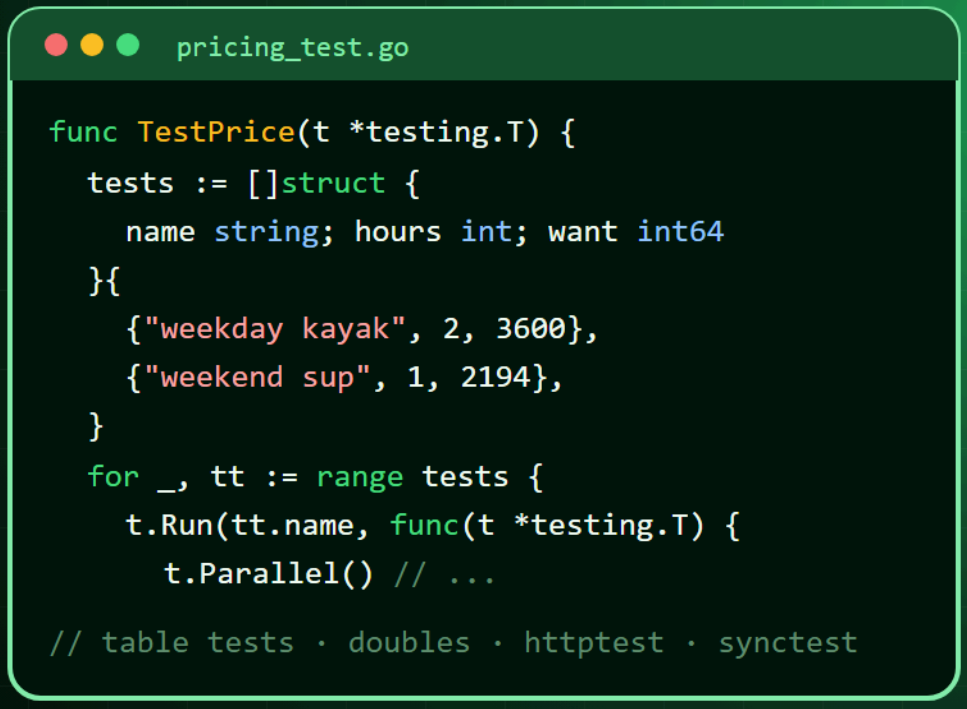
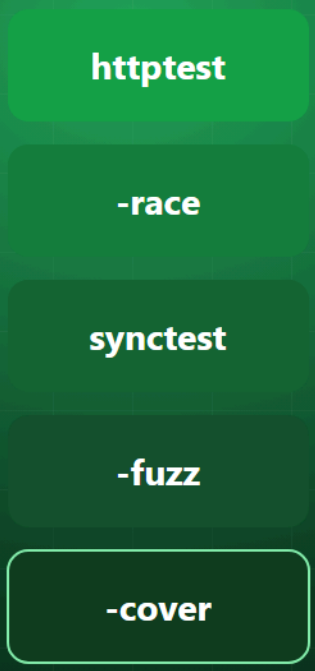





```
$ go test -race -shuffle=on ./...
ok paddlebook/internal/booking
ok paddlebook/internal/httpapi
ok paddlebook/internal/store/sqlitestore
```

A PRACTICAL STRATEGY

Testing Go Applications

From unit tests to HTTP, databases, fuzzing, concurrency, and CI

Go 1.27 · table tests · test doubles · httptest · SQLite
race detector · synctest · golden files · benchmarks

UNIT · INTEGRATION · END-TO-END · CI

THENEXBOOKS

Testing Go Applications

A Practical Strategy from Unit Tests to HTTP, Databases, Fuzzing, Concurrency, and CI
Table tests, test doubles, httptest, SQLite, synctest, golden files, benchmarks, and coverage — for Go 1.27

First Edition

THENEXBOOKS

Testing Go Applications
A Practical Strategy from Unit Tests to HTTP, Databases, Fuzzing, Concurrency, and CI

First Edition, 2026

Copyright © 2026 thenexbooks (thenexbooks.gumroad.com). All rights reserved.

No part of this publication may be reproduced, distributed, or transmitted in any form or by any means without the prior written permission of the publisher, except for brief quotations in reviews and certain other non-commercial uses permitted by copyright law. Code examples may be used freely in your own projects.

The information in this book is provided "as is" without warranty of any kind. While every precaution has been taken in its preparation, the publisher assumes no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein. Go, its tools, and the libraries discussed in this book evolve; always consult the official documentation for the versions you use.

Go and the Go gopher are trademarks of Google LLC. GitHub and GitHub Actions are trademarks of GitHub, Inc. Docker is a trademark of Docker, Inc. PostgreSQL is a trademark of the PostgreSQL Community Association of Canada. SQLite is in the public domain; "SQLite" is a registered trademark of Hipp, Wyrick & Company, Inc. Windows is a trademark of Microsoft Corporation. Other product names are trademarks of their respective owners. Their use in this book is for identification only and does not imply endorsement. This book is independent and is not affiliated with, sponsored by, or endorsed by Google, the Go project, or the maintainers of any library it discusses.

