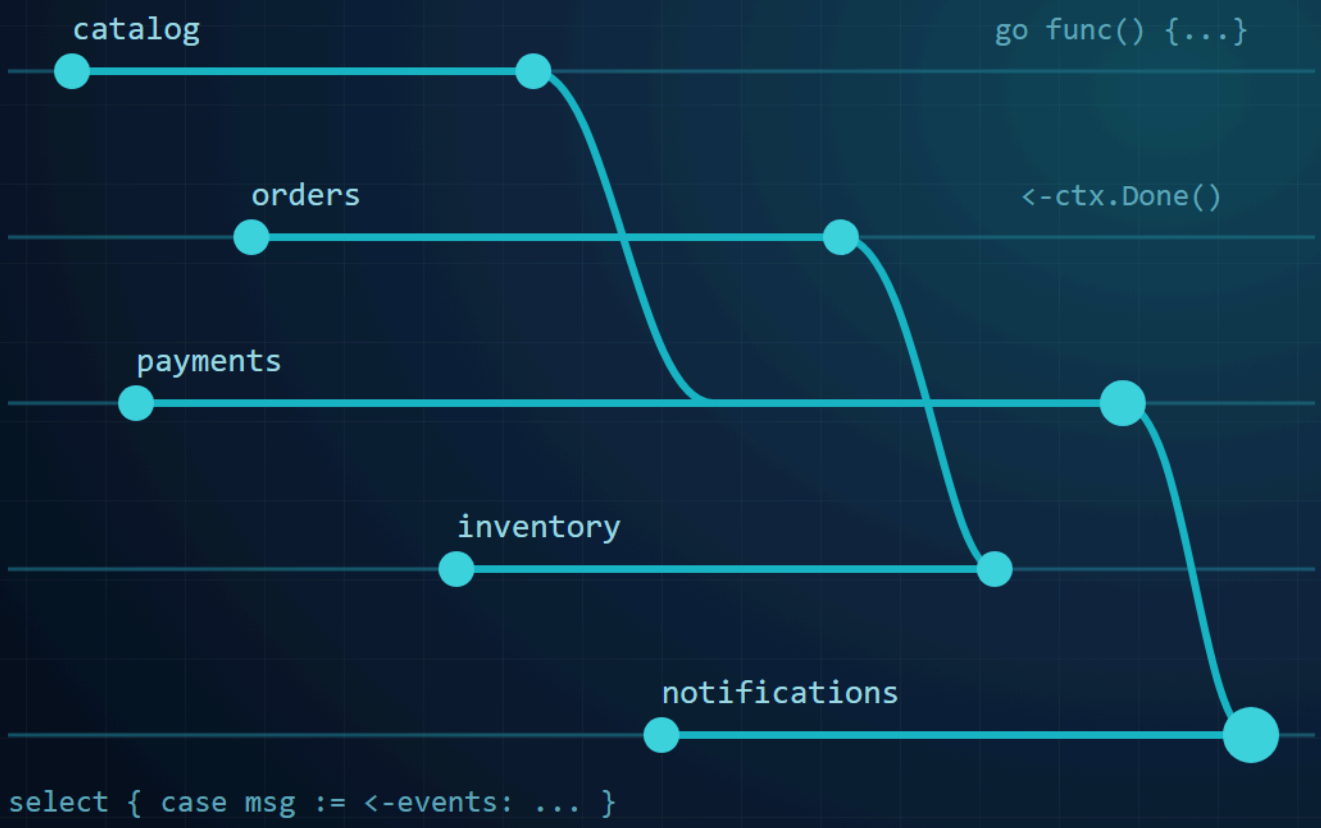


THE COMPLETE PRACTICAL GUIDE

Go (Golang) for Backend Developers

Build Fast, Reliable, Production-Ready Services

Concurrency · HTTP & gRPC APIs · PostgreSQL & Redis · Observability · Kubernetes · Architecture

28 CHAPTERS · 4 APPENDICES · ORBIT STORE CASE STUDY

THENEXBOOKS

Go (Golang) for Backend Developers

Build Fast, Reliable, Production-Ready Services with Go

From Language Fundamentals to Concurrency, APIs, Data, and Operations

First Edition

THENEXBOOKS

Go (Golang) for Backend Developers
Build Fast, Reliable, Production-Ready Services with Go

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 standard library, and third-party libraries evolve; always consult the official documentation for the versions you use.

