    "database": { "state": "ok", "latency_ms": 2 },
    "redis":    { "state": "off",
                  "detail": "REDIS_URL is empty, so caching, jobs
                             and cron are off on purpose" },
    "jobs":     { "state": "unknown" }
  }
}
```

The distinction that makes this endpoint useful is between **off** and **broken**. Redis absent because nobody configured it is a working project without caching. Redis configured and unreachable is an incident. A health check that reports both as "red" gets ignored within a week, and a health check that reports both as "green" is worse.

IF YOU KNOW JAVASCRIPT

The equivalent in most Node projects is `app.get('/health', () => res.send('ok'))`, which answers "is the process running" and nothing else.

The process is almost never the thing that broke. The database connection pool is exhausted, Redis went away, the disk is full. A health check that cannot say which is a status page that is always green during an outage.

The logs, and the ID that joins everything

Every request carries a request ID, returned in a header and attached to every log line it produces. That is what turns "it failed around two o'clock" into one search.

```
grit logs               # the tail of this run
grit logs --errors      # only failures, with stack traces
grit logs --grep 500    # lines matching something
```

`grit start` records everything the API and the frontends print into `.grit/logs/dev.log` and keeps the previous run beside it, which matters more than it sounds: the interesting output is usually from the run that just crashed and scrolled away.

WATCH OUT

Log the error and return a generic message, as chapter forty eight's handler did. The log gets the table name, the query and the stack. The client gets a code and a sentence.

The request ID is what joins them, so it belongs in both: in the response header a user can copy, and in every log line. Without it, a support ticket with a timestamp is a search through a thousand lines around that minute.

Making it a habit

Observability is not installed, it is practised. Four habits that cost nothing:

Read Pulse after building a feature, not after a complaint. The N+1 you find on the day you write the code costs ten minutes.

Check health after every deploy. Not "the site loads", the endpoint.

Look at Sentinel weekly. You will learn what normal traffic looks like, which is the only way to recognise abnormal traffic.

Put the request ID in your error messages. Then a screenshot from a user is actionable.

PROJECT 42: FIND THREE REAL PROBLEMS

Take any application you have built in Part V, put realistic data in it, and go looking.

```
grit seed Invoice --count 5000
grit start
```

One: an N+1 you did not know about. Open the busiest list page, then open Pulse and read the trace. Count the queries. If it is one or two, add a relationship to the list and look again.

Two: the slowest query. Sort Pulse's query list by total time. Explain in `OBSERVED.md` why that query is top: is it slow, or is it run too often? Those need different fixes.

Three: a real attack. Leave the application running on a public port for an hour, or point a credential-stuffing script at your own login endpoint, and then read Sentinel. Record what it saw and what it did.

Four: health under failure. Stop Redis and read `/health`. Stop the database and read it again. Confirm the states differ and that the detail says something a tired person could act on at 3am.

Five: the request ID. Cause an error, find the ID in the response, and find every log line for that request with one command. Time yourself. That number is what observability is worth.

Six: the fix. Fix the N+1 from step one, measure again, and put both numbers in the notes.

YOUR TURN

Turn Pulse off in a staging environment and try to answer "why was the invoice page slow yesterday at 4pm" from logs alone.

Ten minutes of that is the most persuasive argument for instrumentation you will ever get, and it is worth experiencing deliberately rather than during an incident.

ASSIGNMENT

In `part5-building/45-observability`, instrument an application and report what you found.

One. `OBSERVED.md` with three real findings from Pulse: a query count, a slow query with its total time, and one error with its request ID.

Two. A before and after for one fix, with timings from Pulse rather than from a stopwatch.

Three. The health endpoint in three states: everything up, a dependency deliberately off, and a dependency deliberately broken. Paste all three bodies and say which one should page somebody.

Four. One paragraph on what you would add. Every application has a question these tools cannot answer, usually a business one: how many invoices were sent today, how many signups failed at the verification step. Name yours, and say where it would live.

COMMIT AS

```
git commit -m "Observability - Assignment One"
```

46

Testing a Grit application

Chapter eighteen taught the mechanics. This chapter is the harder question: in an application where most of the code was generated, **what is worth testing**, and which tests pay for the time they cost.

The short answer: do not test the generator. Test the things you wrote, the boundaries you configured, and the one property whose failure is silent.

What arrives already written

A new project comes with tests, and they are worth reading before you add any:

`auth_test.go`	Register and login, success and every failure
`user_test.go`	The auth guard, an admin list, a 404, a success
`bench_test.go`	Health check, login, register
`perf_test.go`	Allocation budgets for the per-request hot path
`__tests__/`	Component tests with Vitest and Testing Library
`e2e/`	Playwright: the auth flow and the admin

```
grit test                # Go and frontend
grit test --go --race    # Go only, with the race detector
grit test --cover        # Go coverage
grit test --e2e          # include Playwright, app must be running
```

One report, and suites that do not apply to your architecture are reported as **skipped with the reason** rather than quietly dropped. A test suite that silently does not run is worse than no suite, because it reports green.

The in-memory database trap

This one is worth the whole chapter, because it produces a test that fails six runs in ten and looks like a broken login.

```
db, err := gorm.Open(sqlite.Open(":memory:"), ...)
sqlDB, _ := db.DB()
sqlDB.SetMaxOpenConns(1)     // this line is the whole fix
```

Every connection to `:memory:` gets its own empty database. The pool opens a second connection as soon as two queries overlap, and in this project they do, because registering a user writes an activity row on a goroutine. The login that follows lands on a fresh, empty database, finds no user, and returns 401.

> **WATCH OUT**
> The failure reads as a broken login rather than a test harness problem, which is why it costs an afternoon. The tell is the flakiness: **a test that passes about six runs in ten is almost never a race in your code and almost always a race in your fixture**.
>
> One connection, and only one. If you need concurrency in a test, use a file database in a temporary directory instead.

The same harness makes a second decision worth copying. The email verification token table is **deliberately not migrated**, because that write happens on a goroutine so a slow SMTP call cannot hold the response open. Creating the table would turn a harmless early return into a background writer racing whatever the test does next.

If you are testing verification itself, migrate that table in that test and call the service directly rather than through the register handler.

What is worth testing, and what is not

Do not test	That a generated create returns 201. The generator's own tests cover it, and your copy adds nothing
Test	Every rule you wrote in a service, especially the refusals
Test	The whitelists, by asserting their contents
Test	Ownership isolation between two users
Test	The error boundary: each error kind maps to the status you expect
Test	Anything that broke once. A bug without a test comes back

The whitelist one is not obvious, and chapter forty's date-sorting story is why: the pagination layer **drops** a sort it does not recognise rather than rejecting it, so a missing column fails silently and the endpoint still returns 200.

```
func TestInvoiceSortableColumns(t *testing.T) {
    want := []string{"number", "issued", "due", "total_amount"}
    for _, col := range want {
        if !invoiceListConfig.Sortable[col] {
            t.Errorf("%s is not sortable", col)
        }
```

```
    }
}
```

Four lines, and they catch a class of bug that no endpoint test catches.

Service tests, without HTTP

Most of your tests belong here. The rules live in services, so testing them through HTTP adds a router, a body, a status code and a JSON decode to every assertion, for nothing.

```
func TestCloseRefusesOpenSubtasks(t *testing.T) {
    svc, ctx := newTaskService(t)

    parent := mustCreate(t, svc, ctx, "parent")
    mustCreateChild(t, svc, ctx, parent.ID, "child")

    _, err := svc.Close(ctx, parent.ID)
    if !errors.Is(err, ErrHasOpenSubtasks) {
        t.Fatalf("err = %v, want ErrHasOpenSubtasks", err)
    }
}
```

Note the assertion is on the **sentinel**, not on a message. The message is allowed to change; the identity of the error is the contract, which is chapter eleven's whole point arriving in a test file.

Handler tests, for the contract

Test at the HTTP level only what is about HTTP: the status code, the envelope, the headers.

```
func TestStaleVersionReturns409WithNumericVersion(t *testing.T) {
    api := newTestAPI(t)
    rec := patchInvoice(t, api, 1, "If-Match: 1", `{"title":"x"}`)

    if rec.Code != 409 {
        t.Fatalf("status = %d, want 409", rec.Code)
    }
    var body struct {
        Error struct {
            CurrentVersion int `json:"current_version"`
        } `json:"error"`
    }
    if err := json.Unmarshal(rec.Body.Bytes(), &body); err != nil {
        t.Fatalf("decode: %v", err)
    }
}
```

The decode into an `int` is the assertion. If the server ever regresses to sending `"9"` as a string, this fails, which is the bug chapter thirty three said shipped once.

The highest-value test in the whole project

```
func TestNotesAreIsolatedBetweenUsers(t *testing.T) {
    svc, _ := newNoteService(t)
    ada := ctxFor(t, "ada")
    grace := ctxFor(t, "grace")

    note := mustCreate(t, svc, ada, "ada's note")

    _, err := svc.GetByID(grace, note.ID)
    if !errors.Is(err, ErrNotFound) {
        t.Fatalf("grace could read ada's note: %v", err)
    }
}
```

In a multi-user application this is the test to write first and run in CI forever. The bug it catches is silent, is not a crash, produces no error in any log, and is the one you cannot apologise your way out of.

> **WATCH OUT**
> Write it once per **ownership-scoped resource**, not once per project. The scoping is applied per service, and the one that was added last month by somebody in a hurry is the one missing `s.db(ctx)`.
>
> A table-driven version over every scoped service is about thirty lines and catches the regression on the day it is introduced.

Allocation budgets, and a trap in the toolchain

`perf_test.go` asserts how many allocations the hot path makes:

```
token verification: 56 allocations under go1.26.9
                    41 allocations under go1.27.2
```

A benchmark tells you how long something took on the machine that ran it, which on a shared CI runner moves on its own. An allocation count does not move with the machine or the load, which is what makes these assertions rather than measurements, and why they can fail a build when a benchmark cannot.

> **WATCH OUT**
> Allocation counts **do** move with the Go toolchain, by more than you would guess: fifteen fewer allocations for the same work between two releases.

So a budget is set from the **oldest** toolchain the project supports, because that is the one that allocates most. Set it from whatever is installed on your laptop and it passes locally and fails in CI, which is how that file first failed.

When a budget fails, it is telling you a change allocates more than it used to. That is not automatically wrong: read the change, decide whether the cost is worth it, and if it is, raise the budget **in the same commit**, so the next person can see the new number was a decision.

The frontend, in two layers

Component tests with Vitest and Testing Library, for logic that lives in the browser: a form that disables its button, a list that shows an empty state, a hook that retries. Fast, and they run on every save.

End-to-end tests with Playwright, for the three or four flows that must never break: sign in, create the central record, and whatever your application exists to do. They are slow and they are flaky in proportion to how many you have, so keep the set small and keep it meaningful.

IF YOU KNOW JAVASCRIPT

The temptation is to write end-to-end tests for everything, because they feel like proof. The cost arrives later: a suite that takes twenty minutes and fails for unrelated reasons twice a week gets marked "allowed to fail", and then it is decoration.

Rule of thumb: an end-to-end test for every flow you would stop a release for, and not one more.

What CI should run

Every push	`go build`, `go vet`, Go tests, component tests, type check
Every push	The race detector on the packages with concurrency
Every push	The isolation tests. They are fast and they are the ones that matter
Nightly	End-to-end, the full race suite, the dependency drift check

One practical note from this project's own CI: **never pipe `go test ./...` through `tail`**. Keep the whole log and grep it when something fails. A truncated log turns a five minute diagnosis into a re-run with more logging.

PROJECT 43: MAKE THE SUITE WORTH HAVING

Take the invoicing application from chapter forty one and bring its tests to a state you would defend.

One: the rules. A test for every refusal in the state machine, asserting sentinels rather than messages.

Two: the whitelists. A test asserting the sortable and filterable column sets. Then remove a column from the service, watch the test fail, and confirm that the endpoint test still passes. That gap is the point.

Three: isolation. Two users, every scoped resource, table driven.

Four: the flake. Remove `SetMaxOpenConns(1)` from the test harness and run the suite ten times. Record how many passed. Put it back.

Five: the contract. A handler test for the 409 that decodes `current_version` into an `int`.

Six: end to end. Exactly three Playwright tests: sign in, create an invoice, send it. Time the suite and write the number down.

Seven: CI. A workflow running all of it on push, with the end-to-end suite on a schedule instead. Push a deliberate break and confirm it fails for the right reason.

YOUR TURN

Open your largest test file and count how many tests would still pass if the business rules were deleted.

Every one of those is testing the framework rather than your application. Deleting them makes the suite faster and no weaker, and the exercise tends to be uncomfortable in a useful way.

ASSIGNMENT

In `part5-building/46-testing`, build the suite and report what it is for.

One. Service tests for every rule you wrote, handler tests only for the contract, and the isolation tests table driven across every scoped resource.

Two. The whitelist test, with evidence that removing a column fails it and does not fail an endpoint test.

Three. `TESTS.md` naming, for each suite, the bug class it exists to catch and roughly how long it takes to run.

Four. The flake experiment: ten runs with and without the single connection, with the counts.

Five. One bug you have actually hit in your own work, written as a regression test. If you cannot remember one, go through your commit history for the word “fix” and take the first one.

COMMIT AS

```
git commit -m "Testing Grit - Assignment One"
```

47

Upgrading, deploying, and handing it over

Two things decide whether a framework is a help or a trap, and neither is visible on the day you start: can you take a new version without a rewrite, and can somebody else run this when you are not available.

This chapter closes Part V with both, plus the handover that the last project of your career will thank you for.

The upgrade problem, stated honestly

A framework that generates code into your project has a problem a library does not: it wrote files, you edited them, and the next version wants to write them again.

There are three possible answers, and two of them are bad.

Overwrite	Your edits are gone. Nobody upgrades twice
Never touch	Framework fixes never reach existing projects. Everybody is stranded on the version they started with
Know which	Track what was generated, leave what changed

```
grit upgrade
```

Files Grit wrote and you have not touched are regenerated. Files you have edited are **left alone and reported**, because the upgrade knows the difference: each generated file was recorded when it was written, and a file whose content no longer matches is yours.

```
grit upgrade --diff    # show what it would change in the files it skipped
grit upgrade --plan    # write UPGRADE-PLAN.md, a runbook for those files
grit upgrade --force   # take everything, including your edits
```

> **WATCH OUT**
> `--force` is a real option and it is occasionally right, for instance when you edited a file by accident and want it back.

Commit first. Then `--force` is a diff you can read and revert, rather than a decision you cannot undo. This is true of every generator command that writes into your project, and it is worth a habit rather than a rule.

What an upgrade actually does

Three things, and the second is the one people do not expect.

It regenerates framework-owned files. The admin panel's shared components, the configs, the web app shell.

It repairs generated code it does not own. Your services are yours, and `upgrade` does not rewrite them. But a framework fix that only the **generator** carries reaches new resources and leaves every existing one broken, so there are targeted repairs that go and fix one specific thing in files the upgrade otherwise leaves alone. "A record with a price cannot be saved" is not something to leave broken on existing projects because of a rule about ownership.

It upgrades installed plugins. Left to a command of its own, nobody runs it, and a plugin fix reaches nobody.

WATCH OUT

A repair finds its target by matching an anchor in your file. If your version of that file has drifted far enough, the anchor is not found, and the repair warns and returns rather than guessing.

So **read the warnings from an upgrade**. They are the list of places where the framework wanted to fix something and could not reach it, and each one is a manual edit somebody has to make.

The CLI updates itself

```
grit update
```

Releases follow channels: a plain version is stable, and a version with an `-rc`, `-beta` or `-alpha` suffix is not. `grit update` stays on stable unless you ask otherwise, which is what you want on the machine you work from.

Upgrade the CLI and the project separately, in that order, and **test a fresh project after upgrading the CLI**. A scaffold template change only shows up in something newly generated.

Deployment, in the order of effort

One binary	`go build`, copy, run under systemd. Chapter twenty five
`grit deploy`	Build, upload over SSH, configure systemd, optionally Caddy with automatic TLS
`grit deploy --railway`	Link the project, push the variables from `.env`, upload the API, generate a URL
Docker Compose	`docker-compose.prod.yml`, for a single machine with Postgres and Redis beside it
A platform	Any host that runs a container. The API is one image

Start at the top. A Go API on one server under systemd, with Caddy in front for TLS, serves a great deal more traffic than most applications ever see, and it is a stack you can hold in your head at three in the morning.

> **IF YOU KNOW JAVASCRIPT**
> Coming from Next.js, the thing that takes adjusting is that there is no build output to upload and no Node runtime to match. One file, built on your machine, copied to the server.
>
> The frontends are still Next.js and still need their own hosting or their own container. The simplification is on the API side, and it is the half that used to need the most care.

Two deployment bugs worth not repeating

Both of these were real, and both have the same shape: a path guessed from the wrong directory.

The module is not where you are standing. In a monorepo the Go module is in `apps/api`, and reading `go.mod` from the repository root finds nothing. The deploy announced itself, named the app `grit-app`, and then failed on "go.mod file not found".

A module's base name is not the application's name. The module is `invoices/apps/api`, whose base is `api`, so the systemd unit, the remote directory and the binary were all called `api`. Two Grit projects on one server would have overwritten each other.

> **WATCH OUT**
> The general lesson for any deployment script you write: **derive names from something that identifies the project**, and print every derived value before acting on it. A deploy that says "deploying api to /srv/api" gives somebody the chance to stop it; one that says "deploying" does not.

The checklist before a first deploy

Run these and read the output rather than assuming:

```
grit doctor        # configuration, dependencies, common mistakes
grit routes        # every route, and which group it is on
grit test          # all suites
grit backup        # and then actually restore it somewhere
```

Then the environment:

`APP_URL`	Correct, and matching the real port. Uploads build their URLs from it
`JWT_SECRET`	Generated for this environment, not copied from the example file
Database	Postgres, with a password that was not in a repository
Sentinel storage	Pointed at the real database, not the SQLite file that a redeploy deletes
`MAIL_MAILER`	A real driver, configured, and tested by sending one
Backups	Scheduled, and restored once to prove they work

> **WATCH OUT**
> The one people skip is the restore. A backup that has never been restored is a file, not a backup, and the first restore attempt is not the moment to discover it is missing the uploads or the schema.
>
> Restore into a scratch database, open the application against it, and look at a record. Then you have a backup.

Handing it over

The last thing a project needs is for somebody else to be able to run it. Four files, none of them long:

`README.md` : what this is, the three commands to run it locally, and where the environment variables come from.

`ARCHITECTURE.md` : the request path from route to query, and where rules live. Chapter twenty eight's assignment, kept up to date.

`RUNBOOK.md` : how to deploy, how to roll back, how to take and restore a backup, what to check when it is slow, who to call.

`DECISIONS.md` : why things are the way they are. The sort reason, the currency choice, the reason that endpoint is not cached. Six lines each, and it is the file that stops the next person undoing your work for reasons you already considered.

IF YOU KNOW JAVASCRIPT
Writing these feels like overhead when you are the only person who will ever touch the project. You are not the only person: you in eight months is a different person with none of the context and all of the confidence.

The test for whether they are good enough is simple. Hand somebody the repository, say nothing, and watch where they stop.

Where to go next

You have the language, the standard library, the framework and five applications. Three directions from here, and they are not equal.

Deeper into Go. Part VI builds things from scratch: a parser, a key-value store, a Redis, a Git. That is where the language stops being a tool you use and becomes one you understand.

Deeper into the domain. Pick the application in Part V that was closest to real work and make it genuinely production ready for one real user. The last ten percent is where the learning is.

Deeper into operations. Run something for six months. Watch it break, fix it, and write down what you changed. Nothing in a book teaches what a pager teaches.

PROJECT 44: UPGRADE, DEPLOY, HAND OVER
Take the application you are proudest of from Part V and put it in front of somebody.

One: upgrade. Commit, run `grit upgrade --diff`, read every file it proposes to change, then upgrade. Record in `UPGRADE.md` which files it skipped and why each one was edited.

Two: the warnings. If the upgrade warned about a repair it could not apply, apply it by hand and note what the anchor drift was.

Three: deploy. To a real server or a real platform, with TLS and a domain. Not localhost. Write down every step you took that was not in a command.

Four: break it on purpose. Stop the database in production, watch `/health`, and confirm the application fails the way you expected. Then fix it and time the recovery.

Five: restore. Take a backup, restore it into a second environment, and open the application against the restored data. Confirm a record you created this week is there.

Six: hand it over. Give the repository to somebody who has never seen it, with no verbal explanation, and ask them to get it running locally and deploy a change. Write down every question they asked. Those questions are the missing documentation, and nothing else will find them.

YOUR TURN

Read `UPGRADE-PLAN.md` after running `grit upgrade --plan` on a project you have customised heavily.

It is a list of the places where you and the framework have diverged, which is also a map of where your project is most itself. Some of those divergences are good decisions worth keeping, and some are things you did before you knew the framework had an answer.

ASSIGNMENT

In `part5-building/47-deploy` , document a real deployment of a real application.

One. `RUNBOOK.md` with: deploy, roll back, backup, restore, what to check when it is slow, and the first three things to try during an outage.

Two. `DECISIONS.md` with at least eight decisions and their reasons, in six lines each.

Three. Evidence of a restore: the backup, the restored environment, and a record visible in both.

Four. The upgrade record: which files were skipped, which repairs warned, what you did about each.

Five. The handover test. One person, no explanation, and the list of questions they asked. Then fix the documentation so the next person asks none of them, and say which question was the most embarrassing.

COMMIT AS