Larchmere Paddle Co., the business used as the running example in this book, is fictional, as are the lake Larchmere, the Paddlebook service, its people, customers, bookings, the Brightwind Weather API, and all data. Domains such as example.com and api.brightwind.example are reserved for examples.

Preface

Go makes testing feel almost too easy. The `testing` package ships with the language, `go test` needs no configuration, and a test is just a function whose name starts with `Test`. Within an hour of writing your first Go program you can write your first Go test. That is a gift, and it is also a trap: because starting is easy, many teams never stop to decide *what* to test, *how*, and *where*. The result is a familiar picture — a handful of tests around the easy parts, a few slow and fragile ones around the database, nothing at all around the HTTP layer or the background workers, and a suite that nobody fully trusts.

This book is about the other half of the job: turning Go's simple tools into a complete, practical testing strategy. We cover the whole stack of a real service — domain logic, test doubles, HTTP handlers and clients, databases, integration and end-to-end tests — and every tool in `go test` that helps you keep it honest: table-driven tests and subtests, golden files, fuzzing, benchmarks, the race detector, `testing/synctest` for deterministic time and concurrency, coverage, and continuous integration. At every step we explain the trade-offs, because a testing strategy is a set of judgment calls, not a checklist.

We do it with one realistic codebase. You will take a working but barely tested Go service and build its test suite chapter by chapter, running every command and reading every failure, until you can change any part of it with confidence.

Who This Book Is For

- **Go developers who write few tests** (or none) and want to know what to test first, how, and why.
- **Teams with a slow, flaky, or mock-heavy suite** who want tests that are fast, deterministic, and easy to change.
- **Backend developers building HTTP services** who need reliable tests for handlers, middleware, clients, databases, and background workers.
- **Developers who inherited untested Go code** and need a safe way to get it under test before changing it.

We assume you can already write Go: structs, interfaces, errors, goroutines, `net/http`, and `database/sql`. We do not teach the language itself; we teach how to test it.

What You Will Learn

- How `go test` works: test functions, failures, flags, caching, internal and external test packages, `testdata`, `TestMain`, cleanup, and the newer helpers such as `t.Context`, `t.Chdir`, and `t.ArtifactDir`.
- Table-driven tests, subtests, and clear failure messages with go-cmp, plus an honest comparison with testify.

- Designing Go code for testability with small interfaces, dependency injection, and seams — and writing stubs, fakes, and spies by hand, or generating mocks when interactions really matter.
- Testing time, randomness, the environment, error paths, edge cases, and panics.
- Testing HTTP handlers, JSON APIs, routing, and middleware with `httptest`; testing HTTP clients against fake servers with timeouts and retries; testing a whole server in-process and as a real binary.
- Database tests with real SQLite, repository contract tests, test data builders, and an integration test strategy with build tags and Testcontainers for PostgreSQL.
- Concurrent code: the race detector, goroutine leaks, cancellation, and deterministic tests with `testing/synctest`.
- Golden files, example tests, fuzzing that finds real bugs, and benchmarks that guard performance.
- Coverage that means something, flaky and slow tests diagnosed and cured, a complete CI pipeline, and techniques for getting legacy Go code under test.

How This Book Is Organized

Part I — Foundations of Go Testing (Chapters 1–5) sets out a testing strategy and covers the `testing` package, table-driven tests, comparisons and failure messages, and test organization. **Part II — Testable Design and Test Doubles** (Chapters 6–10) covers designing for testability, hand-written doubles, generated mocks, time and environment, and error paths. **Part III — Testing HTTP Services** (Chapters 11–15) moves from single handlers to JSON APIs and middleware, HTTP clients, the whole server in-process, and end-to-end tests of the real binary. **Part IV — Databases and Integration Tests** (Chapters 16–19) covers database code, contract tests, test data, and an integration test strategy. **Part V — Concurrency, Golden Files, Fuzzing, and Benchmarks** (Chapters 20–24) covers the race detector, `testing/synctest`, golden files, fuzzing, and benchmarks. **Part VI — Suite Quality, Coverage, and CI** (Chapters 25–28) covers coverage, flaky and slow tests, continuous integration, legacy code, and the complete strategy. The appendices provide a cheat sheet, test templates, a production checklist for test suites, and a glossary.

Each chapter ends with **Try It Yourself** exercises and a **Key Takeaways** box for quick review.

The Running Example: Larchmere Paddle Co.

Throughout the book we work on **Paddlebook**, the booking service of **Larchmere Paddle Co.**, a fictional dock that rents kayaks, canoes, a tandem kayak, and stand-up paddleboards by the hour. Ines owns the business, Malik wrote Paddlebook in a hurry last spring, Hana has just joined and is building the test suite with you, and Ruth runs the dock and prints the launch manifest every morning. Paddlebook is a small but real Go service: a domain package with pricing, refunds, and booking codes; in-memory and SQLite storage; a client for a weather API that can stop a booking when it is too windy;

a notifier; a background worker that expires unpaid holds; a daily launch manifest; and a JSON HTTP API. It starts the book with six tests. It ends the book with a complete, trustworthy test suite and a CI pipeline.