Go and the Go logo are trademarks of Google LLC. PostgreSQL, Redis, Kafka, NATS, Docker, Kubernetes, gRPC, OpenTelemetry, Prometheus, and 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 not affiliated with or endorsed by Google or the Go project.

Orbit Store, the running example in this book, is fictional. Any resemblance to real companies or products is coincidental.

Preface

Go was designed for exactly the kind of software most backend teams write every day: networked services that must be fast, stay up, and remain understandable as the team and the codebase grow. It compiles to a single static binary, starts in milliseconds, ships with a production-quality HTTP server in its standard library, and makes concurrency a first-class, readable part of the language rather than a library afterthought.

Yet many developers arrive at Go from Java, C#, Python, Node.js, or PHP and find that their instincts do not quite fit. There are no classes, no exceptions, no annotations, and no framework that decides how your application is structured. Errors are values. Interfaces are satisfied implicitly. Goroutines are cheap, but leaking them is easy. The language is small, and the idioms carry a lot of weight.

This book is a complete, practical guide to building backend services in Go — from your first module to a production system with databases, caches, message queues, observability, and a deployment pipeline. We will not stop at syntax. You will learn *why* Go code is written the way it is, how to structure services that stay maintainable, and what actually goes wrong in production.

Who This Book Is For

- **Backend developers** moving to Go from another language who want to become productive and idiomatic quickly.
- **Go developers** who know the language but want a complete picture of production service development: data, messaging, security, observability, and deployment.
- **Tech leads and architects** evaluating Go for new services or planning a migration from a monolith.
- **Platform and DevOps engineers** who build internal tools, operators, and APIs in Go.

What You Will Learn

- The Go language from a backend perspective: types, interfaces, errors, generics, and the standard library toolbox.
- Testing as a daily habit: table-driven tests, subtests, fuzzing, benchmarks, and integration tests.
- Concurrency done right: goroutines, channels, `select`, `context`, synchronization, and proven patterns.
- HTTP and RPC APIs with `net/http`, middleware, JSON, validation, authentication, and gRPC.
- Data and messaging: PostgreSQL with `database/sql`, pgx, and sqlc; Redis caching; events, the outbox pattern, and background jobs.
- Production readiness: structured logging, metrics, tracing, profiling, resilience, security, containers, Kubernetes, and CI/CD.

- Architecture for large codebases, and two complete case studies — including an AI shopping assistant backend.

How This Book Is Organized

Part I — Go Foundations for Backend Engineers (Chapters 1–6) teaches the language and its tooling with a backend focus. **Part II — Concurrency** (Chapters 7–10) covers goroutines, channels, `context`, synchronization, and patterns. **Part III — Building HTTP and RPC APIs** (Chapters 11–16) builds real APIs step by step. **Part IV — Data and Messaging** (Chapters 17–20) connects services to PostgreSQL, Redis, and message brokers. **Part V — Production Readiness** (Chapters 21–25) addresses observability, performance, resilience, security, and deployment. **Part VI — Architecture and Case Studies** (Chapters 26–28) brings everything together. The appendices provide a cheat sheet, a service template, a production checklist, and a glossary.

Each chapter ends with a **Key Takeaways** box for quick review.

The Running Example: Orbit Store

Throughout the book we build **Orbit Store**, a fictional e-commerce marketplace. Merchants sell products with variants and inventory; customers browse, review, and place orders; payments, shipments, and notifications flow through the system. Orbit Store is built as a set of Go services — `catalog`, `orders`, `payments`, `inventory`, `notifications`, and an API `gateway` — backed by PostgreSQL, Redis, and a message broker.

Conventions Used in This Book

> **TIP**
> A recommended practice or a technique that makes your work easier.

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

> **WARNING**
> A common mistake or a behavior that can cause bugs, data loss, or outages in production.

Code appears in `monospace`. Examples target recent versions of Go; where a feature depends on a specific release, we say so. Features that are still proposals or experiments are labeled as such.