```
git commit -m "Deployment - Assignment One"
```

48

Build your own CRUD API from scratch

Every backend job, in the end, asks for the same thing: create a record, read one, list many with a filter, change one, delete one. Parts I to III gave you every piece. This chapter assembles them into a complete API with no framework and no ORM, and then counts what it cost, because that number is the honest case both for and against the rest of this book.

Nothing here imports anything outside the standard library except a database driver.

What we are building

A tasks API, in one binary:

`POST /tasks`	Create, with validation, returning 201
`GET /tasks`	List, paginated, filtered, sorted, with `meta`
`GET /tasks/{id}`	Read one, or 404
`PATCH /tasks/{id}`	Partial update
`DELETE /tasks/{id}`	Delete, returning 204

Plus the things that separate an API from a demo: one response envelope, real status codes, a request ID in the logs, timeouts on the server, a recover boundary, graceful shutdown and tests that run without a port.

The envelope is the one from chapter thirty three, on purpose. You are about to write by hand the contract a generated handler gives you.

Three layers, decided before any code

```
main.go        wiring: config, database, routes, server
http/          router, handlers, middleware, the envelope
task/          the type, its rules, and the store
```

The rule, which is the whole of chapter thirty two said again in your own project: **a handler parses and responds, and decides nothing**. The store talks to the database and knows no HTTP. In between sits the thing that knows the rules.

> **IF YOU KNOW JAVASCRIPT**
> In an Express application this layering is a convention you keep by discipline, and the reason it slips is that `req` and `res` are in scope everywhere, so any file can reach for them.
>
> Here, the store package cannot see an HTTP request even if it wants to: it does not import `net/http`. The architecture is held by the import graph rather than by review, which is why it survives a busy week.

The type and its rules

```
package task

type Task struct {
    ID        int64      `json:"id"`
    Title     string     `json:"title"`
    Notes     string     `json:"notes,omitempty"`
    Done      bool       `json:"done"`
    DueAt     *time.Time `json:"due_at"`
    CreatedAt time.Time  `json:"created_at"`
    UpdatedAt time.Time  `json:"updated_at"`
}
```

`DueAt` is a pointer because a task may genuinely have no due date, and chapter nineteen's lesson applies: a zero `time.Time` serialises as the year one, which is not "none".

Validation lives with the type, returning field errors rather than one string, because the frontend needs to know which input to highlight:

```
type FieldErrors map[string]string

func (e FieldErrors) Error() string { return "validation failed" }

func (t *Task) Validate() error {
    errs := FieldErrors{}
    if strings.TrimSpace(t.Title) == "" {
        errs["title"] = "this field is required"
    }
    if len(t.Title) > 200 {
        errs["title"] = "must be 200 characters or fewer"
    }
    if t.DueAt != nil && t.DueAt.Before(time.Now()) {
        errs["due_at"] = "must be in the future"
    }
    if len(errs) > 0 {
        return errs
    }
```

```
    return nil
}
```

The store

The schema, applied at startup. For a project this size that is enough, and chapter thirty one explains what you give up by not having real migrations:

```
const schema = `
CREATE TABLE IF NOT EXISTS tasks (
  id         INTEGER PRIMARY KEY AUTOINCREMENT,
  title      TEXT    NOT NULL,
  notes      TEXT    NOT NULL DEFAULT '',
  done       BOOLEAN NOT NULL DEFAULT 0,
  due_at     DATETIME,
  created_at DATETIME NOT NULL,
  updated_at DATETIME NOT NULL
);
CREATE INDEX IF NOT EXISTS tasks_done ON tasks(done);`
```

Create, returning the assigned ID:

```
func (s *Store) Create(ctx context.Context, t *Task) error {
    now := time.Now().UTC()
    t.CreatedAt, t.UpdatedAt = now, now

    res, err := s.db.ExecContext(ctx, `
        INSERT INTO tasks (title, notes, done, due_at,
                           created_at, updated_at)
        VALUES (?, ?, ?, ?, ?, ?)`,
        t.Title, t.Notes, t.Done, t.DueAt, t.CreatedAt, t.UpdatedAt)
    if err != nil {
        return fmt.Errorf("insert task: %w", err)
    }

    t.ID, err = res.LastInsertId()
    if err != nil {
        return fmt.Errorf("task id: %w", err)
    }
    return nil
}
```

Read one, with the sentinel the rest of the program will match on:

```
var ErrNotFound = errors.New("task not found")

func (s *Store) Get(ctx context.Context, id int64) (*Task, error) {
```

```
    var t Task
    err := s.db.QueryRowContext(ctx, `
        SELECT id, title, notes, done, due_at, created_at, updated_at
        FROM tasks WHERE id = ?`, id).
        Scan(&t.ID, &t.Title, &t.Notes, &t.Done, &t.DueAt,
            &t.CreatedAt, &t.UpdatedAt)

    if errors.Is(err, sql.ErrNoRows) {
        return nil, ErrNotFound
    }
    if err != nil {
        return nil, fmt.Errorf("get task %d: %w", id, err)
    }
    return &t, nil
}
```

WATCH OUT

`sql.ErrNoRows` must not escape this package. It is a `database/sql` detail, and if a handler matches on it then swapping the store for an in-memory one breaks the handler, which is exactly the coupling the layering was for.

Translate it once, at the edge of the package that produced it. That is chapter eleven's boundary rule, and this is the line where it is easiest to get right and easiest to skip.

Listing, which is where the work actually is

Create and read are five lines each. Listing is where every API grows hair: pagination, a filter, a search, a sort, and a total count that has to agree with the filter.

```
type ListOptions struct {
    Page, PageSize int
    Search         string
    Done           *bool
    Sort           string
    Desc           bool
}
```

Build the `WHERE` once, and use it for both queries:

```
func (s *Store) List(ctx context.Context, o ListOptions) (
    []Task, int, error) {

    where := []string{"1 = 1"}
    args := []any{}

    if o.Search != "" {
```

```
        where = append(where, "title LIKE ?")
        args = append(args, "%"+o.Search+"%")
    }
    if o.Done != nil {
        where = append(where, "done = ?")
        args = append(args, *o.Done)
    }
    clause := strings.Join(where, " AND ")
```

The count comes from the same clause, or your pagination lies:

```
    var total int
    err := s.db.QueryRowContext(ctx,
        "SELECT COUNT(*) FROM tasks WHERE "+clause, args...).
        Scan(&total)
    if err != nil {
        return nil, 0, fmt.Errorf("count tasks: %w", err)
    }
```

And then the page itself. Note what is concatenated and what is a placeholder:

```
    sortCols := map[string]string{
        "title":      "title",
        "due_at":     "due_at",
        "created_at": "created_at",
    }
    col, ok := sortCols[o.Sort]
    if !ok {
        col = "created_at"
    }
    dir := "ASC"
    if o.Desc {
        dir = "DESC"
    }

    q := fmt.Sprintf(`
        SELECT id, title, notes, done, due_at, created_at, updated_at
        FROM tasks WHERE %s ORDER BY %s %s LIMIT ? OFFSET ?`,
        clause, col, dir)

    args = append(args, o.PageSize, (o.Page-1)*o.PageSize)
    rows, err := s.db.QueryContext(ctx, q, args...)
```

WATCH OUT

A column name cannot be a placeholder. `ORDER BY ?` is not valid SQL, so the sort column has to be concatenated into the string, and that is precisely how a sort parameter becomes an injection.

> The map is the defence, and the important part is that it is a **whitelist of allowed values**, not an escape or a check for suspicious characters. Anything not in the map is not a column, so the worst an attacker achieves is the default sort.
>
> This is the same whitelist a generated Grit service carries, and now you know why it exists rather than being a configuration chore.

Finish the rows properly, because the loop is not the end:

```
    defer rows.Close()

    tasks := []Task{}
    for rows.Next() {
        var t Task
        if err := rows.Scan(&t.ID, &t.Title, &t.Notes, &t.Done,
            &t.DueAt, &t.CreatedAt, &t.UpdatedAt); err != nil {
            return nil, 0, fmt.Errorf("scan task: %w", err)
        }
        tasks = append(tasks, t)
    }
    return tasks, total, rows.Err()
}
```

`rows.Err()` is the line people omit for years without noticing. A connection dropped halfway through a result set ends the loop quietly, and without that check you return half a page and call it success.

A router, written rather than imported

Go 1.22 taught `ServeMux` method and wildcard patterns, so in a real project you would use it. Write one once anyway: a router is about forty lines, and knowing that number changes how you think about frameworks.

```
type route struct {
    method  string
    parts   []string       // "tasks", ":id"
    handler http.HandlerFunc
}

type Router struct{ routes []route }

func (r *Router) Handle(method, pattern string,
    h http.HandlerFunc) {
    r.routes = append(r.routes, route{
        method:  method,
        parts:   strings.Split(strings.Trim(pattern, "/"), "/"),
        handler: h,
```

```
    })
}
```

Matching is a comparison, segment by segment, with the parameters collected into the request context as they are found:

```
func (r *Router) ServeHTTP(w http.ResponseWriter,
    req *http.Request) {

    got := strings.Split(strings.Trim(req.URL.Path, "/"), "/")
    allowed := []string{}

    for _, rt := range r.routes {
        params, ok := match(rt.parts, got)
        if !ok {
            continue
        }
        if rt.method != req.Method {
            allowed = append(allowed, rt.method)
            continue
        }
        ctx := context.WithValue(req.Context(), paramsKey, params)
        rt.handler(w, req.WithContext(ctx))
        return
    }

    if len(allowed) > 0 {
        w.Header().Set("Allow", strings.Join(allowed, ", "))
        Fail(w, req, StatusError{405, "METHOD_NOT_ALLOWED",
            "method not allowed"})
        return
    }
    Fail(w, req, StatusError{404, "NOT_FOUND", "no such route"})
}
```

```
func match(pattern, got []string) (map[string]string, bool) {
    if len(pattern) != len(got) {
        return nil, false
    }
    params := map[string]string{}
    for i, p := range pattern {
        if strings.HasPrefix(p, ":") {
            params[p[1:]] = got[i]
            continue
        }
        if p != got[i] {
            return nil, false
        }
```

```
    }
    return params, true
}
```

The 405 is the detail worth copying. A path that exists for another method is not a 404, and an API that cannot tell the difference sends every client on a debugging trip through its own URL building.

One envelope, one place

```
func JSON(w http.ResponseWriter, status int, body any) {
    w.Header().Set("Content-Type", "application/json")
    w.WriteHeader(status)
    if err := json.NewEncoder(w).Encode(body); err != nil {
        slog.Error("write response", "err", err)
    }
}

func OK(w http.ResponseWriter, data any) {
    JSON(w, 200, map[string]any{"data": data})
}

func Created(w http.ResponseWriter, data any, msg string) {
    JSON(w, 201, map[string]any{"data": data, "message": msg})
}

func Paginated(w http.ResponseWriter, data any, m Meta) {
    JSON(w, 200, map[string]any{"data": data, "meta": m})
}
```

And one failure path that works out the status from the error, so no handler decides it twice:

```
func Fail(w http.ResponseWriter, r *http.Request, err error) {
    var fe task.FieldErrors
    var se StatusError

    switch {
    case errors.As(err, &fe):
        write(w, 422, "VALIDATION_ERROR", "validation failed", fe)
    case errors.Is(err, task.ErrNotFound):
        write(w, 404, "NOT_FOUND", "task not found", nil)
    case errors.As(err, &se):
        write(w, se.Status, se.Code, se.Message, nil)
    default:
        slog.Error("unhandled", "err", err,
            "request_id", RequestID(r.Context()))
```

```
        write(w, 500, "INTERNAL", "something went wrong", nil)
    }
}
```

> **WATCH OUT**
> The `default` branch logs the real error and returns a generic one. That split is deliberate: the message a stranger sees must not contain the table name, the query or the file path, and the message you see must contain all three.
>
> The request ID is what joins the two halves. Return it in a header, log it with every line, and a support message saying “it failed at 14:05” becomes a one-command search.

Handlers, which are now boring

```
func (h *Handler) Create(w http.ResponseWriter, r *http.Request) {
    var t task.Task
    if err := json.NewDecoder(r.Body).Decode(&t); err != nil {
        Fail(w, r, StatusError{400, "BAD_JSON", "invalid JSON"})
        return
    }
    if err := t.Validate(); err != nil {
        Fail(w, r, err)
        return
    }
    if err := h.store.Create(r.Context(), &t); err != nil {
        Fail(w, r, err)
        return
    }
    Created(w, t, "Task created successfully")
}
```

Four steps, in the same order, in every handler in the file: parse, validate, call, respond. When a fifth thing appears in a handler it is nearly always a rule, and it belongs one layer down.

The list handler is the one that parses the query, and it clamps rather than trusts:

```
func (h *Handler) List(w http.ResponseWriter, r *http.Request) {
    q := r.URL.Query()
    o := task.ListOptions{
        Page:     atoiDefault(q.Get("page"), 1),
        PageSize: atoiDefault(q.Get("page_size"), 20),
        Search:   q.Get("search"),
        Sort:     q.Get("sort"),
        Desc:     q.Get("order") == "desc",
    }
```

```
    if o.Page < 1 {
        o.Page = 1
    }
    if o.PageSize < 1 || o.PageSize > 100 {
        o.PageSize = 20
    }
    ...
}
```

WATCH OUT
The clamp on `page_size` is not tidiness. Without it, `?page_size=1000000` is a one-request denial of service that your own API happily performs, and it will be found by a crawler rather than by an attacker.

Middleware, as a chain

```
func Chain(h http.Handler, mw ...func(http.Handler) http.Handler) (
    http.Handler) {
    for i := len(mw) - 1; i >= 0; i-- {
        h = mw[i](h)
    }
    return h
}
```

The recover boundary is the one that earns its place on the first bad day:

```
func Recover(next http.Handler) http.Handler {
    return http.HandlerFunc(func(w http.ResponseWriter,
        r *http.Request) {
        defer func() {
            if v := recover(); v != nil {
                slog.Error("panic", "value", v,
                    "stack", string(debug.Stack()))
                write(w, 500, "INTERNAL", "something went wrong",
                    nil)
            }
        }()
        next.ServeHTTP(w, r)
    })
}
```

Chapter twelve said `recover` belongs only at a boundary. This is the boundary: one panic takes down one request rather than the process, and the stack reaches the log instead of the client.

Wiring it up

```
func main() {
    cfg := config.Load()

    db, err := sql.Open("sqlite", cfg.DatabaseURL)
    if err != nil {
        log.Fatalf("open database: %v", err)
    }
    defer db.Close()
    db.SetMaxOpenConns(cfg.MaxConns)

    if _, err := db.Exec(schema); err != nil {
        log.Fatalf("migrate: %v", err)
    }

    r := &Router{}
    h := &Handler{store: task.NewStore(db)}
    r.Handle("POST", "/tasks", h.Create)
    r.Handle("GET", "/tasks", h.List)
    r.Handle("GET", "/tasks/:id", h.Get)
    r.Handle("PATCH", "/tasks/:id", h.Update)
    r.Handle("DELETE", "/tasks/:id", h.Delete)

    srv := &http.Server{
        Addr:         cfg.Addr,
        Handler:      Chain(r, Recover, RequestIDMW, LogMW),
        ReadTimeout:  5 * time.Second,
        WriteTimeout: 10 * time.Second,
        IdleTimeout:  60 * time.Second,
    }
    serveUntilSignal(srv)
}
```

`serveUntilSignal` is chapter twenty one's graceful shutdown, unchanged.

Testing the whole thing without a port

```
func newTestAPI(t *testing.T) http.Handler {
    t.Helper()
    db, err := sql.Open("sqlite", ":memory:")
    if err != nil {
        t.Fatalf("open: %v", err)
    }
    t.Cleanup(func() { db.Close() })
    if _, err := db.Exec(schema); err != nil {
        t.Fatalf("schema: %v", err)
    }
    return routes(&Handler{store: task.NewStore(db)})
```

```
}

func TestCreateRejectsEmptyTitle(t *testing.T) {
    api := newTestAPI(t)
    req := httptest.NewRequest("POST", "/tasks",
        strings.NewReader(`{"title":""}`))
    rec := httptest.NewRecorder()

    api.ServeHTTP(rec, req)

    if rec.Code != 422 {
        t.Fatalf("status = %d, want 422", rec.Code)
    }
    if !strings.Contains(rec.Body.String(), "title") {
        t.Errorf("body should name the field, got %s", rec.Body)
    }
}
```

An in-memory database per test, no fixtures to clean up, no ports to collide, and the tests run in parallel. This is the shape every API test in your career should have.

Now count it

When it works, count the lines:

```
find . -name '*.go' -not -name '*_test.go' | xargs wc -l | tail -1
```

For one resource, done properly, it lands somewhere around six hundred lines, plus tests. Most of it is not the task at all: it is the envelope, the router, the middleware, the pagination and the whitelist, and all of that is written once and then reused.

Which is the interesting part. **The second resource costs you maybe a hundred and fifty lines**, because the machinery is already there. The tenth costs the same hundred and fifty. That is D1 from chapter twenty six, measured in your own editor rather than taken on trust.

> **IF YOU KNOW JAVASCRIPT**
> Now put this chapter next to chapter twenty nine. `grit generate resource Task` writes the equivalent of these six hundred lines, plus the TypeScript types, the Zod schema, the React Query hooks and an admin screen, from one description.
>
> The reason to have written this chapter first is not to make that look impressive. It is so that when the generated service does something you did not expect, you are reading familiar code rather than magic, and so you can tell, specifically, which parts of it you could do without.

PROJECT 40: TAKE IT PAST CRUD

Build the API above, then add the three things that make it a real service.

One: a transaction. Add `POST /tasks/{id}/complete` that marks the task done and writes a row to a `task_events` table. Both or neither, in one transaction, with a test that proves a failure on the second write leaves no event and an unfinished task.

Two: concurrency. Add a `version` column. `PATCH` requires an `If-Match` header, and a stale version returns 409 with the current one as a **number**. Chapter thirty three explains what the quoted string cost.

Three: a bulk endpoint. `POST /tasks/bulk` takes an array and reports per-row results rather than failing the whole request on row seventeen. Decide deliberately whether a partial success is a 200 or a 207, and write the reason in a comment.

Four: measure it. Put a thousand rows in and time the list endpoint with and without the index on `done`. Keep both numbers.

YOUR TURN

Delete the sort whitelist and point the API at a throwaway database. Then send `?sort=title; DROP TABLE tasks --` and watch what happens to your data.

Do it once, on purpose, in a directory you do not mind losing. An injection you have performed yourself stops being a checklist item and becomes something you can see in a diff.

ASSIGNMENT

Build the whole API in `part6-from-scratch/48-crud-api`, with the three additions from Project 40 and the tests passing.

One. All five endpoints, the envelope, the router, the middleware chain and graceful shutdown. No third-party imports except the database driver.

Two. Tests covering: a 201 with the body, a 422 naming the field, a 404, a 405 with an `Allow` header, and a list whose `meta.total` respects the filter.

Three. `wc -l` on your non-test Go files, recorded in `NOTES.md` with the time it took you. Then add a second resource and record both again. The ratio between the two is the number this book has been arguing about since chapter twenty six.

Four. One paragraph: after writing it, which parts would you want a framework to hand you, and which would you rather keep writing yourself? Keep the answer. Chapter twenty nine is more interesting if you disagree with it from a position of having done the work.

COMMIT AS