Versions and Conventions

Examples target **Go 1.27**, the current release at the time of writing, and the project's `go.mod` declares `go 1.26.0`, so the code runs on Go 1.26 and later. Where a feature needs a specific release — such as `testing/synctest` (stable since Go 1.25) or `httptest.NewTestServer` (Go 1.27) — we say so. Third-party modules are pinned to the versions we used, such as go-cmp v0.7.0 and modernc.org/sqlite v1.60.1; newer versions may differ in details, so a Note points you to the official documentation where that matters.

> **TIP**
> A recommended practice or a technique that makes your tests easier to write, read, or trust.

> **NOTE**
> Background information, version differences, or a useful aside.

> **WARNING**
> A common mistake or something that can make tests lie, flake, or miss real bugs.

Code appears in `monospace`. Long lines are wrapped to fit the page. File names above code blocks, such as `internal/booking/pricing_test.go`, tell you where the code lives. Commands are written for a Unix-like shell and work unchanged on Windows for everything `go` does; where shell syntax differs, we show both forms. Test output is real output from our machine, trimmed to fit; timings, benchmark numbers, and fuzzing statistics will differ on yours.

Contents

Preface

PART I — FOUNDATIONS OF GO TESTING

1 Why Test Go Code? A Strategy, the Test Pyramid, and the Paddlebook Service

2 The testing Package and go test: First Tests, Failures, and Flags

3 Table-Driven Tests and Subtests

4 Comparing Results and Reporting Failures: go-cmp, Errors, Helpers, and testify

5 Organizing Tests: Packages, testdata, TestMain, Cleanup, and the Test Lifecycle

PART II — TESTABLE DESIGN AND TEST DOUBLES

6 Designing Testable Go: Small Interfaces, Dependency Injection, and Seams

7 Hand-Written Test Doubles: Stubs, Fakes, Spies, and a Test Environment

8 Generated Mocks with go.uber.org/mock — and When Not to Use Them

9 Testing Time, Randomness, and the Environment

10 Testing Error Paths, Edge Cases, and Panics

PART III — TESTING HTTP SERVICES

11 Testing HTTP Handlers with httptest

12 Testing JSON APIs, Routing, and Middleware

13 Testing HTTP Clients: Fake Servers, Timeouts, and Retries

14 Testing the Whole Server In-Process: Wiring, Startup, and Shutdown

15 End-to-End Tests: Building the Binary and Driving It from Go

PART IV — DATABASES AND INTEGRATION TESTS

16 Testing Database Code with database/sql and SQLite

17 Repository Contract Tests: One Suite for Every Implementation

18 Test Data: Fixtures, Builders, Seeds, and Isolation

19 Integration Test Strategy: Build Tags, Short Mode, Testcontainers, and Real PostgreSQL

PART V — CONCURRENCY, GOLDEN FILES, FUZZING, AND BENCHMARKS

20 Testing Concurrent Code: The Race Detector, Goroutine Leaks, and Cancellation

21 Deterministic Time and Concurrency with testing/synctest

22 Golden Files, Snapshots, and Example Tests

23 Fuzzing with go test -fuzz

24 Benchmarks as Tests: b.Loop, Allocations, and Catching Regressions

PART VI — SUITE QUALITY, COVERAGE, AND CI

25 Code Coverage: Unit, Integration, and What the Numbers Mean

26 Flaky and Slow Tests: Diagnosis, Parallelism, and Cures

27 Continuous Integration for Go Tests: GitHub Actions, Reports, and Quality Gates

28 Testing Legacy Go Code and Paddlebook's Complete Testing Strategy

APPENDICES

A Go Testing Cheat Sheet

B Test Templates and Reference

C Test-Suite Production Checklist

D Glossary

PART I

Foundations of Go Testing

What a testing strategy is and why Go makes it easy to start but easy to get wrong. The testing package and go test, table-driven tests and subtests, comparisons and failure messages that explain themselves, and the conventions that keep a growing test suite organized.

1 · Why Test Go Code?
2 · The testing Package and go test
3 · Table-Driven Tests and Subtests
4 · Comparing Results and Reporting Failures
5 · Organizing Tests

CHAPTER 1 · PART I — FOUNDATIONS OF GO TESTING

Why Test Go Code? A Strategy, the Test Pyramid, and the Paddlebook Service

Tests are not a ritual you perform after the real work; they are how you earn the right to change code without fear. In this chapter we define what a testing strategy actually is, look honestly at the test pyramid and its critics, and meet Paddlebook, the small but realistic Go service whose test suite we will build together across the whole book.