Contents

Preface

PART I — GO FOUNDATIONS FOR BACKEND ENGINEERS

1 Why Go for Backend Development

2 Tooling, Modules, and Project Layout

3 Types, Structs, Methods, and Interfaces

4 Errors Done Right: Wrapping, Sentinels, and Custom Types

5 Generics, Collections, and the Standard Library Toolbox

6 Testing Fundamentals: Table Tests, Subtests, Fuzzing, and Benchmarks

PART II — CONCURRENCY

7 Goroutines, Channels, and select

8 The context Package: Cancellation, Deadlines, and Request Scope

9 Synchronization: sync, atomic, and the Go Memory Model

10 Concurrency Patterns: Worker Pools, Pipelines, Fan-Out/Fan-In, and errgroup

PART III — BUILDING HTTP AND RPC APIS

11 net/http in Depth: Servers, Handlers, and the Enhanced ServeMux

12 Middleware, Routing Libraries, and the Request Lifecycle

13 JSON, Validation, and REST API Design

14 Authentication and Authorization

15 gRPC and Protocol Buffers

16 Configuration and Structured Logging with log/slog

PART IV — DATA AND MESSAGING

17 Working with PostgreSQL: database/sql and pgx

18 Transactions, Migrations, and Type-Safe SQL with sqlc

19 Caching with Redis

20 Messaging, Events, and Background Jobs

PART V — PRODUCTION READINESS

21 Observability: Metrics, Tracing, and OpenTelemetry

22 Performance: Profiling with pprof, Memory, Allocations, and the GC

23 Resilience: Timeouts, Retries, Rate Limiting, Circuit Breakers, Graceful Shutdown

24 Security for Go Services

25 Containers, Kubernetes, and CI/CD for Go

PART VI — ARCHITECTURE AND CASE STUDIES

26 Structuring Large Go Codebases: Packages, Boundaries, and Hexagonal Design

27 Case Study: Building the Orbit Store Order Service End to End

28 Case Study: Adding an AI Shopping Assistant Backend in Go

APPENDICES

A Go Cheat Sheet

B Service Template and Reference

C Production Checklist

D Glossary

PART I

Go Foundations for Backend Engineers

The language, its tooling, and its idioms — seen through the eyes of a backend developer. Types, interfaces, errors as values, generics, and a testing habit you will use in every chapter that follows.

1 · Why Go for Backend Development
2 · Tooling, Modules, and Project Layout
3 · Types, Structs, Methods, and Interfaces
4 · Errors Done Right
5 · Generics, Collections, and the Standard Library
6 · Testing Fundamentals

CHAPTER 1 · PART I — GO FOUNDATIONS FOR BACKEND ENGINEERS

Why Go for Backend Development

Go was built by engineers who were tired of slow builds, tangled dependencies, and concurrency that was hard to get right. In this chapter we look at what Go is good at, where it is not the best choice, and we meet Orbit Store, the fictional marketplace we will build throughout this book.

The Problem Go Was Designed to Solve

Every programming language is an answer to a question. For Go, the question was roughly this: how do you let large teams build and maintain networked server software quickly, safely, and for many years? Go was created at Google by Robert Griesemer, Rob Pike, and Ken Thompson, announced publicly as an open source project in 2009, and reached its stable 1.0 release in 2012. The people who designed it were not trying to invent a research language. They were trying to remove the day-to-day friction they experienced when writing server software in the mainstream languages of the time.

That friction had a few recognizable shapes. Builds of large codebases took a long time, which slowed down the feedback loop between writing code and seeing whether it worked. Dependency graphs were difficult to reason about, so a change in one corner of the codebase could trigger a rebuild of half the world. Concurrency was available, but it usually meant working directly with threads, locks, and callbacks, which made correct code difficult to write and even harder to review. And codebases grew so large that readability for newcomers mattered more than expressiveness for experts.

Go's answer was deliberately conservative. It has a small language specification that you can read in an afternoon. It compiles to native machine code quickly. It has garbage collection, so you do not manage memory by hand. It has concurrency primitives — goroutines and channels — built directly into the language rather than bolted on as a library. And it ships with an opinionated toolchain that formats, tests, documents, and builds your code the same way on every machine.