```
git commit -m "CRUD API - Assignment One"
```

49

Build your own CLI framework

Every tool you write grows a command line eventually, and most people reach straight for Cobra without ever knowing what it does. A command framework is about three hundred lines. Writing one makes every CLI you use afterwards legible, and it is the fastest way to learn what separates a tool people enjoy from one they tolerate.

We are building: nested subcommands, per-command flags, generated help, "did you mean", correct exit codes, and cancellation on Ctrl-C.

What a good CLI does, before any code

The design decisions matter more than the parser:

Exit codes	0 success, 1 failure, 2 usage error. Scripts read these
Streams	Results to stdout, diagnostics to stderr. Always
Help	Generated from the commands, never written twice
Errors	Say what to do next, not just what went wrong
Cancellation	Ctrl-C stops the work, it does not kill it mid-write

> **IF YOU KNOW JAVASCRIPT**
> The stream rule is the one JavaScript developers break most, because `console.log` goes to stdout and it is right there.
>
> The consequence shows up the first time somebody writes `mytool list > out.json` and finds a progress message in the middle of their JSON. Results on stdout, everything else on stderr, and the tool becomes something people can build on.

The command

```
type Command struct {
    Name  string
    Short string                    // one line, for the index
    Long  string                    // paragraphs, for the help page
    Usage string                    // "add <file> [more...]"
```

```
    Flags *flag.FlagSet
    Run   func(ctx context.Context, args []string) error

    subs map[string]*Command
}

func (c *Command) Add(sub *Command) {
    if c.subs == nil {
        c.subs = map[string]*Command{}
    }
    c.subs[sub.Name] = sub
}
```

Everything else in this chapter is built from that struct. Note what it does not have: no global state, no `init()` registration, no singleton root. A command tree is a value you can build in a test.

Dispatch

Walking the tree is a loop, not recursion, and it stops at the first argument that is not a known subcommand:

```
func (c *Command) find(args []string) (*Command, []string) {
    cmd := c
    for len(args) > 0 {
        sub, ok := cmd.subs[args[0]]
        if !ok {
            return cmd, args
        }
        cmd, args = sub, args[1:]
    }
    return cmd, args
}
```

So `tool db migrate --dry-run status.json` finds `migrate` and hands it `["--dry-run", "status.json"]`.

Then parse that command's own flags and run it:

```
func (c *Command) Execute(ctx context.Context, argv []string) error {
    cmd, rest := c.find(argv)

    if cmd.Flags != nil {
        cmd.Flags.SetOutput(io.Discard) // we print our own errors
        if err := cmd.Flags.Parse(rest); err != nil {
            return usageError{cmd: cmd, err: err}
        }
        rest = cmd.Flags.Args()
```

```
    }

    if cmd.Run == nil {
        return usageError{cmd: cmd, err: errNoRun}
    }
    return cmd.Run(ctx, rest)
}
```

> **WATCH OUT**
> `flag.Parse` stops at the first non-flag argument. `tool add file.txt -v` leaves `-v` in the arguments rather than setting the flag, and the user sees nothing happen.
>
> Go's own tools accept this; GNU tools do not. Decide which you are, say so in the help, and if you want flags anywhere, sort the arguments before parsing rather than writing your own parser.

Two kinds of failure

This is the design decision that most improves a CLI, and almost nobody makes it deliberately.

```
type usageError struct {
    cmd *Command
    err error
}
```

A usage error means the command was typed wrong: an unknown flag, a missing argument. Print the error, then the usage, and exit 2. The flag list is exactly what the user needs.

Every other error means the command was fine and the work failed: a file was missing, the server refused, the database was down. Print the error and exit 1, **without the usage**.

> **WATCH OUT**
> Printing the flag list under "connection refused" is the most common CLI mistake there is. It buries the one sentence that matters under forty lines of options, and it implies the user typed something wrong when they did not.
>
> Grit's own CLI sets `SilenceUsage` the moment it knows the flags parsed and the project was found, for exactly this reason. If you take one thing from this chapter into a tool you already maintain, take this one.

```
func Main(root *Command, argv []string) int {
    ctx := withSignals(context.Background())

    err := root.Execute(ctx, argv)
    switch {
    case err == nil:
        return 0
    case errors.Is(err, context.Canceled):
        fmt.Fprintln(os.Stderr, "cancelled")
        return 130                          // 128 + SIGINT
    case errors.As(err, new(usageError)):
        var ue usageError
        errors.As(err, &ue)
        fmt.Fprintf(os.Stderr, "error: %v\n\n", ue.err)
        ue.cmd.PrintUsage(os.Stderr)
        return 2
    default:
        fmt.Fprintf(os.Stderr, "error: %v\n", err)
        return 1
    }
}
```

`130` is the conventional code for "killed by Ctrl-C", and scripts do check it.

Help, generated

Help that is written by hand goes stale on the first new flag. Generate it:

```
func (c *Command) PrintUsage(w io.Writer) {
    if c.Long != "" {
        fmt.Fprintf(w, "%s\n\n", strings.TrimSpace(c.Long))
    }
    fmt.Fprintf(w, "Usage:\n  %s %s\n", c.path(), c.Usage)

    if len(c.subs) > 0 {
        fmt.Fprintf(w, "\nCommands:\n")
        for _, name := range sortedKeys(c.subs) {
            fmt.Fprintf(w, "  %-12s %s\n",
                name, c.subs[name].Short)
        }
    }
    if c.Flags != nil {
        fmt.Fprintf(w, "\nFlags:\n")
        c.Flags.SetOutput(w)
        c.Flags.PrintDefaults()
    }
    fmt.Fprintf(w, "\nRun '%s <command> --help' for more.\n", c.path())
}
```

The `io.Writer` parameter is not decoration. It is what makes the help testable without capturing os.Stdout, which is chapter twenty's lesson applied to a command line.

Did you mean

A typo should not produce a wall of help. It should produce one suggestion:

```
tool: unknown command "migarte"

Did you mean?
    migrate
```

Edit distance, bounded so nonsense suggests nothing:

```
func suggest(want string, have []string) []string {
    limit := len(want)/3 + 1
    var out []string
    for _, candidate := range have {
        if distance(want, candidate) <= limit {
            out = append(out, candidate)
        }
    }
    return out
}
```

`distance` is Levenshtein, about twenty lines with two rows rather than a full matrix. Writing it once is worth it: the same function powers "did you mean" in every tool you will ever write, and in your API's error messages for an unknown field.

Cancellation that is not a kill

```
func withSignals(parent context.Context) context.Context {
    ctx, cancel := signal.NotifyContext(parent,
        os.Interrupt, syscall.SIGTERM)

    go func() {
        <-ctx.Done()
        cancel()
        // A second Ctrl-C means business.
        c := make(chan os.Signal, 1)
        signal.Notify(c, os.Interrupt)
        <-c
        os.Exit(130)
    }()
    return ctx
}
```

The first interrupt cancels the context, so the command can finish writing the file it is in the middle of and exit cleanly. The second one exits immediately, because a user pressing Ctrl-C twice has stopped asking politely.

Every long-running command then has one obligation, from chapter sixteen: **check `ctx.Done()`**. A command that ignores it is a command Ctrl-C cannot stop.

Output humans read and machines parse

The same command often has two audiences:

```
if *asJSON {
    return json.NewEncoder(out).Encode(results)
}
for _, r := range results {
    fmt.Fprintf(out, "%-20s %s\n", r.Name, r.Status)
}
```

Two rules make this work. **The JSON goes to stdout and nothing else does**, or piping it breaks. And **colour is disabled when stdout is not a terminal**, or your pipeline fills with escape codes:

```
func useColour(w io.Writer) bool {
    if os.Getenv("NO_COLOR") != "" {
        return false
    }
    f, ok := w.(*os.File)
    if !ok {
        return false
    }
    info, err := f.Stat()
    return err == nil && info.Mode()&os.ModeCharDevice != 0
}
```

`NO_COLOR` is an agreed convention. Honour it, and the people who need it will never have to ask you to.

Testing a CLI

Because nothing is global, a test builds a tree and runs it:

```
func TestMigrateRejectsUnknownFlag(t *testing.T) {
    var out bytes.Buffer
    root := newRoot(&out)

    code := Main(root, []string{"db", "migrate", "--nope"})

    if code != 2 {
```

```
        t.Fatalf("exit = %d, want 2 for a usage error", code)
    }
    if !strings.Contains(out.String(), "Usage:") {
        t.Error("a usage error should print the usage")
    }
}
```

IF YOU KNOW JAVASCRIPT

Compare this with testing a CLI built on package-level variables and `init()` registration, which is the shape most tutorials teach.

There, every test shares one root, flags keep values between tests, and the only way to assert on output is to swap `os.Stdout`. The design in this chapter is not cleverer; it just passes its dependencies in, which is the same argument chapter ten made about interfaces.

PROJECT 45: BUILD THE FRAMEWORK, THEN A REAL TOOL WITH IT

In `part6-from-scratch/49-cli`, build the framework above as a package, and then a tool that uses it.

One: the framework. `Command`, `Add`, `find`, `Execute`, `PrintUsage`, `Main`, suggestions, signals. No global state.

Two: the tool. A `notes` CLI with real subcommands:

```
notes add "buy milk" --tag shopping
notes list --tag shopping --json
notes done 3
notes export > notes.json
```

Store them in a JSON file under the user's config directory, found with `os.UserConfigDir`.

Three: the exit codes. Verify with a script:

```
notes list --nope;      echo "usage:   $?"   # 2
notes done 999;         echo "failure: $?"   # 1
notes list > /dev/null; echo "ok:      $?"   # 0
```

Four: the pipe. `notes list --json | jq '.[0].text'` must work, which means no progress output on stdout and no colour codes when piped.

Five: cancellation. Add `notes import <big-file>` that reports progress and honours `ctx`. Interrupt it halfway and confirm the file it was writing is not left half written. Then interrupt twice and confirm it exits immediately.

Six: the typo. `notes lst` suggests `list`. `notes xyzzy` suggests nothing and exits 2.

Seven: tests. Every exit code, the help output, the suggestion, and one test that runs the whole tree against a temporary config directory.

YOUR TURN

Run `--help` on three tools you use daily and read them as a designer rather than a user.

Which one tells you what to do next when you get it wrong? Which one buries the useful line? The difference is almost never the parser; it is the three decisions at the top of this chapter.

ASSIGNMENT

Build the framework and the `notes` tool, and make it something you would actually install.

One. The framework as its own package with no global state, and tests that build a command tree per test.

Two. The tool, with at least four subcommands, one of them nested two levels deep.

Three. `CLI.md` recording the exit code for every failure your tool can produce, and which stream each message goes to.

Four. Proof it composes: one shell pipeline using your tool with `jq`, `grep` and a redirect, pasted with its output.

Five. Cross compile it for three platforms, as in chapter twenty five, and install it on your own machine. Use it for a week before you decide it is finished, and note the first thing that annoyed you.

COMMIT AS

```
git commit -m "CLI Framework - Assignment One"
```

50

Build your own JSON parser

JSON is the smallest grammar worth parsing properly. It has five kinds of value, no ambiguity, and a specification you can read in an afternoon. Once you have written a parser for it, every other parser you meet, a config format, a query language, a template engine, is the same shape with more rules.

We are building a lexer, a recursive descent parser, errors that name a line and a column, and the defences a parser needs when the input comes from strangers.

The grammar, in full

```
value   := object | array | string | number | true | false | null
object  := '{' (string ':' value (',' string ':' value)*)? '}'
array   := '[' (value (',' value)*)? ']'
```

That is the whole language. Strings and numbers have their own rules, which is where most of the work turns out to be.

> **IF YOU KNOW JAVASCRIPT**
> `JSON.parse` in the browser is roughly this, plus a reviver function and some extremely optimised string handling.
>
> The one behavioural difference worth knowing before we start: JavaScript numbers are float64, so `JSON.parse('{"id": 9007199254740993}')` gives you 9007199254740992. The parser is not wrong, the type is. We will hit the same decision later in this chapter, and in Go we get to choose differently.

Tokens

```go
type Kind int

const (
    EOF Kind = iota
    LBrace; RBrace; LBracket; RBracket
    Comma; Colon
    String; Number; True; False; Null
```

```
)

type Token struct {
    Kind  Kind
    Text  string   // the decoded string, or the raw number
    Line  int
    Col   int
}
```

The position goes on the token, not on the parser, because the parser reports errors about tokens it has already read. Retrofitting positions later is miserable, and every parser author does it once.

The lexer

```
type Lexer struct {
    src       []byte
    pos       int
    line, col int
}

func (l *Lexer) Next() (Token, error) {
    l.skipSpace()
    if l.pos >= len(l.src) {
        return Token{Kind: EOF, Line: l.line, Col: l.col}, nil
    }

    start := l.mark()
    switch c := l.src[l.pos]; {
    case c == '{':
        l.advance()
        return start.token(LBrace, "{"), nil
    case c == '"':
        return l.lexString(start)
    case c == '-' || (c >= '0' && c <= '9'):
        return l.lexNumber(start)
    case c == 't' || c == 'f' || c == 'n':
        return l.lexLiteral(start)
    default:
        return Token{}, l.errorf(start,
            "unexpected character %q", c)
    }
}
```

The dull part is `advance`, and it is the part to get right first:

```
func (l *Lexer) advance() byte {
    c := l.src[l.pos]
    l.pos++
    if c == '\n' {
        l.line++
        l.col = 1
    } else {
        l.col++
    }
    return c
}
```

Every byte goes through it, so the line and column are always correct, and no later code has to remember to update them.

Strings, where the work is

```
func (l *Lexer) lexString(start mark) (Token, error) {
    l.advance()                  // the opening quote
    var sb strings.Builder

    for {
        if l.pos >= len(l.src) {
            return Token{}, l.errorf(start, "unterminated string")
        }
        c := l.advance()
        switch {
        case c == '"':
            return start.token(String, sb.String()), nil
        case c == '\\':
            if err := l.escape(&sb); err != nil {
                return Token{}, err
            }
        case c < 0x20:
            return Token{}, l.errorf(start,
                "control character %#02x in string", c)
        default:
            sb.WriteByte(c)
        }
    }
}
```

Three correctness points that separate a parser from a toy.

Raw control characters are illegal inside a JSON string. A literal newline in a string is a malformed document, and accepting it means your parser disagrees with everybody else's.

The escapes are a fixed list: `" \ / b f n r t` and `uXXXX`. Anything else is an error, not a passthrough.

Surrogate pairs are real. 😀 is one emoji expressed as two escapes, and handling each in isolation produces two broken runes:

```
if utf16.IsSurrogate(r) {
    r2, err := l.readEscapeU()   // must be the low half
    if err != nil {
        return err
    }
    sb.WriteRune(utf16.DecodeRune(r, r2))
    return nil
}
```

> **WATCH OUT**
> `utf16.DecodeRune` returns `U+FFFD` for a pair that does not combine, which means a lone high surrogate becomes a replacement character rather than an error.
>
> Decide deliberately which you want. Being strict is right for a validator and wrong for a parser that has to read whatever a legacy system emits, and either way it should be a decision with a comment and not an accident.

Numbers, and the decision JavaScript cannot make

```
func (l *Lexer) lexNumber(start mark) (Token, error) {
    begin := l.pos
    l.accept("-")
    l.acceptDigits()
    if l.accept(".") {
        l.acceptDigits()
    }
    if l.accept("eE") {
        l.accept("+-")
        l.acceptDigits()
    }
    raw := string(l.src[begin:l.pos])
    if !validNumber(raw) {
        return Token{}, l.errorf(start, "invalid number %q", raw)
    }
    return start.token(Number, raw), nil
}
```

Note what it does **not** do: it does not convert. The token carries the raw text, and the decision about what Go type to produce belongs to the caller.

```
type Options struct {
    // UseNumber keeps numbers as their original text rather than
    // converting to float64.
    UseNumber bool
}
```

WATCH OUT

This is not a detail. A 64-bit ID, a currency amount in minor units, a nanosecond timestamp: all of them exceed the 53 bits a float64 can hold exactly, and converting silently changes the value.

`9007199254740993` becomes `9007199254740992`, and no error is produced anywhere. If your parser is reading somebody's API responses, keeping the raw text and converting on demand is the only safe default. This is the same lesson as chapter forty's money, arrived at from the parsing side.

The parser

Recursive descent means one function per production, and the structure reads like the grammar:

```
func (p *Parser) parseValue() (any, error) {
    switch p.tok.Kind {
    case LBrace:
        return p.parseObject()
    case LBracket:
        return p.parseArray()
    case String:
        v := p.tok.Text
        p.next()
        return v, nil
    case Number:
        return p.number()
    case True, False:
        v := p.tok.Kind == True
        p.next()
        return v, nil
    case Null:
        p.next()
        return nil, nil
    default:
        return nil, p.errorf("expected a value, found %s",
            p.tok.Kind)
    }
}
```

An object is a loop over pairs:

```
func (p *Parser) parseObject() (map[string]any, error) {
    p.next()                        // past '{'
    out := map[string]any{}

    if p.tok.Kind == RBrace {
        p.next()
        return out, nil             // {}
    }
    for {
        if p.tok.Kind != String {
            return nil, p.errorf("expected a key, found %s",
                p.tok.Kind)
        }
        key := p.tok.Text
        p.next()

        if err := p.expect(Colon); err != nil {
            return nil, err
        }
        val, err := p.parseValue()
        if err != nil {
            return nil, err
        }
        out[key] = val

        if p.tok.Kind == Comma {
            p.next()
            continue
        }
        return out, p.expect(RBrace)
    }
}
```

Arrays are the same shape with no keys, and that is the parser.

Two defences against hostile input

A parser that will be fed data from outside needs both of these, and neither is obvious until you have been bitten.

A depth limit. `[[[[[[...]]]]]]` with ten thousand levels recurses ten thousand times, and a deep enough document is a stack overflow, which in Go is a crash you cannot recover from.