A Tuesday Morning at Larchmere

Larchmere Paddle Co. rents kayaks, a tandem kayak, canoes, and stand-up paddleboards by the hour from a dock on the lake Larchmere. Every booking goes through **Paddlebook**, a Go HTTP service that Malik, the lead developer, wrote last spring in a hurry before the season opened. It works. Customers book, the dock staff print a launch manifest every morning, refunds go out when people cancel. Ines, the owner, would like it to stay that way.

The trouble starts with a simple request. Ines wants members to get their discount on weekend surcharges too, and she would like the refund rules written down somewhere other than Malik's head. Malik opens `pricing.go`, looks at the integer arithmetic, and hesitates. He is fairly sure he knows what the code does. He is not sure what else depends on it. The service has exactly six tests, and none of them would notice if a weekend canoe suddenly cost a dollar more.

Hana joined the team recently, and her first assignment is to change that situation: build a test suite that lets the team change Paddlebook on a Tuesday and sleep well on Tuesday night. Ruth, who runs the dock and prints the manifest every morning, has offered to report anything that looks wrong on paper. She owns no code, but she is the first person who notices when software lies.

That is a common starting point: a Go service that mostly works, a handful of tests that mostly pass, and a growing sense that the next change might break something nobody will notice until a customer does. Over twenty-eight chapters we will turn Paddlebook into a well-tested service, and you will learn every tool in `go test` and the judgment calls between them.

Why We Test

It is easy to say "tests catch bugs" and stop there. That is true but incomplete, and the incomplete version leads to suites that are large, slow, and oddly unhelpful. Here is the fuller list of what a good test suite buys you, roughly in the order teams discover it.

Confidence to change code

The main reason to test is not to prove that today's code is correct. It is to let you change tomorrow's code safely. Every feature request, dependency upgrade, and Go release is a change. Without tests, each change is a small gamble whose odds you cannot see. With a good suite, you make the change, run `go test ./...`, and find out within seconds whether the behavior you care about still holds. Malik's hesitation over `pricing.go` is exactly the cost of not having that feedback.

Room to refactor

Refactoring means changing the structure of code without changing its behavior. That definition is only useful if you can check the "without changing its behavior" part. Tests are that check. A suite that tests behavior through stable interfaces lets you rename, split, and simplify freely; a suite that is tightly coupled to implementation details does the opposite and punishes every refactor. Much of this book is about writing the first kind.

Executable documentation

A test named `TestRefundHalfWithin24Hours` that cancels a $40.50 booking twenty-three hours before the start and expects $20.25 back is a specification that cannot go out of date. Comments drift; a test that runs on every pull request either stays true or turns red. When Ines asks "what happens if someone cancels at exactly twenty-four hours?", the best answer is a test name and a passing run.

Design feedback

Code that is hard to test is usually hard to use. If testing the booking flow requires a real weather API, a real clock, and a real email server, that is not a testing problem; it is a design problem that tests happen to reveal first. Paddlebook's authors already learned part of this lesson: its domain service receives its collaborators through interfaces. Chapter 6 looks at why that matters and how to get there when code was not written that way.

Bugs found early are cheaper

A bug found by a unit test on your machine costs a few minutes. The same bug found by Ruth on a printed manifest costs a confused customer and an afternoon of reading logs. We will not quote numbers for this effect, but the direction is not controversial: the later a bug is found, the more people it touches.

> **NOTE**
>
> Tests cannot prove the absence of bugs. A passing suite tells you that the behaviors you thought to check still hold. Much of the craft of testing is choosing which behaviors are worth checking, and finding ways (table tests, fuzzing, contract tests) to check more behaviors than you would think of on your own.

What a Testing Strategy Is

"We should write more tests" is a wish, not a strategy. A strategy answers four concrete questions for a particular codebase and team.

Question	What it decides	Paddlebook example
Which tests?	The kinds of test you write for each layer	Table tests for pricing; contract tests for stores
Where?	Which package and file each test lives in	Domain tests next to `internal/booking`
How many?	The balance between fast narrow tests and slow broad ones	Many unit tests, few end-to-end tests
When do they run?	On save, on every pull request, nightly	Race-detector run on every pull request