If you are coming from Java, C#, Python, Node.js, or PHP, some of these choices will feel refreshing and some will feel restrictive at first. That is normal. Much of this book is about showing you *why* Go does things the way it does, so that the restrictions start to feel like guard rails rather than obstacles.

What a Go Service Looks Like

Before we talk about strengths and trade-offs in the abstract, let us look at something concrete. Here is a complete, runnable HTTP service that exposes a health check for the Orbit Store catalog service. It uses nothing but the standard library.

cmd/catalog/main.go

```
package main

import (
    "encoding/json"
    "log"
    "net/http"
    "time"
)

type healthResponse struct {
    Status  string    `json:"status"`
    Service string    `json:"service"`
    Time    time.Time `json:"time"`
}

func main() {
    mux := http.NewServeMux()
    mux.HandleFunc("GET /healthz", func(w http.ResponseWriter, r *http.Request) {
        w.Header().Set("Content-Type", "application/json")
        resp := healthResponse{Status: "ok", Service: "catalog", Time: time.Now().UTC()}
        if err := json.NewEncoder(w).Encode(resp); err != nil {
            log.Printf("encode health response: %v", err)
        }
    })

    srv := &http.Server{
        Addr:              ":8080",
        Handler:           mux,
        ReadHeaderTimeout: 5 * time.Second,
    }

    log.Printf("catalog listening on %s", srv.Addr)
    if err := srv.ListenAndServe(); err != nil {
        log.Fatalf("server stopped: %v", err)
    }
}
```

There are several things worth noticing even at this early stage. There is no framework and no dependency injection container. There is no annotation processing or code generation step. The route pattern `"GET /healthz"` includes the HTTP method, a feature of the standard library's `http.ServeMux` since Go 1.22. The server is configured with a header read timeout, because a production server without timeouts is an invitation for slow clients to exhaust your resources — we return to this topic in Chapter 11 and Chapter 23.

You run this program with `go run ./cmd/catalog`, and you build a deployable binary with `go build`. The result is a single executable file. You can copy it into a minimal container image or onto a virtual machine and run it without installing a runtime, an interpreter, or a package manager on the target machine. For many teams, this property alone changes how they think about deployment.

> **NOTE**
>
> The `log.Fatalf` call at the end is acceptable for a tiny example, but real services need graceful shutdown so in-flight requests can finish when the process receives a termination signal. We build a proper shutdown sequence in Chapter 23.

Go's Core Strengths for Backend Work

Many languages can serve HTTP requests and talk to a database. What makes Go particularly well suited to backend services is not a single feature but the way a set of modest features reinforce each other. Let us walk through them one at a time.

Simplicity and Readability

Go has a small number of keywords and very few ways to express the same idea. There is one loop construct, `for`. There are no classes, no inheritance, no exceptions, no operator overloading, and until Go 1.18 there were no generics. Code formatting is not a matter of taste because `gofmt` decides it for everyone. The practical effect is that Go code written by one team looks very much like Go code written by another team.

This matters more in backend work than it might seem. Services live for years. They are maintained by people who did not write them, often while an incident is in progress. When you open an unfamiliar Go repository at two in the morning to find out why orders are failing, you do not need to learn that team's private dialect of the language first. You can read the code top to bottom and see what it does. Explicit error handling, which we cover in depth in Chapter 4, is a big part of this: the failure paths are visible in the code rather than hidden in an exception hierarchy.

Simplicity has costs, and it would be dishonest to pretend otherwise. Go code can be more verbose than the equivalent in a language with more abstraction features. You will write `if err != nil` many times. You will occasionally write a loop where another language would offer a one-line expression. Most Go developers come to accept this trade: a little more typing in exchange for code that is easy to read, review, and change.

Fast Compilation and a Single Binary

Go was designed so that the compiler can work quickly. The dependency rules are strict — unused imports are compile errors, and import cycles between packages are forbidden — which keeps the dependency graph clean and lets the build system avoid unnecessary work. The build cache means that rebuilding after a small change is usually fast, even in large repositories. We will not quote numbers here, because build time depends on your code, your dependencies, and your hardware; the best approach is to try it on your own project and see.