```
func (p *Parser) enter() error {
    p.depth++
    if p.depth > p.maxDepth {       // 10000 is generous
        return p.errorf("nesting too deep")
```

```
    }
    return nil
}
```

A size limit, upstream. Not the parser's job, but the parser is where the cost appears: a hundred megabyte document becomes several hundred megabytes of `map[string]any`. Cap the reader with `io.LimitReader` before parsing, and remember chapter forty two's decompression bomb is the same family of attack.

> **WATCH OUT**
> Duplicate keys are where the specification shrugs: `{"a":1,"a":2}` is legal and the result is "implementation defined".
>
> Most parsers keep the last. Some keep the first. A few error. The attack this enables is real: a document parsed by two systems that disagree is a way to show one value to a validator and another to a consumer.
>
> Pick last-wins to match everybody else, and offer a strict mode that errors. Then write it down, because the next person will assume whichever one you did not choose.

Errors people can act on

```
parse error at line 7, column 24: expected ',' or '}', found string
   7 | { "name": "Ada" "role": "engineer" }
     |                 ^
```

Three pieces: the position, what was expected, and what was found. The caret line costs about fifteen lines of code and is the difference between a parser people debug with and one they swear at.

Testing it properly

Table tests for the cases you thought of. Then two techniques that find the ones you did not.

Differential fuzzing against the standard library:

```
func FuzzAgainstStdlib(f *testing.F) {
    f.Add([]byte(`{"a":[1,2,{"b":null}]}`))

    f.Fuzz(func(t *testing.T, data []byte) {
        var want any
        wantErr := json.Unmarshal(data, &want)

        got, gotErr := Parse(data)
```

```
        if (wantErr == nil) != (gotErr == nil) {
            t.Fatalf("stdlib err=%v, ours err=%v", wantErr, gotErr)
        }
        if wantErr == nil && !reflect.DeepEqual(want, got) {
            t.Fatalf("mismatch:\n want %#v\n got  %#v", want, got)
        }
    })
}
```

```
go test -fuzz FuzzAgainstStdlib -fuzztime 60s
```

Sixty seconds of that finds the cases a human table never contains: a lone surrogate, a number like `-0`, `1e999`, a string of nothing but escapes.

The JSON test suite. There is a well-known corpus of files named `y_` (must accept), `n_` (must reject) and `i_` (implementation defined). Running against it is how you find out your parser is not as correct as it feels.

> **IF YOU KNOW JAVASCRIPT**
> Fuzzing is built into `go test`, which is unusual and underused. In most languages it is a separate tool with its own configuration, so people do not bother.
>
> Here it is three lines and a flag, and for anything that parses untrusted input it finds more real bugs per minute than any other technique available to you.

How fast is it

Benchmark against `encoding/json` and report the number honestly. A clean hand-written parser usually lands in the same order of magnitude for parsing into `any`, and loses badly to the standard library's struct decoding, which has years of optimisation in it.

That is the right outcome. You did not build this to replace the standard library. You built it so that the next time you read a stack trace through `encoding/json`, you know what each frame is doing.

> **PROJECT 46: A PARSER THAT PASSES THE SUITE**
> In `part6-from-scratch/50-json`, build it completely.
>
> **One: lexer and parser**, producing `any`, with positions on every token.
>
> **Two: errors** with line, column, expected and found, plus the caret line.
>
> **Three: the number decision.** Implement `UseNumber`, and write a test with `9007199254740993` proving the default loses it and the option keeps it.

Four: surrogates. A test with an emoji written as a surrogate pair, one with a lone high surrogate, and your documented behaviour for each.

Five: defences. A depth limit with a test that would otherwise overflow the stack, and a documented size limit.

Six: the fuzzer. Run the differential fuzz for five minutes. Every difference it finds is a bug in your parser until proven otherwise. Record the first three.

Seven: the suite. Run the standard conformance corpus and record your pass rate for `y_`, `n_` and `i_` separately. Getting `n_` high is harder and more interesting than `y_`.

Eight: the benchmark. Yours against `encoding/json` on a 1 MB document, in `RESULTS.md`, with one sentence on what you would optimise first.

YOUR TURN

Feed your parser a 50 MB JSON file and watch the memory with `/usr/bin/time -v` or the Task Manager.

The ratio between the file size and the resident memory is the real cost of `map[string]any`, and it is the argument for streaming, which is the next thing to build: a decoder that yields tokens rather than constructing the whole tree.

ASSIGNMENT

Build the parser, then prove it against things you did not write.

One. The full parser with positions, errors, the depth limit and the number option.

Two. `CONFORMANCE.md`: your pass rate on the standard corpus, broken into the three categories, with three specific files you fail and why.

Three. Five minutes of differential fuzzing, with the first three differences it found and the fix for each.

Four. The number test, with both values printed.

Five. One paragraph comparing your error messages with `encoding/json`'s for the same malformed document. Paste both. If yours is not better, say so and say what you would change.

COMMIT AS

```
git commit -m "JSON Parser - Assignment One"
```

51

Build your own key-value store: a write-ahead log and crash recovery

A map in memory is a key-value store until the process restarts. This chapter builds one that survives a crash, which turns out to be a question about exactly one system call.

Everything here uses only the standard library, and everything you need is in Parts I to III.

The problem, stated precisely

```
m := map[string]string{}
m["user:1"] = "Ada"
// power cut
```

The data is gone. The obvious fix, writing the map to a file on every change, is both slow and **still wrong**: a crash halfway through writing leaves a corrupt file, and you have traded losing everything for losing everything in a less obvious way.

The standard answer is a **write-ahead log**. Before changing memory, append the change to a file. On startup, replay the file. Appending is fast because it is sequential, and a partially written record at the end is detectable and discardable.

> **IF YOU KNOW JAVASCRIPT**
> This is what Postgres, SQLite, Kafka and every serious database do. The WAL is the durable thing; the in-memory structure is a cache of it that can be rebuilt. Once you have written one, a great deal of database documentation stops being mysterious.

The record format

A log is a sequence of records. Each needs a length, a checksum and a payload:

```
+--------+----------+--------+---------+-------+---------+
| crc32  | keyLen   | valLen |   op    |  key  |  value  |
| 4 byte | 4 byte   | 4 byte | 1 byte  |  ...  |   ...   |
+--------+----------+--------+---------+-------+---------+
```

The checksum is what makes a half-written record detectable. Without it a truncated write looks like a valid record with garbage in it.

```
package kv

import (
    "bufio"
    "encoding/binary"
    "errors"
    "fmt"
    "hash/crc32"
    "io"
    "os"
    "sync"
)

type op byte

const (
    opSet op = 1
    opDel op = 2
)

const headerSize = 4 + 4 + 4 + 1   // crc, keyLen, valLen, op
```

Encoding a record

```
func encode(o op, key, val []byte) []byte {
    buf := make([]byte, headerSize+len(key)+len(val))

    binary.BigEndian.PutUint32(buf[4:8], uint32(len(key)))
    binary.BigEndian.PutUint32(buf[8:12], uint32(len(val)))
    buf[12] = byte(o)
    copy(buf[headerSize:], key)
    copy(buf[headerSize+len(key):], val)

    // The checksum covers everything after itself.
    sum := crc32.ChecksumIEEE(buf[4:])
    binary.BigEndian.PutUint32(buf[0:4], sum)
    return buf
}
```

`encoding/binary` writes fixed-width integers in a defined byte order. Big-endian is conventional for file formats; the only rule that matters is that the reader and writer agree.

WATCH OUT

> Do not use `fmt.Sprintf` or JSON for a log format. Both are variable-width, so you cannot know how many bytes to read without scanning for a delimiter, and a delimiter can appear inside a value. A length prefix removes the question entirely: read the header, then read exactly that many bytes.

The store

```
type Store struct {
    mu   sync.RWMutex
    data map[string][]byte

    log  *os.File
    w    *bufio.Writer
    path string
}

func Open(path string) (*Store, error) {
    f, err := os.OpenFile(path, os.O_RDWR|os.O_CREATE|os.O_APPEND, 0o600)
    if err != nil {
        return nil, fmt.Errorf("opening log %s: %w", path, err)
    }

    s := &Store{
        data: map[string][]byte{},
        log:  f,
        w:    bufio.NewWriter(f),
        path: path,
    }

    // Rebuild memory from the log before accepting any writes.
    if err := s.replay(); err != nil {
        f.Close()
        return nil, fmt.Errorf("replaying log: %w", err)
    }
    return s, nil
}

func (s *Store) Close() error {
    if err := s.w.Flush(); err != nil {
        s.log.Close()
        return fmt.Errorf("flushing: %w", err)
    }
    return s.log.Close()
}
```

`O_APPEND` matters: every write goes to the end regardless of where the file offset happens to be, which is what makes concurrent appends safe at the operating system level.

Replay, and tolerating a torn tail

```
func (s *Store) replay() error {
    if _, err := s.log.Seek(0, io.SeekStart); err != nil {
        return err
    }
    r := bufio.NewReader(s.log)
    header := make([]byte, headerSize)
    offset := int64(0)

    for {
        if _, err := io.ReadFull(r, header); err != nil {
            if errors.Is(err, io.EOF) {
                break                       // clean end
            }
            if errors.Is(err, io.ErrUnexpectedEOF) {
                break                       // torn header: stop here
            }
            return err
        }

        sum := binary.BigEndian.Uint32(header[0:4])
        keyLen := binary.BigEndian.Uint32(header[4:8])
        valLen := binary.BigEndian.Uint32(header[8:12])
        o := op(header[12])

        body := make([]byte, keyLen+valLen)
        if _, err := io.ReadFull(r, body); err != nil {
            break                           // torn body: stop here
        }

        // Verify before trusting any of it.
        check := crc32.ChecksumIEEE(append(header[4:], body...))
        if check != sum {
            break                           // corrupt: the log ends here
        }

        key := string(body[:keyLen])
        switch o {
        case opSet:
            s.data[key] = body[keyLen:]
        case opDel:
            delete(s.data, key)
        }
        offset += int64(headerSize) + int64(len(body))
    }

    // Discard anything after the last good record, so the next append does
    // not sit behind garbage.
    if err := s.log.Truncate(offset); err != nil {
```

```
        return fmt.Errorf("truncating to %d: %w", offset, err)
    }
    _, err := s.log.Seek(0, io.SeekEnd)
    return err
}
```

The three `break`s are the whole crash-recovery story. A torn header, a torn body and a failed checksum all mean the same thing: **the log ends at the last good record**, and everything after it was never acknowledged to a caller, so discarding it is correct.

> **WATCH OUT**
> `append(header[4:], body...)` is the slice trap from chapter six wearing a disguise. `header[4:]` has spare capacity, so `append` may write into `header` rather than allocating. It happens to be harmless here because `header` is re-read next iteration, but it is exactly the pattern that corrupts data elsewhere. The careful version:
>
> ```
> h := crc32.NewIEEE()
> h.Write(header[4:])
> h.Write(body)
> check := h.Sum32()
> ```
>
> Which is also faster, because it allocates nothing.

Writing

```
func (s *Store) Set(key string, val []byte) error {
    s.mu.Lock()
    defer s.mu.Unlock()

    if _, err := s.w.Write(encode(opSet, []byte(key), val)); err != nil {
        return fmt.Errorf("appending set: %w", err)
    }
    if err := s.sync(); err != nil {
        return err
    }

    // Memory is updated only after the log is durable. The other order
    // means a reader can see a value that a crash will erase.
    s.data[key] = val
    return nil
}

func (s *Store) Delete(key string) error {
    s.mu.Lock()
    defer s.mu.Unlock()
```

```
    if _, err := s.w.Write(encode(opDel, []byte(key), nil)); err != nil {
        return fmt.Errorf("appending delete: %w", err)
    }
    if err := s.sync(); err != nil {
        return err
    }
    delete(s.data, key)
    return nil
}

func (s *Store) Get(key string) ([]byte, bool) {
    s.mu.RLock()
    defer s.mu.RUnlock()
    v, ok := s.data[key]
    return v, ok
}
```

The ordering in `Set` is the correctness argument, and it is worth saying out loud: **log first, then memory**. If memory were updated first, a reader could observe a value that the crash removes, and the store would have lied.

The one system call that matters

```
func (s *Store) sync() error {
    if err := s.w.Flush(); err != nil {       // bufio -> kernel
        return fmt.Errorf("flushing: %w", err)
    }
    if err := s.log.Sync(); err != nil {      // kernel -> disk
        return fmt.Errorf("syncing: %w", err)
    }
    return nil
}
```

Two steps, and people routinely do only the first.

`Flush` moves bytes out of Go's buffer into the operating system. At that point the data survives your **process** crashing, because the kernel has it.

`Sync` is `fsync`, and it asks the kernel to put the bytes on the physical device. Only after this does the data survive the **machine** losing power.

> **WATCH OUT**
> `fsync` is slow: on spinning disks milliseconds, on an SSD still far slower than anything else in this program. Calling it on every write makes a store that does maybe a few thousand writes a second.

That is the durability dial, and every database exposes it. Postgres calls it `synchronous_commit`, Redis calls it `appendfsync`, and the options are always the same three:

every write	Slowest, loses nothing
every N ms	Fast, loses up to N milliseconds on a power cut
never	Fastest, the operating system decides, can lose a lot

There is no correct answer, only a choice about what you can afford to lose. A store that does not tell you which one it made is the one to distrust.

Compaction

The log grows forever. Setting one key a million times means a million records and a one-entry map.

```
// Compact writes a fresh log containing only the current state, then swaps
// it in atomically.
func (s *Store) Compact() error {
    s.mu.Lock()
    defer s.mu.Unlock()

    tmp := s.path + ".compact"
    f, err := os.OpenFile(tmp, os.O_RDWR|os.O_CREATE|os.O_TRUNC, 0o600)
    if err != nil {
        return fmt.Errorf("creating %s: %w", tmp, err)
    }

    w := bufio.NewWriter(f)
    for k, v := range s.data {
        if _, err := w.Write(encode(opSet, []byte(k), v)); err != nil {
            f.Close()
            os.Remove(tmp)
            return fmt.Errorf("writing compacted record: %w", err)
        }
    }
    if err := w.Flush(); err != nil {
        f.Close()
        os.Remove(tmp)
        return err
    }
    if err := f.Sync(); err != nil {        // durable before the swap
        f.Close()
        os.Remove(tmp)
        return err
    }
```

```
    f.Close()

    // Rename is atomic on the same filesystem: the log is either the old
    // one or the new one, never a half-written mixture.
    if err := os.Rename(tmp, s.path); err != nil {
        os.Remove(tmp)
        return fmt.Errorf("swapping log: %w", err)
    }

    s.log.Close()
    nf, err := os.OpenFile(s.path, os.O_RDWR|os.O_APPEND, 0o600)
    if err != nil {
        return fmt.Errorf("reopening log: %w", err)
    }
    s.log, s.w = nf, bufio.NewWriter(nf)
    return nil
}
```

`os.Rename` on the same filesystem is atomic, which is the trick that makes this safe. Write the new file completely, sync it, then swap. A crash at any point leaves either the old log or the new one, and both are valid.

IF YOU KNOW JAVASCRIPT

Write to a temp file, fsync, rename is **the** durable-replace pattern, and it applies far beyond databases. Any time you overwrite a file that matters, a configuration file or an export, this is the correct way to do it. Writing in place can leave a half-written file that nothing can read.

PROJECT 34: MAKE IT CRASH AND WATCH IT RECOVER

The tests are the chapter. Each proves a property the code was written for.

```
package kv

import (
    "os"
    "path/filepath"
    "testing"
)

func TestSurvivesReopen(t *testing.T) {
    path := filepath.Join(t.TempDir(), "test.log")

    s, err := Open(path)
    if err != nil {
        t.Fatal(err)
    }
```

```
    if err := s.Set("user:1", []byte("Ada")); err != nil {
        t.Fatal(err)
    }
    if err := s.Set("user:2", []byte("Grace")); err != nil {
        t.Fatal(err)
    }
    if err := s.Delete("user:1"); err != nil {
        t.Fatal(err)
    }
    s.Close()

    // Reopen: everything comes from the log.
    s2, err := Open(path)
    if err != nil {
        t.Fatal(err)
    }
    defer s2.Close()

    if _, ok := s2.Get("user:1"); ok {
        t.Error("deleted key came back")
    }
    v, ok := s2.Get("user:2")
    if !ok || string(v) != "Grace" {
        t.Errorf("user:2 = %q %v, want Grace true", v, ok)
    }
}

// The important one: a log cut off mid record must not break the store.
func TestTornTailIsDiscarded(t *testing.T) {
    path := filepath.Join(t.TempDir(), "torn.log")

    s, _ := Open(path)
    s.Set("a", []byte("one"))
    s.Set("b", []byte("two"))
    s.Close()

    // Simulate a crash during the third write by appending a partial
    // record: a header with no body behind it.
    f, _ := os.OpenFile(path, os.O_WRONLY|os.O_APPEND, 0o600)
    f.Write(encode(opSet, []byte("c"), []byte("three"))[:8])
    f.Close()

    s2, err := Open(path)
    if err != nil {
        t.Fatalf("a torn tail must not fail the open: %v", err)
    }
    defer s2.Close()

    if _, ok := s2.Get("c"); ok {
        t.Error("a half written record was replayed")
    }
```

```
    if v, ok := s2.Get("b"); !ok || string(v) != "two" {
        t.Error("records before the tear were lost")
    }

    // And the store must still be writable afterwards.
    if err := s2.Set("d", []byte("four")); err != nil {
        t.Fatalf("writing after recovery: %v", err)
    }
}

func TestCompactionKeepsState(t *testing.T) {
    path := filepath.Join(t.TempDir(), "compact.log")
    s, _ := Open(path)

    for i := 0; i < 1000; i++ {
        s.Set("counter", []byte("value"))
    }
    before, _ := os.Stat(path)

    if err := s.Compact(); err != nil {
        t.Fatal(err)
    }
    after, _ := os.Stat(path)

    if after.Size() >= before.Size() {
        t.Errorf("compaction did not shrink: %d -> %d",
            before.Size(), after.Size())
    }
    if v, ok := s.Get("counter"); !ok || string(v) != "value" {
        t.Error("compaction lost the data")
    }
    s.Close()

    s2, _ := Open(path)
    defer s2.Close()
    if _, ok := s2.Get("counter"); !ok {
        t.Error("compacted log did not replay")
    }
}
```

```
go test -race -v ./...
```

`TestTornTailIsDiscarded` is the one to run first. It writes eight bytes of a record that should have been twenty, which is exactly what a power cut mid write leaves behind, and asserts three things: the open succeeds, the partial record is not replayed, and the store still accepts writes. A store that fails any of those will one day refuse to start, and refusing to start is how a durable store loses data.

YOUR TURN

Three extensions, each teaching something different.

Add a `syncEvery` option so a caller can choose the durability dial, and benchmark `Set` with sync on every write against sync every 100ms. The ratio you measure is the reason databases have that setting.

Add automatic compaction when the log exceeds a multiple of the live data size, and think about what happens if the process dies during compaction. The rename makes it safe; prove it with a test that leaves a stale `.compact` file behind.

Then replace the map with the B-tree from the next chapter and watch the store gain ordered range queries, which is the step from a key-value store to something you could call a database.

ASSIGNMENT

Finish the store, then try to corrupt it.

One. Complete the write-ahead log, the replay and the compaction as written, with tests for each.

Two. Crash it for real. Write a test that starts the store in a subprocess, writes, and kills it with no chance to clean up:

```
go test -run TestCrashRecovery -v
```

Reopen and confirm every acknowledged write is present.

Three. Truncate the log file mid-record, on purpose, and show the store recovering to the last whole record rather than refusing to start.

Four. Benchmark with and without the sync on every write, and put both numbers in `NOTES.md` with one sentence on which you would default to. That trade, durability against throughput, is the one every database makes.

COMMIT AS

```
git commit -m "Key Value Store - Assignment One"
```

52

Build your own on-disk index: pages and a B-tree

The key-value store in chapter fifty one has a limit it does not advertise: on startup it replays the whole log into a map, so the index lives in memory and the startup time grows with the data. At a few hundred megabytes that is fine. At fifty gigabytes it is not a database, it is a long wait.

The answer is an index that lives on disk and is read a page at a time. That is what every real database does, and the structure is nearly always a B-tree.

Why a tree and not a hash

A hash map finds one key in constant time, and that is all it does. A database needs more:

Range scan	`WHERE created_at > ?` reads a contiguous run of keys
Ordering	`ORDER BY` with no sort step
Prefix	`LIKE 'inv-2026%'` is a range
Locality	Nearby keys are on the same page, so one read serves many

A B-tree gives all four because it keeps keys in order, and it is shaped for disks rather than for memory: wide and shallow, so finding a key costs very few reads.

> **IF YOU KNOW JAVASCRIPT**
> The arithmetic is the whole argument. With a 4 KB page and keys around 30 bytes, one page holds roughly a hundred entries.
>
> One hundred to the power of three is a million. So a million keys is **three page reads deep**, and a hundred million is four. A binary tree with the same million keys is twenty levels, which is twenty seeks.
>
> Depth is the number of disk reads, and fanout is how you buy it down.

Pages, the unit of everything

```
const PageSize = 4096

type PageID uint32

type Page struct {
    ID   PageID
    Data [PageSize]byte
}
```

Everything is read and written in whole pages, at offset `int64(id) * PageSize`. The size matches what the operating system and the disk move anyway, so a partial read costs the same as a full one.

A page starts with a header, then entries:

```
offset  bytes  meaning
0       1      node type: 1 leaf, 2 internal
1       2      number of keys
3       4      next leaf (leaves only), 0 for none
7       ...    entries
```

Fixed offsets, little endian, no reflection, no JSON. The point of a page format is that reading it is a few `binary.LittleEndian.Uint16` calls, not a decoder.

Leaves hold data, internal nodes hold directions

This is a B+tree, which is what databases actually use:

```
                [ 50 | 120 ]              internal: separators
               /     |      \
        [10,30,45] [55,90] [130,180]       leaves: keys and values
             |         |         |
             +-------->+-------->+          leaves are linked
```

Two properties follow, and both matter.

All values are in the leaves. An internal node is only directions, so it fits more separators, so the fanout is higher and the tree is shallower.

Leaves are linked. A range scan descends once and then walks sideways, reading consecutive pages without touching the tree again.

Searching

```
func (t *Tree) Get(key []byte) ([]byte, error) {
    pid := t.root
    for {
        node, err := t.read(pid)
        if err != nil {
            return nil, err
        }
        if node.isLeaf() {
            i, found := node.search(key)
            if !found {
                return nil, ErrNotFound
            }
            return node.value(i), nil
        }
        pid = node.child(node.childIndex(key))
    }
}
```

`search` is a binary search **inside the page**, because the keys in a page are sorted:

```
func (n *node) search(key []byte) (int, bool) {
    lo, hi := 0, n.count()
    for lo < hi {
        mid := (lo + hi) / 2
        switch bytes.Compare(n.key(mid), key) {
        case 0:
            return mid, true
        case -1:
            lo = mid + 1
        default:
            hi = mid
        }
    }
    return lo, false
}
```

Note the loop in `Get` rather than recursion. Descending a tree iteratively costs nothing extra and removes any question about stack depth.

Inserting, and the split

Inserting is "find the leaf, put it in". The interesting half is what happens when the leaf is full.