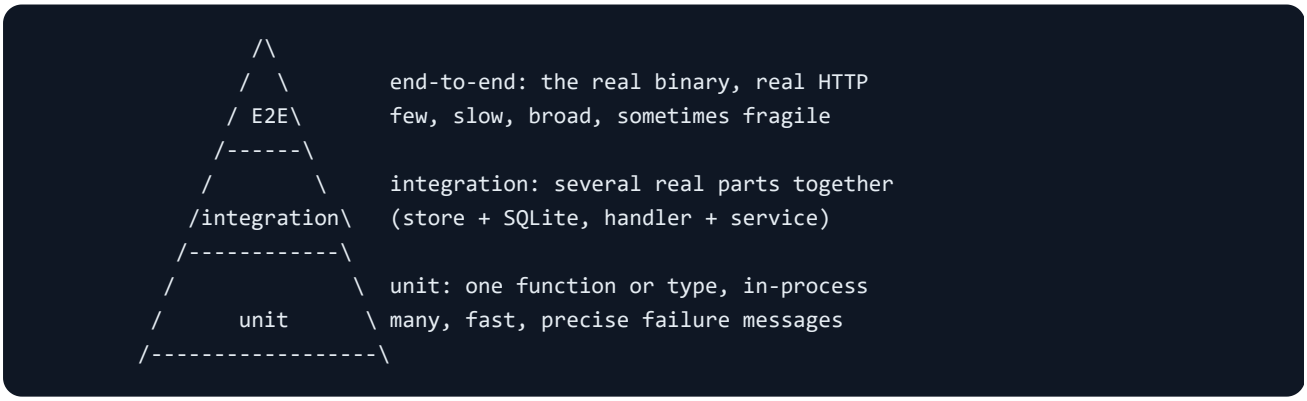
Notice that none of these questions is "how much coverage do we need?" Coverage is a useful signal, and Chapter 25 measures it carefully, but it is an outcome of a strategy rather than the strategy itself. A suite can execute every line of a function and still never check that the function returns the right answer.

A strategy also has to fit the team. Paddlebook is maintained by two developers, so a strategy that needs a farm of containers on every laptop or a forty-minute pipeline would not survive its first busy week. We will make choices a small team can sustain, and say so when a larger team might choose differently.

The Test Pyramid and Its Critics

The most widely repeated picture of a testing strategy is the test pyramid. It has many variations, but the core idea is simple: write many small, fast tests at the bottom, fewer broader tests in the middle, and only a handful of slow end-to-end tests at the top.

The classic test pyramid



The pyramid's argument is economic. Unit tests are cheap to write, run in milliseconds, and point straight at the broken function when they fail. End-to-end tests exercise everything at once, which is valuable, but they are slower, they fail for more reasons (ports, timing, build steps), and a red end-to-end test often tells you only that *something* is wrong somewhere.

The pyramid has critics, and they make good points. The "testing trophy" shape argues that integration tests deserve the widest band, because unit tests that replace every collaborator with a fake can all pass while the real pieces fail to fit together. The "honeycomb" shape, popular for microservices, argues that a service with little internal logic is best tested mostly at its boundaries, through its API with real dependencies, with only a thin layer of unit tests for the few complicated parts.

We take a practical position: the right shape depends on where the risk lives in your code. Paddlebook has real business logic (pricing with truncating integer division, refund windows, a hold lifecycle, check-digit booking codes), so it deserves a broad base of unit tests on its domain package. It also has a database layer, an HTTP API, and a client for an external weather service, where the risk lies in how the pieces fit together, so it deserves real integration tests. And it is a binary that someone deploys, so a few end-to-end tests that build and start it are worth their cost. By the end of the book the shape will be closer to a pyramid with a thick middle than to any idealized picture.

> **TIP**
> When you argue about the shape of a test suite, argue about specific risks instead of shapes. "What would break if the SQLite store stored times in local time?" leads to a better decision than "we need more integration tests."

Unit, Integration, and End-to-End in Go Terms

The words "unit", "integration", and "end-to-end" mean different things to different teams. In this book they have concrete meanings that map onto how Go code is built and run.

Kind	What runs	Typical Go tools	Paddlebook chapters
Unit	One package's code, in the test process, doubles for slow collaborators	`testing`, table tests, go-cmp	2-10
Integration	Several real packages together, in-process: handler + service + store, store + SQLite	`httptest`, SQLite, contract tests	11-14, 16-19
End-to-end	The compiled `paddlebookd` binary, started as a separate process and driven over HTTP	`os/exec`, build tags	15

Go makes the middle of this table unusually comfortable. Because an HTTP server in Go is just a value that implements `http.Handler`, you can run the whole API inside a test process, with real routing and real JSON, without opening a port. Because `database/sql` drivers are ordinary packages, a pure-Go

SQLite driver lets integration tests use a real SQL database without installing anything. Many tests that would be "end-to-end" in other ecosystems are cheap in-process integration tests in Go, and we will exploit that heavily in Parts III and IV.

All three kinds run with `go test`; later chapters use build tags and the `-short` flag to decide which ones run where.

Meet the Team

Four people from Larchmere appear in examples throughout the book. Each represents a point of view worth keeping in mind when you write tests.

- **Ines** owns Larchmere. She cares that bookings and refunds are right, and she asks the questions that turn into test names: "What if someone books exactly sixty days ahead?"
- **Malik** is the lead developer and wrote most of Paddlebook. He knows where the bodies are buried and is happy to finally have tests that let him move them.
- **Hana** joined recently and is building the test suite with you. When this book says "we", it is usually Hana and you at the keyboard.
- **Ruth** runs the dock and prints the launch manifest every morning. She owns no code but reports bugs with admirable precision, and her printed sheet becomes a golden file in Chapter 22.

A Tour of Paddlebook

Paddlebook is deliberately small enough to read in an afternoon and large enough to need every kind of test. It is a single Go module with the module path `example.com/paddlebook`, and its `go.mod` says `go 1.26.0`, so the code runs on Go 1.26 and Go 1.27. Here is the layout.

Project layout

```
paddlebook/
  go.mod                       module example.com/paddlebook; go 1.26.0
  cmd/paddlebookd/main.go      the server binary; run(ctx, args, stdout, stderr)
  internal/booking/            domain: craft, reservations, pricing, refunds,
                               booking codes, Service, Expirer
  internal/store/memstore/     in-memory Repository (mutex + maps)
  internal/store/sqlitestore/  SQLite Repository (schema.sql embedded)
  internal/weather/            client for the Brightwind Weather API
  internal/notify/             notify.Writer: prints messages to an io.Writer
  internal/manifest/           manifest.Render: the daily launch sheet (text)
  internal/httpapi/            JSON HTTP API (httpapi.New returns an http.Handler)
  internal/seed/               the fleet and the demo bookings
```

The domain package

Package `booking` is the heart of the service. It defines the `Craft` and `Reservation` types, the reservation statuses (`held`, `confirmed`, `cancelled`, `expired`), a set of sentinel errors such as `ErrSlotTaken` and `ErrHoldExpired`, and the `Service` type that enforces the business rules. The service never talks to a database, the clock, or the network directly. It receives everything through a `Deps` struct of small interfaces:

internal/booking/service.go (excerpt)

```
// Deps are the collaborators of a Service. Only Repo is required.
type Deps struct {
    Repo     Repository
    Clock    Clock        // nil: SystemClock
    Wind     WindChecker  // nil: no wind check
    Notifier Notifier     // nil: messages are dropped
    IDs      IDGenerator  // nil: RandomIDs
    Logger   *slog.Logger // nil: logs are discarded
}
```

That one struct is the reason Paddlebook is testable at all. A test can pass an in-memory repository, a clock that never moves, a wind checker that always reports a calm day, and an ID generator that hands out predictable booking codes. We will lean on this design from Chapter 4 onward and build a full set of reusable test doubles in Chapter 7.

The business rules

Larchmere's rules are simple to state and surprisingly easy to get wrong in code. They are the rules our tests will pin down.

- **Rates.** Kayak $18.00 per hour, tandem $28.00, canoe $32.00, SUP $19.50. All money is integer cents.
- **Day cap.** You never pay for more than five hours in one booking; a booking may last from one to eight hours.
- **Weekend surcharge.** Bookings that start on a Saturday or Sunday (in UTC) cost 25% more.
- **Member discount.** Members get 10% off the base price plus surcharge. Integer division truncates, so a one-hour Saturday SUP for a member costs 1950 + 487 - 243 = 2194 cents.
- **Opening hours.** Bookings start on the hour, no earlier than 08:00 UTC, and every craft is back by 20:00 UTC. Starts must not be in the past and at most sixty days ahead.
- **Holds.** A new booking is *held* for fifteen minutes. If it is not confirmed in time it expires, and a background `Expirer` marks expired holds periodically.
- **Wind.** If the weather service forecasts wind at or above a craft's limit (SUP 20 km/h, canoe 25, kayak and tandem 30), the booking is refused.
- **Refunds.** Cancelling a confirmed booking at least 24 hours before the start refunds everything; later than that refunds half, truncated; once the booking has started, it cannot be cancelled.

- **Booking codes.** Codes look like `LPC-000019`: a prefix, five base-32 digits, and a check digit that catches most typing mistakes.

The HTTP API

Package `httpapi` exposes the service as JSON over HTTP using the Go 1.22 method-and-wildcard patterns of `http.ServeMux`. Errors come back as `{"error":"...","code":"..."}` with a status code chosen from the error's type.

Route	Purpose
`GET /healthz`	Liveness check, returns `ok`
`GET /v1/craft`	The fleet as a JSON array
`GET /v1/quote`	Price a booking without reserving it
`POST /v1/reservations`	Create a hold (201 with a `Location` header)
`GET /v1/reservations/{id}`	Fetch one reservation
`POST /v1/reservations/{id}/confirm`	Confirm a live hold
`POST /v1/reservations/{id}/cancel`	Cancel, with a refund if confirmed
`GET /v1/manifest?date=`	The launch manifest (staff token required)

The rest of the service

The two stores implement the same `booking.Repository` interface: `memstore` keeps everything in maps behind a mutex, and `sqlitestore` uses the pure-Go `modernc.org/sqlite` driver, so it needs no C compiler. The `weather` package is a client for the fictional Brightwind Weather API, with retries and backoff. The `manifest` package renders Ruth's morning sheet as plain text. The `seed` package holds the eight craft of the fleet and five demo bookings for Saturday 2026-07-18, which we will use as shared sample data. And `cmd/paddlebookd` wires it all together behind command-line flags such as `-addr`, `-db`, and `-demo`.