```
leaf is full:
  [10, 30, 45, 55]  + 60
```

```
split:
  [10, 30]  [45, 55, 60]
       \       /
   parent gains separator 45
```

The split propagates: if the parent is now full, it splits too. If the **root** splits, a new root is made above it and **the tree gets one level deeper**. That is the only way a B-tree grows in height, which is why its depth is uniform and why every lookup costs the same.

```go
func (t *Tree) insert(pid PageID, key, val []byte) (
    split *splitResult, err error) {

    node, err := t.read(pid)
    if err != nil {
        return nil, err
    }
    if node.isLeaf() {
        node.put(key, val)
        if node.fits() {
            return nil, t.write(node)
        }
        return t.splitLeaf(node)
    }

    i := node.childIndex(key)
    sp, err := t.insert(node.child(i), key, val)
    if err != nil || sp == nil {
        return nil, err
    }
    node.insertSeparator(i, sp.key, sp.right)
    if node.fits() {
        return nil, t.write(node)
    }
    return t.splitInternal(node)
}
```

WATCH OUT

`fits()` rather than "count < max". Keys and values are variable length, so a page is full when the bytes no longer fit, not when some count is reached.

Writing this as a count works perfectly until a long key arrives, and then corrupts a page by overwriting the one after it. Size the check in bytes and test it with a key that is deliberately enormous.

Deleting, honestly

Textbook B-trees merge underfull nodes and rebalance. It is fiddly, it is where the subtle bugs live, and many real systems do something simpler:

Mark the entry deleted and leave the page alone. Space is reclaimed when the page is rewritten or during a compaction pass.

The cost is pages that are emptier than they could be, which costs reads. The benefit is a much smaller surface for corruption. Start here, and implement merging only when you can measure that it matters.

The free list

Pages become free when a compaction empties them. Without a list of them, the file only grows:

```
type freeList struct {
    ids []PageID
}

func (t *Tree) allocate() PageID {
    if id, ok := t.free.pop(); ok {
        return id
    }
    t.nextPage++
    return t.nextPage - 1
}
```

The free list lives in a page of its own, usually near the start, and it is the second thing to write after the root pointer.

Durability, and the page that was half written

A page write is not atomic. The power goes out, and 4096 bytes are partly old and partly new: a **torn page**. A B-tree with a torn internal node is not a tree with one bad row, it is a tree that cannot be traversed.

Two defences, and real databases pick one:

Write-ahead log	Write the change to a log and `fsync` it, then update the page. Recovery replays the log. This is chapter fifty one's machinery, reused
Copy on write	Never overwrite a live page. Write a new one, then flip the root pointer with one small atomic write

Copy on write is the one to build first here, because it fits the structure:

```
// Every modified page is written to a NEW page id. The parents
// are rewritten to point at it, all the way to the root. Then
// one write updates the header's root pointer, and only that
// write has to be atomic.
```

A header small enough to live in a single sector gets atomicity from the hardware, which is why database file headers are tiny.

> **WATCH OUT**
> `fsync` is the whole of durability, and forgetting it makes everything look fast and work perfectly until the first real power cut.
>
> Order matters too: write the new pages, `fsync`, **then** write the root pointer, `fsync` again. Flipping the root before the pages it points at are durable gives you a root pointing at garbage, which is worse than losing the write.

A buffer pool, because reading the same page is silly

```
type pool struct {
    mu     sync.Mutex
    pages  map[PageID]*Page
    order  *list.List            // LRU
    max    int
}
```

The top two levels of the tree are read on every single lookup, so caching turns three reads into one. A few thousand pages of cache covers the internal nodes of a surprisingly large tree, and that is why databases want memory.

Testing a data structure you cannot see

Three techniques, in increasing order of how much they find.

Differential testing against a sorted map. Any sequence of operations must produce the same answers as `map` plus a sort:

```
func TestMatchesReference(t *testing.T) {
    tree := openTemp(t)
    ref := map[string]string{}

    for i := 0; i < 50_000; i++ {
        k, v := randKey(), randValue()
        tree.Put([]byte(k), []byte(v))
        ref[k] = v
    }
    for k, want := range ref {
```

```
        got, err := tree.Get([]byte(k))
        if err != nil || string(got) != want {
            t.Fatalf("Get(%q) = %q, %v; want %q", k, got, err, want)
        }
    }
}
```

An invariant check. Walk the tree and assert: keys sorted within each page, every key in a child within its parent's separators, all leaves at the same depth, pages at least half full except the root. Run it after every thousand inserts in a test.

A crash test. Kill the process mid-write, reopen, and check the invariants plus every acknowledged key. This is the one that finds the `fsync` ordering bug, and nothing else will.

PROJECT 48: THE INDEX, MEASURED

In `part6-from-scratch/52-btree`, build it and then find out what you built.

One: pages. A pager with `read`, `write`, `allocate`, a free list and a header page. Test that reopening finds the root.

Two: the tree. `Get`, `Put`, `Delete` with tombstones, splits for leaves and internal nodes, and a root split that deepens the tree.

Three: the invariant checker, run after every thousand operations in a test.

Four: the differential test against a reference map, with fifty thousand random keys.

Five: ranges. `Scan(from, to)` using the leaf links. Prove with a timing that scanning ten thousand consecutive keys is far cheaper than ten thousand `Get` calls.

Six: the depth. Insert one million keys and report the tree height and the fanout you achieved. Compare it with the arithmetic at the top of this chapter.

Seven: durability. Copy on write with the root flip last. Then kill the process in a loop during writes, reopen each time, and assert the invariants and every acknowledged key.

Eight: the cache. Add an LRU buffer pool and measure lookups per second with it at 0, 100 and 10000 pages. Put the three numbers in `RESULTS.md`.

YOUR TURN

Open your tree file in a hex viewer after inserting a few keys.

```
xxd -l 256 index.db
```

Finding your header, your node type byte and your first key in the bytes is the moment a file format stops being abstract. Every database file you will ever meet is this, with more fields.

ASSIGNMENT

Build the index and report it like an engineer rather than a tutorial.

One. The pager, the B+tree with splits, range scans over linked leaves, and the free list.

Two. The invariant checker and the differential test, both in CI.

Three. `RESULTS.md` with: tree height at 1 thousand, 100 thousand and 1 million keys; lookups per second cold and warm; the scan comparison; and the file size against the raw data size, which is your space amplification.

Four. The crash test, run at least fifty times, with the number of runs and any failure you found.

Five. One honest paragraph: what does SQLite do that your index does not? Name three things. Knowing the list is more valuable than closing it.

COMMIT AS

```
git commit -m "B-Tree Index - Assignment One"
```

53

Build your own Redis: RESP over raw TCP

Everything you have written so far has spoken HTTP, which is a protocol somebody else parsed for you. This chapter goes one layer down: a TCP listener, a wire format you implement yourself, and a server that the real `redis-cli` can connect to.

That last part is what makes this the best protocol exercise there is. You are not testing against your own client. You are testing against a program written by people who have never heard of you, and when it works, your implementation is correct in a way no unit test proves.

The protocol, in one page

RESP is deliberately simple. The first byte says the type:

```
+OK\r\n                    simple string
-ERR unknown command\r\n   error
:42\r\n                    integer
$5\r\nhello\r\n            bulk string: length, then bytes
$-1\r\n                    null bulk string
*2\r\n$4\r\nECHO\r\n$2\r\nhi\r\n   array of two bulk strings
```

A client sends commands as **arrays of bulk strings**. `SET name Ada` arrives as:

```
*3\r\n$3\r\nSET\r\n$4\r\nname\r\n$3\r\nAda\r\n
```

That is the whole format. Five types, one of which is a container.

> **IF YOU KNOW JAVASCRIPT**
> Compare this with JSON, which you parsed in chapter fifty. RESP carries the **length** of every string before its bytes, so the reader never scans for a delimiter and never has to escape anything.
>
> That is the difference between a format designed for humans and one designed for throughput, and it is why `$5\r\nhello\r\n` looks redundant and is not: a string containing `\r\n` needs no special handling at all.

The server loop

```
func Serve(ln net.Listener, db *Store) error {
    for {
        conn, err := ln.Accept()
        if err != nil {
            return err
        }
        go handle(conn, db)
    }
}

func handle(conn net.Conn, db *Store) {
    defer conn.Close()
    r := bufio.NewReader(conn)
    w := bufio.NewWriter(conn)

    for {
        args, err := readCommand(r)
        if err != nil {
            if err != io.EOF {
                writeError(w, "ERR "+err.Error())
                w.Flush()
            }
            return
        }
        execute(db, args, w)

        // Flush only when nothing more is buffered: that is what
        // makes pipelining fast rather than chatty.
        if r.Buffered() == 0 {
            w.Flush()
        }
    }
}
```

Goroutine per connection, which in Go is the obvious design and in most languages is not. Ten thousand connections is ten thousand goroutines and a few hundred megabytes, which chapter thirteen promised and this is where you cash it in.

> **WATCH OUT**
> The flush condition is the single most important line for performance.
>
> Flushing after every command turns a pipeline of a thousand commands into a thousand small writes and a thousand round trips. Flushing only when the read buffer is empty lets a client that sent a thousand commands in one packet receive one packet back.

Measured with `redis-benchmark -P 100`, the difference is roughly an order of magnitude.

Reading a command

```
func readCommand(r *bufio.Reader) ([]string, error) {
    line, err := readLine(r)
    if err != nil {
        return nil, err
    }
    if len(line) == 0 {
        return nil, errors.New("protocol error: empty line")
    }
    if line[0] != '*' {
        // Inline command, what you get from telnet.
        return strings.Fields(string(line)), nil
    }

    n, err := strconv.Atoi(string(line[1:]))
    if err != nil || n < 0 {
        return nil, errors.New("protocol error: bad multibulk")
    }
    args := make([]string, 0, n)
    for i := 0; i < n; i++ {
        s, err := readBulk(r)
        if err != nil {
            return nil, err
        }
        args = append(args, s)
    }
    return args, nil
}

func readBulk(r *bufio.Reader) (string, error) {
    line, err := readLine(r)
    if err != nil {
        return "", err
    }
    if len(line) == 0 || line[0] != '$' {
        return "", errors.New("protocol error: expected $")
    }
    n, err := strconv.Atoi(string(line[1:]))
    if err != nil || n < 0 || n > maxBulk {
        return "", errors.New("protocol error: bad bulk length")
    }
    buf := make([]byte, n+2)          // payload plus CRLF
    if _, err := io.ReadFull(r, buf); err != nil {
        return "", err
    }
```

```
    return string(buf[:n]), nil
}
```

`io.ReadFull` is the heart of it, and it is the line that separates a working server from a mysteriously broken one.

> **WATCH OUT**
> **TCP is a stream, not a sequence of messages.** One `Read` can return half a command, or two commands, or one and a half. Nothing preserves the boundaries the sender used. So a bulk string of five bytes is read with `io.ReadFull`, which loops until it has five, rather than with one `Read` that might return three. Every protocol bug that only appears over a real network and never over localhost is this, because localhost rarely splits packets and the internet does it constantly.

Note `n > maxBulk` as well. A client claiming a 4 GB bulk string must not cause a 4 GB allocation, which is chapter forty two's bomb wearing a different hat.

The store, and the lock

```
type Store struct {
    mu   sync.RWMutex
    data map[string]entry
}

type entry struct {
    value   string
    expires time.Time         // zero means never
}

func (s *Store) Get(key string) (string, bool) {
    s.mu.RLock()
    e, ok := s.data[key]
    s.mu.RUnlock()

    if !ok {
        return "", false
    }
    if !e.expires.IsZero() && time.Now().After(e.expires) {
        s.Delete(key)              // lazy expiry
        return "", false
    }
    return e.value, true
}
```

Expiry is **lazy plus active**, which is what Redis itself does: a key is dropped when somebody touches it after its deadline, and a background sweep samples random keys

to catch the ones nobody asks for. Without the sweep, keys nobody reads live forever and the memory never comes back.

> **WATCH OUT**
> The real Redis is single threaded for command execution, so it needs no lock at all. Your Go version has a goroutine per connection and therefore does.
>
> That is a genuine trade: the lock costs you contention on a hot key, and Redis's single thread costs it one core. Before deciding yours is better, benchmark with `redis-benchmark` and look at the numbers, because a global `RWMutex` under write-heavy load is less impressive than it sounds. Sharding the map by key hash is the usual next step.

Commands, as a table

```
type command struct {
    arity   int               // -N means "at least N"
    handler func(*Store, []string, *bufio.Writer)
}

var commands = map[string]command{
    "ping":   {1, cmdPing},
    "echo":   {2, cmdEcho},
    "set":    {-3, cmdSet},
    "get":    {2, cmdGet},
    "del":    {-2, cmdDel},
    "exists": {-2, cmdExists},
    "incr":   {2, cmdIncr},
    "expire": {3, cmdExpire},
    "ttl":    {2, cmdTTL},
    "keys":   {2, cmdKeys},
    "command": {-1, cmdCommand},
}
```

Arity checked once, in `execute`, rather than in every handler. And `command` is in the list for a practical reason: `redis-cli` sends `COMMAND DOCS` on connect, and a server that errors on it looks broken before you have typed anything.

Incrementing, which is where atomicity shows up

```
func cmdIncr(s *Store, args []string, w *bufio.Writer) {
    s.mu.Lock()
    defer s.mu.Unlock()

    e := s.data[args[1]]
    n, err := strconv.ParseInt(e.value, 10, 64)
```

```
    if e.value != "" && err != nil {
        writeError(w, "ERR value is not an integer or out of range")
        return
    }
    n++
    e.value = strconv.FormatInt(n, 10)
    s.data[args[1]] = e
    writeInteger(w, n)
}
```

The read, the increment and the write are inside **one** lock. Doing `Get` then `Set` from the handler would be two operations with a window between them, and two clients incrementing at once would lose a count. This is chapter fifteen's lesson in its smallest possible form, and it is exactly why `INCR` exists as a command rather than being left to the client.

Persistence, which you have already built

An append-only file: every write command is appended to a log, and on startup the log is replayed. That is chapter fifty one's write-ahead log, unchanged.

The questions are the same too. `fsync` on every write is durable and slow; once a second is fast and loses up to a second. Redis makes that a configuration option, and so should you, with the trade written down where the person choosing can see it.

Testing against the real thing

```
go run . &                      # your server on :6380

redis-cli -p 6380 ping
redis-cli -p 6380 set name Ada
redis-cli -p 6380 get name
redis-cli -p 6380 incr counter

redis-benchmark -p 6380 -t set,get -n 100000 -q
redis-benchmark -p 6380 -t set,get -n 100000 -P 100 -q
```

The benchmark with and without pipelining is the measurement that tells you whether your flush logic is right.

> **IF YOU KNOW JAVASCRIPT**
> This is the most satisfying validation in the book. `redis-cli` knows nothing about your code, implements the protocol strictly, and will tell you immediately when a reply type is wrong.

> Interoperating with somebody else's client is a different standard of correctness from passing your own tests, and it is the standard that matters when you write a protocol anybody else will speak.

What you are not building

Be clear about the gap, because "I built Redis" is a sentence that needs qualifying:

Data types	Real Redis has lists, sets, sorted sets, hashes, streams. This has strings
Pub/sub	A different connection mode, where the server pushes
Scripting	Lua, executed atomically on the server
Replication	A whole protocol of its own
Cluster	Key hashing, slots, redirects

Strings plus expiry is perhaps five percent of Redis and a hundred percent of the protocol work. That is the right trade for a chapter, and the remaining list is a very good source of weekend projects.

PROJECT 49: A SERVER REDIS-CLI CANNOT TELL APART

In `part6-from-scratch/53-redis`, build it.

One: the protocol. Reader and writer for all five RESP types, with `io.ReadFull` for bulk payloads and a cap on declared lengths.

Two: the commands. `PING`, `ECHO`, `SET` with `EX`, `GET`, `DEL`, `EXISTS`, `INCR`, `DECR`, `EXPIRE`, `TTL`, `KEYS`, `COMMAND`.

Three: expiry. Lazy on read plus an active sampler. Prove with a test that a key nobody reads is eventually reclaimed.

Four: the real client. Every command above verified through `redis-cli`, pasted into `SESSION.md`.

Five: the benchmark. `redis-benchmark` with and without `-P 100`, both recorded. If pipelining does not help by a large factor, your flush logic is wrong and that is the finding.

Six: concurrency. Run the benchmark with 50 clients under `-race`. Then shard the map into sixteen and measure again.

Seven: persistence. An append-only log with a configurable sync policy. Kill the server mid-load, restart, and report what survived under each policy.

Eight: one data type. Add lists with `LPUSH`, `RPUSH`, `LRANGE` and `LLEN`. Then look up how Redis actually stores a list and write a paragraph on why a Go slice is not what it uses.

YOUR TURN

Connect to your server with `telnet localhost 6380` and type `PING` followed by Enter.

That is the inline command path, and seeing a raw protocol answer a human typing into a socket is the moment networking stops being abstract. Then try `SET a b` and `GET a` the same way.

ASSIGNMENT

Build the server and prove it against tools you did not write.

One. The full implementation with the command table, expiry and persistence.

Two. `SESSION.md`: a real `redis-cli` session exercising every command, including the error cases (wrong arity, `INCR` on a non-number).

Three. `BENCH.md` with four numbers: get and set, pipelined and not. Plus the same four from a real Redis on your machine if you have one, and one sentence on the difference.

Four. The framing test: a client that sends a command **one byte at a time** with a delay between bytes. Your server must handle it. If it does not, you found the bug this chapter warned about.

Five. One paragraph on what you would have to build next to run this in production, in the order you would build it.

COMMIT AS

```
git commit -m "RESP Server - Assignment One"
```

54

Build your own HTTP server, from a TCP socket

Chapter twenty one used `net/http`, which hides the protocol so completely that most web developers have never seen a request line. This chapter removes the hiding: a TCP listener, HTTP/1.1 parsed by hand, keep-alive, chunked bodies, and the timeouts without which a server can be held open by one slow client.

The payoff is not a better server. It is that the next time you read about request smuggling, a reverse proxy misbehaving or a mysterious hanging connection, you will know exactly which bytes are involved.

What a request actually is

```
POST /invoices?draft=1 HTTP/1.1\r\n
Host: api.example.com\r\n
Content-Type: application/json\r\n
Content-Length: 27\r\n
\r\n
{"client":1,"total":2500}
```

A request line, headers, a blank line, and optionally a body. The response has the same shape with a status line instead:

```
HTTP/1.1 201 Created\r\n
Content-Type: application/json\r\n
Content-Length: 42\r\n
\r\n
{"data":{"id":7},"message":"created"}
```

That is HTTP/1.1. Everything else is rules about how to know where the body ends and whether the connection stays open.

> **IF YOU KNOW JAVASCRIPT**

This is why `curl -v` is such a good debugging tool: it prints exactly these lines. Once you have written the parser, `curl -v` stops being noise and becomes the clearest view of what your framework is actually sending.

The accept loop, with the timeouts that matter

```
func (s *Server) Serve(ln net.Listener) error {
    for {
        conn, err := ln.Accept()
        if err != nil {
            return err
        }
        go s.handleConn(conn)
    }
}

func (s *Server) handleConn(conn net.Conn) {
    defer conn.Close()
    r := bufio.NewReaderSize(conn, 8<<10)
    w := bufio.NewWriter(conn)

    for {
        conn.SetReadDeadline(time.Now().Add(s.ReadHeaderTimeout))

        req, err := readRequest(r)
        if err != nil {
            writeError(w, err)
            w.Flush()
            return
        }

        conn.SetReadDeadline(time.Now().Add(s.ReadTimeout))
        conn.SetWriteDeadline(time.Now().Add(s.WriteTimeout))

        keepAlive := s.serve(w, req)
        w.Flush()

        if !keepAlive {
            return
        }
        conn.SetReadDeadline(time.Now().Add(s.IdleTimeout))
    }
}
```

WATCH OUT

`ReadHeaderTimeout` is separate from `ReadTimeout`, and the reason has a name: **Slowloris**.

The attack is to open a thousand connections and send one header byte every twenty seconds. Each connection is technically active, so a single overall timeout never fires, and the server runs out of connections while using almost no bandwidth.

A deadline on the **headers** specifically is the defence, and it is why `net/http` grew that field. A server with no header deadline is held open by a laptop.

Parsing the request

```
func readRequest(r *bufio.Reader) (*Request, error) {
    line, err := readLine(r, maxLine)
    if err != nil {
        return nil, err
    }
    method, target, version, ok := splitRequestLine(line)
    if !ok {
        return nil, badRequest("malformed request line")
    }
    if version != "HTTP/1.1" && version != "HTTP/1.0" {
        return nil, &httpError{505, "version not supported"}
    }

    headers, err := readHeaders(r)
    if err != nil {
        return nil, err
    }
    body, err := bodyReader(r, headers)
    if err != nil {
        return nil, err
    }
    return &Request{
        Method: method, Target: target,
        Version: version, Headers: headers, Body: body,
    }, nil
}
```

Headers are a loop until the blank line, with two defences:

```
func readHeaders(r *bufio.Reader) (Header, error) {
    h := Header{}
    for i := 0; ; i++ {
        if i > maxHeaders {            // 100 is plenty
            return nil, badRequest("too many headers")
        }
        line, err := readLine(r, maxLine)   // 8 KB each
        if err != nil {
            return nil, err
```

```
        }
        if len(line) == 0 {
            return h, nil
        }
        name, value, ok := strings.Cut(line, ":")
        if !ok {
            return nil, badRequest("malformed header")
        }
        h.Add(textproto.CanonicalMIMEHeaderKey(name),
            strings.TrimSpace(value))
    }
}
```

Header names are case insensitive, so they are canonicalised once on the way in. Doing it at every lookup instead is both slower and a source of bugs where `Content-Type` and `content-type` are two different entries.

The limits are not politeness. Without them, a client sending headers forever is an out-of-memory error with no request ever completing.

Where does the body end

This is the whole of HTTP/1.1's complexity, and there are exactly three answers:

`Content-Length: n`	Read exactly n bytes
`Transfer-Encoding: chunked`	Read chunks until a zero-length one
Neither	There is no body. A GET, usually

Chunked is the one worth implementing, because it is how a server streams something whose size it does not know yet:

```
7\r\n
Mozilla\r\n
9\r\n
Developer\r\n
0\r\n
\r\n
```

Each chunk is a hexadecimal length, CRLF, that many bytes, CRLF. A zero-length chunk ends the body.

```
func bodyReader(r *bufio.Reader, h Header) (io.Reader, error) {
    te := strings.ToLower(h.Get("Transfer-Encoding"))
    cl := h.Get("Content-Length")

    if te == "chunked" && cl != "" {
```

```
        // Both present: refuse. See below.
        return nil, badRequest("both Transfer-Encoding and " +
            "Content-Length")
    }
    switch {
    case te == "chunked":
        return newChunkedReader(r), nil
    case cl != "":
        n, err := strconv.ParseInt(cl, 10, 64)
        if err != nil || n < 0 {
            return nil, badRequest("bad Content-Length")
        }
        return io.LimitReader(r, n), nil
    default:
        return emptyReader{}, nil
    }
}
```

WATCH OUT

That refusal is not pedantry. A message with **both** framing headers is the basis of **HTTP request smuggling**, and it is one of the highest-impact web vulnerabilities there is.

The attack works when two servers disagree: a front-end proxy uses `Content-Length` and your back-end uses `Transfer-Encoding`, or the reverse. The bytes one of them considers to be the end of request A, the other considers the start of request B. An attacker appends a request that your server attributes to the **next** user's connection.

The defence is to refuse the ambiguity rather than resolve it. If both headers are present, the message is malformed: 400, and close the connection.

Keep-alive, and the body you forgot to read

HTTP/1.1 connections are persistent by default. After responding, you loop and read the next request from the same socket.

```
func keepAlive(req *Request) bool {
    conn := strings.ToLower(req.Headers.Get("Connection"))
    if req.Version == "HTTP/1.0" {
        return conn == "keep-alive"
    }
    return conn != "close"
}
```

And the trap:

```
// Before reading the next request, drain whatever the handler
// did not read of this one. Otherwise the leftover body bytes
// are parsed as the next request line, which produces a
// baffling "malformed request line: {"client":1..."
io.Copy(io.Discard, req.Body)
```

WATCH OUT

This bug is unforgettable once seen. A handler returns 400 early without reading the body, the next loop iteration reads the unread JSON as a request line, and the error message contains fragments of the previous request.

Worse, it is intermittent: it only happens when a handler returns early, so it shows up under error conditions, which is when you are least able to reason about it.

Drain or close. `net/http` drains a limited amount and closes beyond it, which is a good policy to copy.

Writing the response

```
func (w *responseWriter) WriteHeader(status int) {
    fmt.Fprintf(w.buf, "HTTP/1.1 %d %s\r\n",
        status, statusText(status))
    w.header.Set("Date", time.Now().UTC().
        Format(http.TimeFormat))

    if _, ok := w.header["Content-Length"]; !ok {
        w.header.Set("Transfer-Encoding", "chunked")
        w.chunked = true
    }
    for name, values := range w.header {
        for _, v := range values {
            fmt.Fprintf(w.buf, "%s: %s\r\n", name, v)
        }
    }
    w.buf.WriteString("\r\n")
}
```

If the handler knows the length, send `Content-Length`. If it does not, switch to chunked. That is the decision `net/http` makes for you: it buffers a little, and if the handler finishes within the buffer it sets a length, otherwise it goes chunked.

Two responses have no body at all and must not have a length: 204 and 304. A server that sends `Content-Length: 0` with a 304 confuses caches in ways that take a day to find.

100-continue, the handshake nobody knows about

```
Expect: 100-continue
```

A client about to upload something large asks first. The server answers either:

```
HTTP/1.1 100 Continue\r\n\r\n
```

and the client sends the body, or it answers 413 or 401 immediately and the client never uploads a gigabyte that was going to be rejected anyway.

`curl` sends this automatically for large bodies. A server that ignores it makes those clients wait for a timeout before uploading, which presents as "uploads are slow from curl and fine from the browser".

Routing, which you have already written

The router from chapter forty eight drops straight in: match the method and the path segments, collect parameters, call the handler. Nothing about routing changes because you parsed the request yourself.

What you are not building

TLS	`crypto/tls` wraps your listener in about ten lines. Not hard, just not HTTP
HTTP/2	Binary framing, HPACK header compression, multiplexed streams. A different project of a similar size
HTTP/3	QUIC, over UDP. A much bigger project
Compression	`Accept-Encoding` and a gzip writer. A good exercise

Testing it

Against real clients, as in chapter fifty three:

```
curl -v http://localhost:8080/health
curl -v -X POST localhost:8080/echo -d '{"a":1}'
curl -v --http1.1 -H "Transfer-Encoding: chunked" \
  -T big.json localhost:8080/upload
```

And against raw bytes, which is where the interesting failures live:

```
printf 'GET / HTTP/1.1\r\nHost: x\r\n\r\n' | nc localhost 8080

# Two requests in one packet: pipelining
```

```
R='GET /a HTTP/1.1\r\nHost: x\r\n\r\n'
printf "$R$R" | nc localhost 8080
```

PROJECT 50: THE SERVER, AND THE ATTACKS IT SURVIVES

In `part6-from-scratch/54-http`, build it.

One: the parser. Request line, headers with limits and canonical names, and all three body framings including chunked.

Two: the writer. Status line, `Date`, `Content-Length` or chunked, and correct handling of 204 and 304.

Three: keep-alive. Persistent connections, the drain, and `Connection: close` honoured in both directions.

Four: the timeouts. Separate header, read, write and idle deadlines, each configurable.

Five: Slowloris. Write a client that opens 200 connections and sends one header byte every two seconds. With the header timeout off, show the server stop accepting. Turn it on and show it survive. Record both.

Six: smuggling. Send a request with both `Content-Length` and `Transfer-Encoding` and confirm a 400. Then read the two smuggling variants and write a paragraph on which one your refusal prevents.

Seven: pipelining. Two requests in one packet, two responses in order.

Eight: interoperability. Point a real HTTP client at it, Go's own `http.Client` and `curl`, and make both work. Then serve a file and open it in a browser.

YOUR TURN

Run your server and the standard library's side by side, and compare the exact bytes with `curl -v` for the same handler.

Every difference is a decision `net/http` made that you have not. Some will be headers you forgot; some will be behaviour you now understand well enough to disagree with, which is the point of the chapter.

ASSIGNMENT

Build the server, then attack it.

One. The full implementation with chunked bodies, keep-alive, pipelining and the four timeouts.

Two. `ATTACKS.md`: Slowloris with and without the header timeout and the connection counts; the smuggling refusal with the raw bytes you sent; and one oversized-header attempt.

Three. The drain bug, demonstrated deliberately: remove the drain, send a POST whose handler returns early, and paste the baffling error from the next request on that connection.

Four. Interoperability evidence: `curl -v` output, a Go `http.Client` test, and a screenshot of a browser loading a page from it.

Five. One paragraph on what you would need before putting this on the internet, honestly ordered. TLS is on the list, but it is not first.

COMMIT AS

```
git commit -m "HTTP From TCP - Assignment One"
```

55

Build your own Git

Git is the tool you use most and understand least. Its interface is famously awkward and its data model is famously elegant, and the gap between those two facts is why most people learn commands rather than concepts.

The model is small enough to implement in an afternoon: a content-addressed store with four object types. By the end of this chapter your program will write objects that the real `git` command reads, and `git log` will show commits your code created.

Everything is an object, named by its hash

```
blob    a file's contents. No name, no permissions, just bytes
tree    a directory: names, modes, and the hashes they point at
commit  a tree, parents, author, message
tag     an annotated tag
```

An object is stored as:

```
<type> <size>\0<contents>
```

compressed with zlib, and its name is the SHA-1 of **that whole thing**, uncompressed, in hexadecimal. The path is the first two hex characters as a directory and the rest as the filename:

```
.git/objects/e6/9de29bb2d1d6434b8b29ae775ad8c2e48c5391
```

```
func hashObject(typ string, content []byte) (string, []byte) {
    header := fmt.Sprintf("%s %d\x00", typ, len(content))
    full := append([]byte(header), content...)

    sum := sha1.Sum(full)
    return hex.EncodeToString(sum[:]), full
}
```

Fifteen lines, and that function is the whole of Git's identity model.

> **IF YOU KNOW JAVASCRIPT**
> "Content addressed" means the name **is** the content, which has three consequences that explain most of Git's behaviour.
>
> **Identical files are stored once**, however many directories or branches contain them.
>
> **Nothing can be modified.** Changing a file produces a different hash, so it is a new object. The old one is still there.
>
> **A commit's hash covers its entire history**, because it names its tree and its parent, which name theirs. That is why rewriting one old commit changes the hash of every commit after it, and why a force push is a different history rather than an edited one.

init, and the directory

```
func cmdInit(dir string) error {
    for _, d := range []string{
        "objects", "refs/heads", "refs/tags",
    } {
        if err := os.MkdirAll(filepath.Join(dir, ".git", d),
            0o755); err != nil {
            return err
        }
    }
    return os.WriteFile(filepath.Join(dir, ".git", "HEAD"),
        []byte("ref: refs/heads/main\n"), 0o644)
}
```

That is a valid repository. `git status` will run in it.

`HEAD` is a **symbolic reference**: a file containing the name of another reference. That one indirection is the entire branch mechanism. Switching branches rewrites that line; a detached HEAD is the same file containing a hash instead.

Writing and reading objects

```
func writeObject(root, typ string, content []byte) (string, error) {
    sum, full := hashObject(typ, content)
    path := objectPath(root, sum)

    if _, err := os.Stat(path); err == nil {
        return sum, nil          // already have it, by definition
    }
    if err := os.MkdirAll(filepath.Dir(path), 0o755); err != nil {
        return "", err
    }
```

```
    var buf bytes.Buffer
    zw := zlib.NewWriter(&buf)
    if _, err := zw.Write(full); err != nil {
        return "", err
    }
    if err := zw.Close(); err != nil {
        return "", err
    }
    return sum, os.WriteFile(path, buf.Bytes(), 0o444)
}
```

Two details that are not accidents. **If the file exists, you already have the content**, because the name is the content, so there is nothing to compare. And the mode is `0444`, read-only: objects are immutable, and the filesystem is told so.

Trees, and the format that catches everybody

A tree is a sequence of entries, with no separators and no JSON:

```
<mode> <name>\0<20 raw bytes of sha>
```

```
func writeTree(root, dir string) (string, error) {
    entries, err := os.ReadDir(dir)
    if err != nil {
        return "", err
    }
    var buf bytes.Buffer

    for _, e := range sortTreeEntries(entries) {
        if e.Name() == ".git" {
            continue
        }
        var mode, sum string

        if e.IsDir() {
            mode = "40000"
            sum, err = writeTree(root, filepath.Join(dir, e.Name()))
        } else {
            mode = fileMode(e)          // 100644 or 100755
            data, rerr := os.ReadFile(filepath.Join(dir, e.Name()))
            if rerr != nil {
                return "", rerr
            }
            sum, err = writeObject(root, "blob", data)
        }
        if err != nil {
            return "", err
        }
```

```
        raw, _ := hex.DecodeString(sum)
        fmt.Fprintf(&buf, "%s %s\x00", mode, e.Name())
        buf.Write(raw)
    }
    return writeObject(root, "tree", buf.Bytes())
}
```

The modes are a short list: `100644` a file, `100755` an executable, `40000` a directory, `120000` a symlink, `160000` a submodule. They look like Unix permissions and are not: Git stores only whether a file is executable.

> **WATCH OUT**
> Two things here will produce objects real Git rejects, and both are silent.
>
> **The hash is twenty raw bytes, not forty hex characters.** It is the one place in the format where the hash is not text.
>
> **Entries must be sorted**, and the sort is not plain byte order on the name: a directory sorts as though its name ended with `/`. So `lib.rs` comes before `lib/`, which is the opposite of what a naive sort gives.
>
> Get either wrong and your objects hash differently from Git's for the same content, which means your repository and a real one disagree about history that is identical. Check with `git cat-file -p` early rather than after building three more layers.

Commits

```
func commitTree(root, tree string, parents []string,
    author string, when time.Time, msg string) (string, error) {

    var b strings.Builder
    fmt.Fprintf(&b, "tree %s\n", tree)
    for _, p := range parents {
        fmt.Fprintf(&b, "parent %s\n", p)
    }
    stamp := fmt.Sprintf("%d %s", when.Unix(), when.Format("-0700"))
    fmt.Fprintf(&b, "author %s %s\n", author, stamp)
    fmt.Fprintf(&b, "committer %s %s\n", author, stamp)
    fmt.Fprintf(&b, "\n%s\n", msg)

    return writeObject(root, "commit", []byte(b.String()))
}
```

Plain text, which is why `git cat-file -p HEAD` is readable.

Zero parents is the first commit. One parent is ordinary. **Two or more parents is a merge**, and that is the entire representation of merging: a commit that points at both histories. There is no other record of a merge anywhere in Git.

References, and what a branch is

```
func updateRef(root, ref, sum string) error {
    path := filepath.Join(root, ".git", ref)
    if err := os.MkdirAll(filepath.Dir(path), 0o755); err != nil {
        return err
    }
    return os.WriteFile(path, []byte(sum+"\n"), 0o644)
}
```

A branch is a file containing forty one bytes. That is the whole implementation.

It explains things that otherwise seem arbitrary. Creating a branch is instant because it writes one small file. Deleting a branch deletes a file and **not a single commit**, which is why the commits are still there and `git reflog` can find them. Moving a branch is overwriting those forty one bytes, which is what `reset --hard` does.

Walking the log

```
func cmdLog(root, start string) error {
    for sum := start; sum != ""; {
        obj, err := readObject(root, sum)
        if err != nil {
            return err
        }
        c := parseCommit(obj.content)

        fmt.Printf("commit %s\n%s\n\n    %s\n\n",
            sum, c.Author, c.Message)

        if len(c.Parents) == 0 {
            return nil
        }
        sum = c.Parents[0]       // first parent only
    }
    return nil
}
```

`git log` with no flags does exactly this: follow the first parent. The difference between that and `--graph` is whether you also walk the other parents of merges, which needs a queue ordered by commit time and is where log's real complexity lives.

The index, and why it exists

The staging area is a file, `.git/index`, holding the paths and hashes that will go into the next tree, plus the stat information Git uses to decide whether a file changed without reading it.

That last part is the real purpose. `git status` in a repository with a hundred thousand files is fast because it compares size and modification time from the index rather than hashing every file.

For this project, a simple text index is enough:

```
100644 e69de29bb2d1d6434b8b29ae775ad8c2e48c5391 src/main.go
```

Real Git's index is binary and versioned. Replacing it is a chapter of its own, and it is the right thing to leave out.

What you are not building

Packfiles	Thousands of objects delta-compressed into one file. This is how a repository stays small, and it is the biggest omission here
The network	`git fetch` and `git push` speak their own protocol, with negotiation about which objects the other side lacks
Merging	Creating a merge commit is easy. Computing the merged **content** is a three-way diff algorithm
Rebase	Replaying commits onto a new base, which is merge plus bookkeeping

Loose objects are correct and are what Git itself writes before `gc` runs. Your repository will simply be larger than it needs to be, which for a learning project is not a cost at all.

> **WATCH OUT**
> SHA-1 is broken for collision resistance, and Git still uses it.
>
> That is survivable because Git is not using the hash as a security control against a motivated attacker with your repository, and because Git adds a collision detection step on top. The project has a SHA-256 mode and a long transition plan, because changing the hash changes every object name in every repository that exists.
>
> The lesson worth taking: **an identifier is a decision you cannot reverse**. If you build a content-addressed anything today, use SHA-256, and store the algorithm next to the hash so the next change is possible.

> **PROJECT 51: A REPOSITORY REAL GIT CAN READ**
> In `part6-from-scratch/55-git`, build it, and validate against the real thing at every step.

One: `init` and `hash-object`. Hash a file with your tool and with real Git:

```
echo "hello" > a.txt
./mygit hash-object -w a.txt
git hash-object a.txt
```

The two hashes must be identical. If they are not, stop here: your header format is wrong and nothing later will work.

Two: `cat-file`. Read any object back, printing its type, size and contents. Check against `git cat-file -p`.

Three: `write-tree`. Build a tree from a directory. Then verify with real Git:

```
./mygit write-tree
git cat-file -p <the hash it printed>
```

Real Git must be able to print your tree. This is where the sorting rule and the raw-bytes rule get caught.

Four: `commit-tree` and `log`. Make a commit, update `refs/heads/main`, and then run **real** `git log`. Your commit must appear with its message and author.

Five: `status`. Compare the working directory against the index and the HEAD tree. Report added, modified and deleted files.

Six: `checkout`. Restore the working directory from a tree. Then make a commit, change files, and check out the earlier commit with your own tool.

Seven: the proof. Create a repository entirely with your tool, then run `git fsck`. It checks every object's hash and every reference. A clean run is the strongest statement you can make about this project.

YOUR TURN

In any repository you have, run:

```
git cat-file -p HEAD
git cat-file -p HEAD^{tree}
git cat-file -p HEAD:README.md
```

Three commands, and you have walked from a commit to a tree to a file's contents by hand. Everything Git does is built on that walk, and after this chapter you can see it.

ASSIGNMENT

Build the tool and prove interoperability rather than claiming it.

One. `init`, `hash-object`, `cat-file`, `write-tree`, `commit-tree`, `log`, `status`, `checkout`.

Two. `INTEROP.md` with paired output: your hash next to `git hash-object`, your tree printed by `git cat-file -p`, and your commit shown by real `git log`.

Three. A clean `git fsck` on a repository your tool created, with the output pasted.

Four. One deliberate break: sort the tree entries naively, show the hash differing from Git's for the same directory, then fix it and show them matching.

Five. A paragraph explaining to a colleague, in plain language, why rewriting an old commit changes every commit after it. If you can write that from the data model rather than from memory, you have got the chapter.

COMMIT AS