One detail of the binary is worth noticing now: `main` does almost nothing. The real work happens in `run(ctx, args, stdout, stderr)`, which takes its arguments and output streams as parameters. Chapter 14 will show why that small decision makes the whole server testable in-process.

Running the Starter Suite

To follow along, you need Go 1.26 or later; the outputs in this book come from Go 1.27.1. Open a terminal in the project directory and run every test in every package. The pattern `./...` means "this directory and everything below it".

Terminal

```
$ go version
go version go1.27.1 windows/amd64
$ go test ./...
?       example.com/paddlebook/cmd/paddlebookd  [no test files]
ok      example.com/paddlebook/internal/booking 0.041s
ok      example.com/paddlebook/internal/httpapi 0.093s
?       example.com/paddlebook/internal/manifest    [no test files]
?       example.com/paddlebook/internal/notify  [no test files]
?       example.com/paddlebook/internal/seed    [no test files]
ok      example.com/paddlebook/internal/store/memstore  0.071s
ok      example.com/paddlebook/internal/store/sqlitestore   0.148s
?       example.com/paddlebook/internal/weather [no test files]
```

Each line is one package. `ok` means every test in the package passed; the duration is the time the package's test binary ran (on our test machine; your numbers will differ). `[no test files]` means the package has no `_test.go` files at all. Five of Paddlebook's nine packages are in that state, including the weather client and the manifest renderer that Ruth depends on every morning.

Add `-v` (verbose) to see each test by name. We also add `-count=1`, which forces a real run instead of reusing cached results; Chapter 2 explains the cache.

Terminal

```
$ go test -count=1 -v ./...
?       example.com/paddlebook/cmd/paddlebookd  [no test files]
=== RUN   TestFormatCode
--- PASS: TestFormatCode (0.00s)
=== RUN   TestParseCodeRoundTrip
--- PASS: TestParseCodeRoundTrip (0.00s)
=== RUN   TestPriceWeekdayKayak
--- PASS: TestPriceWeekdayKayak (0.00s)
PASS
ok      example.com/paddlebook/internal/booking 0.043s
=== RUN   TestHealthz
--- PASS: TestHealthz (0.00s)
PASS
ok      example.com/paddlebook/internal/httpapi 0.037s
?       example.com/paddlebook/internal/manifest    [no test files]
?       example.com/paddlebook/internal/notify  [no test files]
?       example.com/paddlebook/internal/seed    [no test files]
=== RUN   TestReserveRejectsOverlap
--- PASS: TestReserveRejectsOverlap (0.00s)
PASS
ok      example.com/paddlebook/internal/store/memstore  0.044s
=== RUN   TestOpenSeedsFleet
--- PASS: TestOpenSeedsFleet (0.15s)
PASS
ok      example.com/paddlebook/internal/store/sqlitestore   0.209s
?       example.com/paddlebook/internal/weather [no test files]
```

Six tests in four packages, all green. A quick look at coverage with `-cover` confirms the impression that green does not mean safe:

Terminal

```
$ go test -cover ./...
    example.com/paddlebook/cmd/paddlebookd      coverage: 0.0% of statements
ok      example.com/paddlebook/internal/booking 0.385s  coverage: 20.4% of statements
ok      example.com/paddlebook/internal/httpapi 0.164s  coverage: 23.3% of statements
    example.com/paddlebook/internal/manifest        coverage: 0.0% of statements
    example.com/paddlebook/internal/notify      coverage: 0.0% of statements
    example.com/paddlebook/internal/seed        coverage: 0.0% of statements
ok      example.com/paddlebook/internal/store/memstore  0.167s  coverage: 32.3% ...
ok      example.com/paddlebook/internal/store/sqlitestore   0.186s  coverage: 20.4% ...
    example.com/paddlebook/internal/weather     coverage: 0.0% of statements
```

Roughly a fifth of the domain package runs during its tests, and nothing at all runs in the weather client, the manifest, or the server binary. We will come back to what these numbers do and do not mean in Chapter 25. For now they simply quantify Malik's unease.

The Six Starter Tests and What They Miss

It is worth reading the existing tests before writing new ones, partly to learn the house style and partly to see the gaps. Here is the whole of the pricing test:

internal/booking/pricing_test.go (starter)

```
func TestPriceWeekdayKayak(t *testing.T) {
    start := time.Date(2026, time.July, 14, 10, 0, 0, 0, time.UTC) // a Tuesday
    q, err := Price(Kayak, start, 2, false)
    if err != nil {
        t.Fatalf("Price returned error: %v", err)
    }
    if q.TotalCents != 3600 {
        t.Errorf("TotalCents = %d, want 3600", q.TotalCents)
    }
}
```

This is a perfectly good test. It uses a fixed date in UTC, it stops early with `t.Fatalf` if there is an error, and its failure message says what it got and what it wanted. Its only problem is that it is alone. The table below lists all six starter tests and the most important things each area leaves untested.