```
git commit -m "Build Git - Assignment One"
```

56

Build your own reverse proxy and load balancer

Every production system has one of these in front of it, and most developers treat it as a configuration file somebody else wrote. A proxy is a few hundred lines, and writing one explains a list of things that are otherwise folklore: why `X-Forwarded-For` can lie, why some headers must be stripped, why a retry can corrupt data, and why your streaming endpoint buffers when it goes through nginx.

What a reverse proxy does

```
client  →  proxy  →  one of several backends
                ←  response
```

Four jobs, and only the first is obvious:

Distribute	Spread requests over healthy backends
Protect	Timeouts, limits, and TLS termination in one place
Hide	One public address for many internal services
Survive	Take a backend out when it fails, put it back when it recovers

The core, which is shorter than you expect

```
func (p *Proxy) ServeHTTP(w http.ResponseWriter, r *http.Request) {
    backend, release, ok := p.pool.Pick(r)
    if !ok {
        http.Error(w, "no healthy backend", 503)
        return
    }
    defer release()

    outreq := r.Clone(r.Context())
    outreq.URL.Scheme = backend.URL.Scheme
    outreq.URL.Host = backend.URL.Host
    outreq.RequestURI = ""              // must be empty for a client
```

```
    stripHopByHop(outreq.Header)
    addForwarded(outreq, r)

    resp, err := p.transport.RoundTrip(outreq)
    if err != nil {
        p.pool.Fail(backend)
        http.Error(w, "upstream unavailable", 502)
        return
    }
    defer resp.Body.Close()

    stripHopByHop(resp.Header)
    copyHeader(w.Header(), resp.Header)
    w.WriteHeader(resp.StatusCode)
    p.copyBody(w, resp.Body)
}
```

Everything after this is the detail, and the detail is where proxies go wrong.

Hop-by-hop headers, and why they must be removed

Some headers describe **this connection**, not the message. Forwarding them onward is wrong because the next connection is a different connection:

```
var hopByHop = []string{
    "Connection", "Keep-Alive", "Proxy-Authenticate",
    "Proxy-Authorization", "TE", "Trailer",
    "Transfer-Encoding", "Upgrade",
}
```

And the subtle part: `Connection` can **name** other headers that are also hop-by-hop for this connection, so those must be stripped too.

> **WATCH OUT**
> Forwarding `Transfer-Encoding: chunked` to a backend that also sets a `Content-Length` is the smuggling setup from chapter fifty four, built by your own proxy.
>
> Strip, then let your HTTP client decide the framing for the connection it is making. Never copy framing headers across a hop.

X-Forwarded-For, and the header that lies

```
func addForwarded(out, in *http.Request) {
    ip, _, _ := net.SplitHostPort(in.RemoteAddr)

    prior := in.Header.Get("X-Forwarded-For")
```

```
    if prior != "" {
        ip = prior + ", " + ip
    }
    out.Header.Set("X-Forwarded-For", ip)
    out.Header.Set("X-Forwarded-Proto", scheme(in))
    out.Header.Set("X-Forwarded-Host", in.Host)
}
```

The chain grows as the request passes each proxy, so the **leftmost** entry is the original client and each later one was added by a hop.

> **WATCH OUT**
> **A client can send `X-Forwarded-For` itself.** Nothing stops it. So the leftmost value is not the client's IP; it is whatever the client claimed, followed by the truth.
>
> The consequence is real: an application that rate limits, geolocates or audits by the leftmost entry can be fooled by anybody who sends a header. Logs attribute actions to an invented address.
>
> The rule: **count from the right, and trust exactly as many entries as you have proxies you control**. One proxy means the real client IP is the last entry your proxy appended, not the first in the list. Configure the count explicitly; do not infer it.

Choosing a backend

Round robin is two lines and is the right default:

```
func (p *Pool) roundRobin() *Backend {
    n := atomic.AddUint64(&p.next, 1)
    healthy := p.healthy()
    if len(healthy) == 0 {
        return nil
    }
    return healthy[n%uint64(len(healthy))]
}
```

Least connections is better when request durations vary a lot, which is most real traffic:

```
func (p *Pool) leastConns() *Backend {
    var best *Backend
    for _, b := range p.healthy() {
        if best == nil ||
            b.inflight.Load() < best.inflight.Load() {
            best = b
        }
    }
```

```
    return best
}
```

Round robin	Even request counts. Fine when requests cost the same
Least connections	Even **load**. Better with mixed fast and slow endpoints
Weighted	Backends of different sizes
Consistent hash	The same key goes to the same backend, for cache locality or sticky sessions

Round robin's failure mode is worth knowing: one slow backend keeps receiving its share, so a quarter of your users wait on the sick machine while three healthy ones idle. Least connections drains away from it automatically, because its in-flight count stays high.

Health, actively and passively

Active: ask each backend, on a timer.

```
func (p *Pool) probe(b *Backend) {
    ctx, cancel := context.WithTimeout(
        context.Background(), 2*time.Second)
    defer cancel()

    req, _ := http.NewRequestWithContext(ctx, "GET",
        b.URL.String()+"/health", nil)
    resp, err := p.probeClient.Do(req)

    switch {
    case err != nil || resp.StatusCode >= 500:
        b.markUnhealthy()
    default:
        resp.Body.Close()
        b.markHealthy()
    }
}
```

Passive: notice real traffic failing. A backend that refuses three connections in a row is out, regardless of what its health endpoint says.

Recovery must be careful. A backend that has been out comes back **half open**: it gets one request, and only if that succeeds does it rejoin.

> **WATCH OUT**

Without the half-open step, a backend that is still broken rejoins the pool on a timer, takes its full share, fails them all, and leaves again. The pool oscillates and a steady fraction of users get errors for as long as the backend is sick.

This is the circuit breaker pattern, and the half-open state is the part people leave out.

Thresholds matter too: two consecutive failures to eject, three consecutive successes to restore. One failure is a blip, and treating it as an outage makes the proxy less reliable than no proxy.

Retries, and the one that corrupts data

Retrying a failed request sounds obviously good. It is conditional:

```
func retryable(r *http.Request, err error, sentBytes bool) bool {
    if sentBytes {
        return false        // the backend may have acted on it
    }
    switch r.Method {
    case "GET", "HEAD", "OPTIONS", "PUT", "DELETE":
        return isConnectionError(err)
    default:
        return false        // POST and PATCH are not idempotent
    }
}
```

WATCH OUT

Never retry a POST that was fully sent. A connection that failed after the request reached the backend may have failed on the way back, and the work is already done. Retrying charges the card twice.

A connection **refused** is safe to retry: nothing was delivered. A connection **reset mid-response** is not.

And cap the total: a retry budget of ten percent of traffic. Without a cap, a failing backend turns into three times the traffic at the exact moment the system is least able to take it, which is how a small outage becomes a retry storm.

Streaming, and the buffering that breaks it

```
func (p *Proxy) copyBody(w http.ResponseWriter, body io.Reader) {
    if flusher, ok := w.(http.Flusher); ok && p.flushInterval > 0 {
        // Server-sent events, long polling, log tails: the client
        // needs each write as it happens, not when the buffer is
        // full or the response ends.
```

```
        copyAndFlush(w, body, flusher, p.flushInterval)
        return
    }
    io.Copy(w, body)
}
```

This is why your streaming endpoint works locally and arrives all at once through a proxy: the default is to buffer, and a stream needs the opposite. Detect it by `Content-Type: text/event-stream`, or by the absence of a `Content-Length`, and flush.

WebSockets need the connection itself, not the body: on a `101 Switching Protocols`, hijack the client connection and copy bytes in both directions until either side closes.

Draining a backend

Deployment is where a proxy earns its place. Taking a backend out **gracefully** means: stop sending it new requests, let the in-flight ones finish, then stop the process.

```
func (b *Backend) Drain(ctx context.Context) error {
    b.accepting.Store(false)
    for b.inflight.Load() > 0 {
        select {
        case <-ctx.Done():
            return ctx.Err()
        case <-time.After(50 * time.Millisecond):
        }
    }
    return nil
}
```

That loop, plus chapter twenty one's graceful shutdown in the backend, is a zero-downtime deploy. There is nothing else to it.

PROJECT 52: THE PROXY, MEASURED UNDER FAILURE

In `part6-from-scratch/56-proxy`, build it and then break things on purpose.

One: the proxy. Header stripping, forwarded headers, round robin, and a configurable backend list.

Two: three backends. Write a tiny backend that reports its own name and can be told to be slow or to fail:

```
./backend -name a -port 9001
./backend -name b -port 9002 -latency 500ms
./backend -name c -port 9003 -fail-rate 0.5
```

Three: the strategies. Round robin and least connections, both implemented. Drive 10,000 requests through each with `hey` or `k6` and record p50, p95 and p99 for both. The difference with backend `b` slow is the whole lesson.

Four: health. Active probes and passive ejection with thresholds, plus the half-open recovery. Kill backend `a` mid-load and record how many requests failed before it was ejected. Then restart it and time the recovery.

Five: the oscillation. Remove the half-open step, make a backend fail its probe but fail real traffic, and watch the error rate pulse. Put it back.

Six: streaming. Add an SSE endpoint to a backend and prove the events arrive one at a time through the proxy. Then set the flush interval to zero and watch them arrive in a lump.

Seven: the forged header. Send a request with your own `X-Forwarded-For` and show what your backend logs. Then implement the trusted-hop count and show it logging the truth.

Eight: the drain. Deploy a new version of a backend with zero failed requests under load. That is the proof.

YOUR TURN

Put your proxy in front of the Grit API from Part V, with two copies of the API running.

Then stop one mid-session in the admin panel. If you did the health checks and draining properly, the person using it sees nothing at all, which is the most satisfying negative result in this book.

ASSIGNMENT

Build the proxy and report its behaviour under failure, with numbers.

One. The proxy with both balancing strategies, active and passive health checks, half-open recovery, conditional retries with a budget, and streaming support.

Two. `BALANCING.md`: p50, p95 and p99 for both strategies, with all backends healthy and with one slow, as four columns of a table.

Three. `FAILURE.md`: requests lost when a backend dies, before and after passive ejection; the oscillation experiment with and without half-open; and the drain with zero errors.

Four. The forged `X-Forwarded-For`, before and after the trusted-hop fix, with both log lines.

Five. One paragraph: which of nginx, Caddy, HAProxy or an application load balancer would you use instead of this, for what, and which one feature of yours would you miss.

COMMIT AS

```
git commit -m "Reverse Proxy - Assignment One"
```

57

Build your own job queue

Chapter forty three used a queue. This chapter builds one, on the database you already have, which for most applications is the right answer: no Redis, no second thing to operate, and jobs that can be enqueued **inside the same transaction** as the work that caused them.

The hard parts are not the obvious ones. Storing a job is a row. Handing that row to exactly one worker, and getting it back when that worker dies, is the engineering.

The table

```
CREATE TABLE jobs (
  id           BIGSERIAL PRIMARY KEY,
  queue        TEXT        NOT NULL DEFAULT 'default',
  type         TEXT        NOT NULL,
  payload      JSONB       NOT NULL,

  state        TEXT        NOT NULL DEFAULT 'pending',
  attempts     INT         NOT NULL DEFAULT 0,
  max_attempts INT         NOT NULL DEFAULT 5,

  run_at       TIMESTAMPTZ NOT NULL DEFAULT now(),
  locked_at    TIMESTAMPTZ,
  locked_by    TEXT,
  last_error   TEXT,
  unique_key   TEXT,
  created_at   TIMESTAMPTZ NOT NULL DEFAULT now()
);

CREATE INDEX jobs_claim ON jobs (queue, state, run_at)
  WHERE state = 'pending';

CREATE UNIQUE INDEX jobs_unique_key ON jobs (unique_key)
  WHERE unique_key IS NOT NULL AND state <> 'done';
```

Two index decisions carry most of the performance.

The partial index. Claiming only ever looks for pending rows, and a finished job is dead weight in the index. A partial index stays small even when the table holds millions of completed jobs.

The partial unique index is idempotency: a job with a `unique_key` that is already pending or running cannot be enqueued twice, and the database enforces it rather than the application checking first and racing.

Enqueue, inside your transaction

```
func (q *Queue) Enqueue(ctx context.Context, tx *sql.Tx,
    j Job) error {

    _, err := tx.ExecContext(ctx, `
        INSERT INTO jobs (queue, type, payload, run_at, unique_key)
        VALUES ($1, $2, $3, $4, $5)
        ON CONFLICT DO NOTHING`,
        j.Queue, j.Type, j.Payload, j.RunAt, j.UniqueKey)
    return err
}
```

> **IF YOU KNOW JAVASCRIPT**
> That `tx` parameter is the whole reason to put a queue in your database.
>
> With Redis, enqueueing and committing are two systems, which is chapter forty four's publish-or-commit problem and needs an outbox. Here the job row and the business row commit together or not at all: the outbox **is** the queue.
>
> It is the strongest argument for a database-backed queue, and it disappears the moment your throughput needs something specialised.

Claiming work, which is the whole problem

Several workers poll the same table. Two must never get the same row, and a worker waiting on a lock is a worker doing nothing.

```
UPDATE jobs SET
  state     = 'running',
  locked_at = now(),
  locked_by = $1,
  attempts  = attempts + 1
WHERE id IN (
  SELECT id FROM jobs
  WHERE queue = $2 AND state = 'pending' AND run_at <= now()
  ORDER BY run_at
  LIMIT $3
  FOR UPDATE SKIP LOCKED
)
RETURNING id, type, payload, attempts;
```

`FOR UPDATE SKIP LOCKED` is the line that makes this work, and it exists in Postgres for exactly this use case.

> **WATCH OUT**
> Without `SKIP LOCKED`, ten workers all select the same oldest rows, nine block on the first one's lock, and your queue processes jobs one at a time with nine idle workers holding transactions open.
>
> With it, each worker takes the rows nobody else has locked and moves on.
>
> MySQL 8 has it too. SQLite does not, and serialises writers anyway, so there a single claiming goroutine handing work to a pool is the honest design.

Claim a batch rather than one row. The round trip dominates at small sizes, and ten or twenty jobs per claim is usually an order of magnitude more throughput.

When a worker dies

The job is `running`, `locked_by` names a process that no longer exists, and nothing will ever finish it. The lease is what fixes this:

```sql
-- The reaper, every 30 seconds
UPDATE jobs
SET state = 'pending', locked_at = NULL, locked_by = NULL
WHERE state = 'running'
  AND locked_at < now() - interval '5 minutes';
```

And a worker on a long job keeps its lease alive:

```go
func (w *Worker) heartbeat(ctx context.Context, id int64) {
    t := time.NewTicker(30 * time.Second)
    defer t.Stop()
    for {
        select {
        case <-ctx.Done():
            return
        case <-t.C:
            w.db.ExecContext(ctx,
                `UPDATE jobs SET locked_at = now() WHERE id = $1`,
                id)
        }
    }
}
```

> **WATCH OUT**
> The lease is why delivery is **at least once** and can never be exactly once.

A worker that finishes the work and dies before marking the job done will have that job reclaimed and run again. There is no ordering of those two steps that removes the window, which is the same impossibility chapter forty three described for mail.

So the handler must be idempotent, and the lease duration must exceed your longest job. A five minute lease on a job that takes ten minutes means the job is reclaimed and run again **while it is still running**, which is the worst outcome available: two workers doing the same work at the same time.

Failure, backoff and the dead letter

```
func (w *Worker) fail(ctx context.Context, j Job, err error) {
    if j.Attempts >= j.MaxAttempts {
        w.db.ExecContext(ctx, `
            UPDATE jobs SET state = 'dead', last_error = $1
            WHERE id = $2`, err.Error(), j.ID)
        return
    }
    delay := backoff(j.Attempts)
    w.db.ExecContext(ctx, `
        UPDATE jobs SET state = 'pending', run_at = $1,
            last_error = $2, locked_at = NULL, locked_by = NULL
        WHERE id = $3`, time.Now().Add(delay), err.Error(), j.ID)
}

func backoff(attempt int) time.Duration {
    base := time.Duration(math.Pow(2, float64(attempt))) *
        time.Second
    if base > 10*time.Minute {
        base = 10 * time.Minute
    }
    // Jitter, and it is not optional.
    jitter := time.Duration(rand.Int63n(int64(base / 2)))
    return base/2 + jitter
}
```

WATCH OUT

The jitter is the part people skip, and it is the part that matters during an incident.

A downstream service goes down for a minute. Five thousand jobs fail at once. Without jitter, all five thousand retry at exactly the same instant, four seconds later, and again sixteen seconds later. The service comes back, gets hit by five thousand simultaneous requests, and goes down again. You have built a mechanism that prevents recovery.

Spread the retries. Full jitter, half jitter, anything: the shape matters far more than the formula.

A dead job is not deleted. It sits in the table with its error, where somebody can look at it, fix the cause and requeue it. A queue with no dead letter state loses work silently, which is the one thing a queue must never do.

Polling, and the latency it costs

Polling every second means up to a second of latency for every job. For a reminder email that is irrelevant; for a user waiting on a thumbnail it is not.

Postgres gives you a way out:

```
NOTIFY jobs_default;
```

```
// The worker waits on the notification OR a timer, whichever
// comes first. The timer is the floor: notifications can be
// missed, and polling is the thing that makes it correct.
select {
case <-notified:
case <-time.After(pollInterval):
}
```

Notifications are an optimisation, not a guarantee. Keeping the poll as a floor means a missed notification costs latency rather than a stuck job.

Fairness, and the queue that starves

With several queues, the naive loop takes everything from the first queue before looking at the second, so a flood of image jobs stops every email.

Weighted selection, by starting from a random position in a weighted list, is enough: `default` weight 5, `email` 3, `images` 2. Over any reasonable window each queue gets its share, and no queue can be starved by another.

Knowing it works

Four numbers, and the second is the one to alert on:

Counts by state	Pending, running, dead. The shape of the backlog
Oldest pending age	**The single best queue alert there is**
Duration by type	Which jobs are getting slower
Dead per hour	A rise means something downstream broke

Depth alone is misleading: ten thousand jobs enqueued a second ago is fine, and one job pending for an hour is an outage. Age catches both.

Shutting down without losing work

```
func (w *Worker) Shutdown(ctx context.Context) error {
    w.stopClaiming()                 // no new jobs
    done := make(chan struct{})
    go func() { w.wg.Wait(); close(done) }()

    select {
    case <-done:
        return nil
    case <-ctx.Done():
        // Out of time: release the leases we hold so another
        // worker picks them up now rather than in five minutes.
        return w.releaseHeldJobs()
    }
}
```

That last branch is worth the few lines. A deploy that releases its leases makes the next worker pick those jobs up immediately, instead of them sitting `running` until the reaper notices.

When this is the wrong design

Be honest about the ceiling. A database-backed queue handles a few thousand jobs a second on decent hardware, which is more than most applications ever produce. Beyond that, the claim query becomes the bottleneck and the table becomes a hot spot.

Database queue	You already have a database, you want transactional enqueue, and volume is moderate
Redis queue	High throughput, low latency, jobs that are not tied to a transaction
Kafka or similar	Streams, many consumers, replay, retention. A different tool for a different problem

PROJECT 53: A QUEUE YOU WOULD TRUST

In `part6-from-scratch/57-queue`, build it against Postgres in Docker.

One: the schema and enqueue, with the partial indexes and transactional enqueue taking a `*sql.Tx`.

Two: claiming with `FOR UPDATE SKIP LOCKED`, batched. Run eight workers against ten thousand jobs and assert every job ran exactly once.

Three: prove `SKIP LOCKED` matters. Remove it, run the same test, and record the throughput difference and what the workers were doing.

Four: the dead worker. Kill a worker mid-job with `kill -9`. Confirm the job is stuck `running`, then watch the reaper return it. Measure the recovery time.

Five: the lease trap. Set the lease to ten seconds and run a job that takes thirty. Observe two workers running it at once, and write down what that would have done if the job charged a card.

Six: backoff. Make a job fail every time and plot its retry times. Then remove the jitter, fail five thousand jobs at once, and graph the arrivals. The spike is the chapter's point.

Seven: fairness. Flood one queue and confirm another still progresses.

Eight: observability. A `/queue/stats` endpoint with the four numbers, and an alert rule on oldest pending age. State the threshold and why.

YOUR TURN

Enqueue a job inside a transaction that then rolls back.

Nothing should be queued. Now do the same against a Redis queue from chapter forty three and watch the job run for a row that does not exist. That one experiment is the clearest demonstration of why the outbox exists.

ASSIGNMENT

Build the queue and prove each guarantee with an experiment, not a claim.

One. The full implementation: transactional enqueue, batched claiming with `SKIP LOCKED`, leases and a reaper, jittered backoff, a dead letter state, weighted fairness and graceful shutdown.

Two. `GUARANTEES.md`, one short section each: exactly once or at least once and why, what happens when a worker dies, what happens when a job outlives its lease, and what happens when the database is briefly down.

Three. The eight-worker correctness test, with the count of jobs run exactly once out of ten thousand.

Four. The jitter graph, before and after, even if it is ASCII.

Five. One paragraph on when you would replace this with Redis or Kafka for your own workload, with the number that would make you switch.

COMMIT AS

```
git commit -m "Job Queue - Assignment One"
```

58

Build your own test runner, and what go test really does

`go test` feels like magic in a way the rest of Go does not. There is no registration, no decorator, no configuration file. You write a function whose name starts with `Test`, and it runs.

This chapter is how. By the end you will have a runner that discovers tests, builds a binary, runs them in parallel with timeouts and isolation, reports failures usefully, and emits machine-readable output. And you will know precisely which of Go's conveniences are compiler features and which are ordinary code you could have written.

The trick: it writes a program

`go test` does not interpret your test files. It **generates a `main` package** that imports your package, lists your test functions in a slice, and calls `testing.Main` with them. Then it compiles that program and runs it.