Starter test	What it checks	What nobody checks
`TestFormatCode`	Code for 1 is `LPC-000019`	Other values, wrap-around at the maximum
`TestParseCodeRoundTrip`	Six codes parse back	Lower case, aliases, bad input
`TestPriceWeekdayKayak`	One weekday kayak price	Weekends, members, truncation, day cap
`TestHealthz`	`/healthz` says `ok`	Every other route and error mapping

`TestReserveRejectsOverlap`	memstore rejects an overlap	Whether SQLite behaves the same
`TestOpenSeedsFleet`	SQLite store opens and seeds	Reservations, time round trips

The gaps fall into recognizable categories, and each category has a chapter. Refunds and most of the pricing rules have no tests at all (Chapters 2 and 3). The service's booking flow, with its holds, confirmations, and cancellations, is untested (Chapters 4 and 7). The weather client's retries have never been exercised (Chapter 13). The two stores have never been checked against each other (Chapter 17). The expirer runs in a goroutine on a timer, which is notoriously hard to test without making the suite slow and flaky (Chapters 20 and 21). And the booking-code parser accepts input from customers, which means it accepts input from people who will type things the developer never imagined; Chapter 23 lets a fuzzer imagine them for us.

A First Surprise: Time Is a Dependency

Before writing a single test, Hana does what every new developer does: she runs the demo server and pokes it. The `-demo` flag loads the fleet and the five demo bookings for Saturday 2026-07-18. Fetching Ada's demo booking works fine:

Terminal

```
$ go build -o paddlebookd ./cmd/paddlebookd
$ ./paddlebookd -demo
paddlebookd listening on http://127.0.0.1:8080

$ curl -s http://127.0.0.1:8080/v1/reservations/lpc-000019
{"id":"LPC-000019","craft_id":"K-01","customer":"Ada Brennan","email":"ada@example.com",
"start":"2026-07-18T09:00:00Z","hours":2,"member":true,"status":"confirmed",
"total_cents":4050,"refund_cents":0,"hold_until":"2026-07-14T09:15:00Z",
"created_at":"2026-07-14T09:00:00Z"}
```

(We ran the server in a second terminal and wrapped the JSON for print; the server sends it on one line. On Windows, name the output `paddlebookd.exe`.) Notice that the lowercase `lpc-000019` in the URL was accepted: the service normalizes booking codes. Then Hana asks for a quote for Mei's Saturday paddleboard slot, using the same date as the demo data:

Terminal

```
$ curl -s \
"http://127.0.0.1:8080/v1/quote?craft=S-01&start=2026-07-18T13:00:00Z&hours=1&member=true"
{"error":"invalid start: must not be in the past","code":"invalid"}
```

Nothing is broken. The server uses the real clock, and when we ran this command the demo Saturday was already in the past. But this tiny surprise contains one of the most important lessons in the book. The answer to "what does a quote for Saturday cost?" depends on *when you ask*. Any test that calls the

real clock gives different answers on different days, and a test that passes in July will fail in October without a single line of code changing.

Paddlebook's design already anticipates this: the service reads time through the `Clock` interface, and tests can supply a clock that always says "Tuesday 2026-07-14, 09:00 UTC". Every test in this book that involves the service uses such a fixed "now", so that the canonical numbers (a $40.50 Saturday kayak, a $20.25 late refund) come out the same on every machine, on every day, forever.

> **WARNING**
> A test that depends on the real current time, the machine's time zone, random numbers, the network, or the order of map iteration is a flaky test waiting for its day. Each of these is a hidden input. Chapter 9 and Chapter 26 are devoted to finding and removing them, but the cheapest fix is never to introduce them: pass time, randomness, and I/O in as dependencies from the start.

Paddlebook's Testing Policy

Before writing tests, the team agrees on a short policy. These are *our choices* for this codebase, not universal laws, and each one is argued for in the chapter that puts it into practice. Writing them down early matters, because a policy that is decided case by case in code review tends not to be decided at all.

1. **Every bug fix starts with a failing test.** The test reproduces the bug, fails for the right reason, and then passes with the fix. It stays in the suite forever as a guard.
2. **Hand-written fakes for our own interfaces.** We build small, readable doubles in a `bookingtest` package (Chapter 7). Generated mocks (gomock, from `go.uber.org/mock`) are used only where the interaction order or arguments are the point, such as "notify exactly once with this subject" (Chapter 8).
3. **No `time.Sleep` to wait for things.** We inject a `Clock`, use `testing/synctest`, or wait on an explicit signal. Sleeping makes tests both slow and flaky.
4. **`go vet ./...` and `go test -race ./...` on every pull request.** Integration tests (`-tags integration`) and end-to-end tests (`-tags e2e`) run on the main branch and nightly, and every fuzz target gets one minute of fuzzing per night.
5. **Flaky tests are quarantined within one day** with `t.Skip("flaky: PB-123")` pointing at an issue, and fixed or deleted within two weeks.
6. **Coverage is a signal, not a target.** In Chapter 25 we set a floor below the value we actually measure, to catch accidental drops without rewarding tests that execute code without checking it.