```
// Roughly what go test generates, simplified
func main() {
    tests := []testing.InternalTest{
        {Name: "TestAdd", F: pkg.TestAdd},
        {Name: "TestSub", F: pkg.TestSub},
    }
    testing.Main(matchString, tests, benchmarks, examples)
}
```

> **IF YOU KNOW JAVASCRIPT**
> This is the answer to a question most Go developers never ask: how is there no registration step?
>
> In JavaScript, `describe` and `it` are function calls that push into a registry at import time. In Go there is no import-time registry, because the **tooling** walks the source, finds the functions and writes the list. The magic is in the build step, not the language.
>
> Which means you can build it yourself, and that is exactly what this chapter does.

Step one: discovery

Parse the package with `go/parser` and find the functions. This is the same `go/ast` work `grit sync` does to read models, so the shape should be familiar:

```
func discover(dir string) ([]Test, error) {
    fset := token.NewFileSet()
    pkgs, err := parser.ParseDir(fset, dir, isTestFile,
        parser.ParseComments)
    if err != nil {
        return nil, err
    }

    var out []Test
    for _, pkg := range pkgs {
        for path, file := range pkg.Files {
            for _, decl := range file.Decls {
                fn, ok := decl.(*ast.FuncDecl)
                if !ok || !isTestFunc(fn) {
                    continue
                }
                out = append(out, Test{
                    Name: fn.Name.Name,
                    File: path,
                    Line: fset.Position(fn.Pos()).Line,
                })
            }
        }
    }
    sort.Slice(out, func(i, j int) bool {
        return out[i].Name < out[j].Name
    })
    return out, nil
}
```

`isTestFunc` is the rule, written out:

```
func isTestFunc(fn *ast.FuncDecl) bool {
    if fn.Recv != nil {                    // not a method
        return false
    }
    if !strings.HasPrefix(fn.Name.Name, "Test") {
        return false
    }
    // TestXxx: the character after "Test" must not be lowercase,
    // so Testing() is not a test and TestAdd is.
    rest := fn.Name.Name[len("Test"):]
    if rest != "" {
        r, _ := utf8.DecodeRuneInString(rest)
```

```
        if unicode.IsLower(r) {
            return false
        }
    }
    return hasSignature(fn, "*testing.T")
}
```

That lowercase rule is real and catches people: a function called `Testing` is not a test, and `Testadd` is not either.

Step two: generate and build

```
const mainTmpl = `package main

import (
    "os"
    "testing"

    target "{{.ImportPath}}"
)

func main() {
    tests := []testing.InternalTest{
{{range .Tests}}        {Name: "{{.Name}}", F: target.{{.Name}}},
{{end}}    }
    m := testing.MainStart(deps{}, tests, nil, nil, nil)
    os.Exit(m.Run())
}
`
```

Write it to a temporary directory, then:

```
go build -o runner.test ./tmpdir
```

Notice what this means: **your test runner is a compiler front end**. It produces Go source and hands it to the Go toolchain, which is how `go test` gets types, build tags and vet checks for free.

> **WATCH OUT**
> The generated file imports your package, so a test in package `foo` can only see exported identifiers. That is why Go has **two** kinds of test file: `package foo` tests, compiled into the package and able to see everything, and `package foo_test` tests, which see only the public API.

`go test` handles both by generating two sets of entries. It is worth supporting in your version, because the distinction is one of the better habits Go encourages: a `_test` package forces you to use your own API the way a caller does.

Step three: running, in parallel, in isolation

```
func (r *Runner) Run(ctx context.Context, tests []Test) Report {
    sem := make(chan struct{}, r.Parallel)
    results := make([]Result, len(tests))
    var wg sync.WaitGroup

    for i, t := range tests {
        wg.Add(1)
        go func(i int, t Test) {
            defer wg.Done()
            sem <- struct{}{}
            defer func() { <-sem }()

            results[i] = r.runOne(ctx, t)
        }(i, t)
    }
    wg.Wait()
    return report(results)
}
```

Each test runs as a subprocess invocation of the built binary with `-test.run=^TestName$`:

```
func (r *Runner) runOne(ctx context.Context, t Test) Result {
    ctx, cancel := context.WithTimeout(ctx, r.Timeout)
    defer cancel()

    dir, err := os.MkdirTemp("", "testrun-")
    if err != nil {
        return Result{Test: t, Err: err}
    }
    defer os.RemoveAll(dir)

    cmd := exec.CommandContext(ctx, r.Binary,
        "-test.run=^"+t.Name+"$", "-test.v")
    cmd.Dir = dir
    var out bytes.Buffer
    cmd.Stdout, cmd.Stderr = &out, &out

    start := time.Now()
    err = cmd.Run()

    return Result{
```

```
        Test:     t,
        Passed:   err == nil,
        Output:   out.String(),
        Duration: time.Since(start),
        TimedOut: ctx.Err() == context.DeadlineExceeded,
    }
}
```

A process per test is heavier than `go test`'s goroutine per test, and it buys two things worth having: a panic kills one test rather than the run, and a test that leaks goroutines or global state cannot affect its neighbours.

> **WATCH OUT**
> Each test gets its own temporary working directory, and that is not politeness.
>
> A test that writes `output.json` in the current directory passes alone and fails when run next to another test that does the same. Those failures are order dependent, so they appear when somebody adds an unrelated test, and they are blamed on the new test.
>
> Isolate the filesystem and the environment, and **randomise the order** occasionally. A suite that passes in a random order is a suite whose tests are genuinely independent.

Step four: output people can use

Two audiences, as in chapter forty nine.

For a human, failures only, with the file and line so an editor can jump there:

```
FAIL  TestInvoiceTotal  (0.03s)
  invoice_test.go:42: Total() = 1999, want 2000

ok    38 passed, 1 failed, 2 skipped  in 1.4s
```

For a machine, one JSON object per event, which is what `go test -json` produces and what every CI system consumes:

```
{"Action":"run","Test":"TestInvoiceTotal"}
{"Action":"output","Test":"TestInvoiceTotal","Output":"..."}
{"Action":"fail","Test":"TestInvoiceTotal","Elapsed":0.03}
```

Streaming the events rather than printing a report at the end matters more than it sounds: a twenty minute suite that prints nothing until it finishes is indistinguishable from a hung one.

> **WATCH OUT**
> Print the **failures** last, or print them twice.

Scrolling back through four thousand lines of passing output to find the one failure is a daily annoyance in a hundred CI systems. A summary at the end, repeating each failure with its output, costs nothing and is the single most appreciated feature of a runner.

This is also why you never pipe a test log through `tail`: the summary survives and the context that explains it does not.

The features that make a runner good

Having built the core, these are each an afternoon and each one you will miss if you leave it out:

Filtering	`-run` with a regular expression. The most used flag there is
Timeouts	Per test and overall, with the stack dumped on timeout
Retry flaky	`-retry 2`, and **report** the retry rather than hiding it
Shuffle	Random order with a printed seed, so a failure is reproducible
Sharding	`-shard 2/5`, for splitting a suite across CI machines
Slowest	Print the ten slowest tests. The list is always surprising

IF YOU KNOW JAVASCRIPT

The retry one deserves a warning. A runner that silently retries turns a flaky test into a passing test, and the flakiness becomes invisible until it is twenty tests and nobody trusts the suite.

Retry if you must, but print `FLAKY: TestX passed on attempt 2` and count it in the summary. Make the flakiness visible, or you are building a machine for ignoring problems.

What `go test` does that yours will not

Worth knowing, because the list is a map of the remaining depth:

Caching	Results keyed by the inputs: source, environment, files read. This is why an unchanged test prints `(cached)`
Coverage	Rewriting the source to add counters before compiling
Benchmarks	Adaptive iteration counts until the timing is stable
Fuzzing	Corpus management, mutation, minimisation of a failing input
Subtests	`t.Run`, with its own parallelism and naming rules
Vet	A subset of checks run automatically before the tests

The caching one is the most interesting piece of engineering in the list: it hashes everything that could affect the outcome, including the files the test read, and that is why it is safe to trust.

PROJECT 54: A RUNNER YOU WOULD USE

In `part6-from-scratch/58-runner`, build it and then run it on real code.

One: discovery. Parse a package, list its tests with file and line. Compare your list with `go test -list '.*'` on the same package: they must match exactly, including the `Testing` and `Testadd` edge cases.

Two: generate and build the runner binary.

Three: run with configurable parallelism, a per-test timeout, a temporary working directory each, and a summary that repeats the failures.

Four: both outputs. Human and JSON-lines. Pipe the JSON into `jq` to print the slowest five tests.

Five: the ordering bug. Write two tests that both write `out.txt` in the current directory. Show them failing without isolation and passing with it.

Six: shuffle and shard. Random order with a printed seed, and `-shard i/n`. Prove the shards are disjoint and cover everything.

Seven: run it on something real. Point it at the test suite of a project from Part V and compare the output with `go test`. Every difference is either a feature you have not built or a bug.

Eight: time it. Your runner against `go test` on the same package, both cold. Report both numbers and explain the difference, which will mostly be process startup.

YOUR TURN

Run `go test -x ./...` on any package and read the commands it prints.

You will see the generated `_testmain.go`, the build, and the invocation. Everything in this chapter is visible in that output, and after building your own, it reads as a description rather than as noise.

ASSIGNMENT

Build the runner, and use it on real code rather than on examples.

One. Discovery, generation, parallel isolated execution, timeouts, both output formats, shuffle with a seed, and sharding.

Two. `COMPARE.md`: your test list against `go test -list`, your pass and fail counts against `go test`, and the timing of both on a real package.

Three. The isolation demonstration: two tests that collide, failing and then passing.

Four. One flaky test, run with retries, showing that your output **reports** the retry rather than hiding it.

Five. A paragraph on caching: what would you have to hash to make a cached result trustworthy? List everything you can think of, then look up what the Go tool actually uses and note what you missed.

This is the last build in the book. When it runs your own earlier chapters' tests, you have gone from "I have never written Go" to "I wrote the thing that runs my tests", which is a reasonable place to stop.

COMMIT AS

```
git commit -m "Test Runner - Assignment One"
```

Appendix: the JavaScript to Go phrasebook

The fastest way to write Go in your first month is to know what the thing you already do is called. This is that list: the JavaScript you reach for, and the Go you want instead. It is a lookup table, not a tutorial. Each row names the chapter that explains why.

Values and types

JavaScript	Go
`let x = 1`	`x := 1`
`const x = 1`	`const x = 1`, or `x := 1` you do not reassign
`undefined` / `null`	The zero value, or a pointer that is `nil`. Ch 3
`typeof x`	`fmt.Sprintf("%T", x)`, or a type switch. Ch 10
`Number(s)`	`strconv.Atoi(s)`, which returns an error too
`String(n)`	`strconv.Itoa(n)` or `fmt.Sprint(n)`
`` `hi ${name}` ``	`fmt.Sprintf("hi %s", name)`
`x ?? y`	`if x == zero { x = y }`. No coalescing operator
`x?.y`	`if x != nil { ... }`. No optional chaining. Ch 8

Collections

JavaScript	Go
`[]`	`[]T{}` or `make([]T, 0, n)`. Ch 6
`arr.push(v)`	`arr = append(arr, v)`, and the assignment matters
`arr.length`	`len(arr)`

`arr.map(f)`	A `for` loop, or a generic helper you write once. Ch 17
`arr.filter(f)`	A `for` loop with an `if`
`arr.find(f)`	A `for` loop with an early `return`
`arr.includes(v)`	`slices.Contains(arr, v)`
`arr.sort(cmp)`	`sort.Slice(arr, less)` or `slices.SortFunc`
`[...a, ...b]`	`append(append([]T{}, a...), b...)`
`{}`	`map[string]T{}` or a struct. Ch 5, Ch 7
`obj.key`	`m["key"]`, or `s.Field` on a struct
`key in obj`	`v, ok := m[key]`. Ch 7
`delete obj.key`	`delete(m, key)`
`Object.keys(o)`	A loop over the map, then sort. Order is random
`new Set()`	`map[T]struct{}{}`

Functions and errors

JavaScript	Go
`function f(a) {}`	`func f(a int) {}`
`const f = (a) => a * 2`	`f := func(a int) int { return a * 2 }`
`try { } catch (e) { }`	`v, err := f()` then `if err != nil`. Ch 4
`throw new Error("x")`	`return errors.New("x")`
`e instanceof MyError`	`errors.As(err, &myErr)`. Ch 11
`e.cause`	`errors.Unwrap(err)`, from `%w`
`finally { }`	`defer`. Ch 12
`process.exit(1)`	`os.Exit(1)`, or return an error to `main`

Concurrency

JavaScript	Go
`async function f()`	Nothing. Every function can block. Ch 13
`await f()`	`f()`
`f().then(...)`	`go func() { ... }()`
`Promise.all(ps)`	`sync.WaitGroup`, or `errgroup`. Ch 13

`Promise.race(ps)`	`select` over channels. Ch 14
`AbortController`	`context.WithCancel`. Ch 16
`setTimeout(f, ms)`	`time.AfterFunc(d, f)`
`setInterval(f, ms)`	`time.NewTicker(d)` and a loop
A shared mutable object	`sync.Mutex`, or do not share it. Ch 15

Modules and tooling

JavaScript	Go
`package.json`	`go.mod`. Ch 9
`node_modules/`	A module cache outside your project
`npm install x`	`go get x`
`npm install`	`go mod download`, usually implicit
`import { x } from "y"`	`import "y"`, then `y.X`
`export const x`	A capital letter. Ch 2
`export default`	Nothing. Name the thing
`npm test`	`go test ./...`. Ch 18
`npm run build`	`go build ./...`. Ch 25
`nodemon`	`air`, or just rebuild. It is fast
`eslint`	`go vet`, plus `gofmt` which is not optional
`prettier`	`gofmt`. There are no options and that is the feature

The standard library, by what you would npm install

npm package	Go
`express`	`net/http`. Ch 21
`axios` / `node-fetch`	`net/http` client
`dotenv`	`os.Getenv`, plus twenty lines. Ch 24
`uuid`	`crypto/rand`, or one small dependency
`lodash`	`slices`, `maps`, and a loop
`jest` / `vitest`	`testing`. Ch 18
`zod`	Struct tags plus a validator. Ch 19, Ch 36
`date-fns`	`time`. Ch 40

`winston` / `pino`	`log/slog`
`commander` / `yargs`	`flag`, or Ch 49

Five habits to unlearn

Checking for truthiness. `if x` is not valid Go, and that is deliberate. An empty string, a zero and a `nil` are different facts. Ch 3.

Reaching for a library first. The standard library covers more than you expect, and a Go project with four dependencies is normal rather than impressive. Ch 9.

Returning different shapes from one function. Go has one return type and an error. A function that sometimes returns a string and sometimes an object is not expressible, and the discipline is the point. Ch 4.

Mutating what you were handed. Everything is passed by value, but a slice or a map holds a reference to shared memory. The function that sorts its argument has changed the caller's data. Ch 6, Ch 8.

Ignoring the second return value. `_` is a decision to proceed on something you did not look at, and the compiler will let you. The cases where that is genuinely right are rare and deserve a comment. Ch 4, Ch 40.

Afterword: what to do on Monday

You have read a book, which is the easy part. What follows is the shortest path from here to being the person on a team who can be handed a backend.

The three things that actually move you

Finish the repository. If you did the assignments, you have fifty-odd programs and a commit history that shows a year of progress compressed into however long this took. If you skipped them, the single highest-value thing you can do this week is go back to chapter zero, make the repository, and work forward doing only the assignments. Reading teaches recognition; writing teaches recall, and only one of those survives an interview or a deadline.

Ship one thing to one real user. Not a portfolio piece, not a clone. Pick the application from Part V closest to something somebody you know does by hand, build it, and get them to use it for a month. Everything that matters and is not in this book, the questions users ask, the data they paste in, the ways they misunderstand your forms, arrives in that month.

Read code that is better than yours. The Go standard library is unusually readable, and it is right there: `go doc -src net/http.Server`. Read `io`, `errors`, `sync.Once`. They are short and they are the work of people who thought hard about exactly the things this book has been teaching you to think about.

The uncomfortable parts, named

Four things that will be harder than this book made them look, said plainly so that you recognise them rather than concluding you are bad at this.

Concurrency is easy to start and hard to be sure about. `go` is one keyword and correct concurrent code is a skill. Use the race detector, prefer channels and simple ownership, and be suspicious of any design you cannot explain in two sentences.

The framework is not the hard part. Modelling is. Deciding that an enrolment is a thing rather than a join table, that a sent invoice is immutable, that a date is not a timestamp: those decisions determine whether the project is pleasant in a year, and no generator makes them for you.

Production is a different skill. Writing the feature is a day. Operating it, backing it up, knowing when it is slow, deploying it without dropping requests, is the other half of the job and the half that is usually paid for.

You will be wrong about performance. Every time. Measure before optimising, and prefer the clearer code until a profile says otherwise. Chapter forty five exists for this.

What to learn next, in order

SQL, properly	Indexes, query plans, transactions and isolation levels. It will do more for your applications than any language feature
One more Part VI build	Pick the one you were most curious about. They are the fastest way to deepen the language
Profiling	`pprof`, flame graphs, allocation profiles. Three evenings, and you can answer questions most teams guess at
Linux basics	Processes, file descriptors, signals, systemd. Your binary runs there
A second domain	Payments, scheduling, inventory. Depth in a domain outlasts depth in a framework

On using a framework at all

This book argued for one for two hundred pages and then had you write a CRUD API by hand to see what it costs. Hold both.

A framework is worth it when it removes work you have already done and would do the same way again. It is a trap when it removes decisions you should be making, or when you cannot read what it produced. The reason Part VI exists is so that neither is true for you: the generated service is code you could have written, which means you can also judge when to leave it.

If you take one sentence from the whole book, take this one: **the thing that makes software expensive is rarely the code you wrote, it is the code nobody understands**. Generated or hand-written, read it.

Thank you, and one request

If you got here, you did a serious amount of work. Go is a language people describe as boring, and that is the compliment it is meant to be: it gets out of the way of the part that is actually hard, which is understanding the problem you were asked to solve.

The request: when you find an error in this book, say so. A wrong claim about the framework is checkable against the codebase, and a wrong claim about Go is checkable against the compiler. Both are worth fixing for the next person, who is in exactly the position you were in on page one.

Now go and build something somebody needs.