The output of a Go build is a statically linked native executable by default for pure Go code. Cross-compiling is built in: set the `GOOS` and `GOARCH` environment variables and the same toolchain produces a Linux binary on your laptop, or an ARM binary for a cloud instance. Chapter 2 covers the build flags you will use in practice, and Chapter 25 turns those binaries into small container images.

Concurrency Built Into the Language

Backend services spend most of their time waiting — for the database, for another service, for the network. Go's answer to this is the goroutine, a lightweight function execution managed by the Go runtime rather than directly by the operating system. You start one with the `go` keyword. Goroutines are cheap enough to create that the standard library's HTTP server starts one for every incoming connection, and your code inside a handler is written in a plain, sequential, blocking style.

This is a significant difference from callback-based or `async` / `await` models. In Go, there is no split between "sync" and "async" functions. A function that reads from the database simply blocks its goroutine until the data arrives, while the runtime schedules other goroutines onto the available operating system threads. You get the throughput benefits of non-blocking I/O without restructuring your code around it.

Here is a small example from the Orbit Store catalog. When a client asks for a product page, we want to load the product details and the current stock level at the same time, since they come from different services.

internal/catalog/page.go

```
package catalog

import (
    "context"
    "fmt"
)

type Product struct {
    ID         string
    Name       string
    PriceCents int64
    Currency   string
}

type ProductStore interface {
    Product(ctx context.Context, id string) (Product, error)
}

type StockChecker interface {
    Available(ctx context.Context, productID string) (int, error)
}

type ProductPage struct {
    Product   Product
    Available int
}
```

```
// LoadPage fetches product details and stock level concurrently.
func LoadPage(ctx context.Context, ps ProductStore, sc StockChecker, id string) (
    ProductPage, error,
) {
    type stockResult struct {
        n   int
        err error
    }
    stockCh := make(chan stockResult, 1)

    go func() {
        n, err := sc.Available(ctx, id)
        stockCh <- stockResult{n: n, err: err}
    }()

    p, err := ps.Product(ctx, id)
    if err != nil {
        return ProductPage{}, fmt.Errorf("load product %s: %w", id, err)
    }

    res := <-stockCh
    if res.err != nil {
        return ProductPage{}, fmt.Errorf("check stock for %s: %w", id, res.err)
    }
    return ProductPage{Product: p, Available: res.n}, nil
}
```

Do not worry about every detail yet. The shape is what matters: we start a goroutine to check stock, load the product on the current goroutine, then receive the stock result from a channel. The channel is buffered with a capacity of one, so the goroutine can always deliver its result and exit, even if we return early because the product lookup failed. That small detail prevents a goroutine leak — one of the most common concurrency bugs in production Go code. Part II of this book, starting with Chapter 7, covers goroutines, channels, `context`, and patterns such as `errgroup` that make this kind of code shorter and safer.

A Batteries-Included Standard Library

Go's standard library is one of its most underrated features. For backend work, it includes a production-grade HTTP client and server (`net/http`), TLS (`crypto/tls`), JSON encoding (`encoding/json`), a database abstraction (`database/sql`), structured logging (`log/slog`, added in Go 1.21), templating, compression, cryptographic primitives, and a testing framework (`testing`). Since Go 1.21 it also includes generic helper packages such as `slices` and `maps`.

In practice, this means a Go service often has a surprisingly short list of third-party dependencies. You typically add a database driver such as pgx, perhaps a router such as chi, an observability library, and a few focused utilities. There is no need for a large framework to get started. Fewer dependencies mean fewer supply-chain risks, fewer version conflicts, and fewer surprises when you upgrade.

Performance That Is Good by Default

Go compiles to native code, has value types that let you control memory layout, and has a garbage collector designed for low pause times. For typical backend workloads — parsing requests, calling databases, serializing responses — Go services tend to use modest amounts of memory and CPU without much tuning. We are deliberately not quoting benchmark numbers here: published comparisons between languages are notoriously sensitive to the workload and to how much effort went into each implementation. What we can say is that Go gives you the tools to measure your own service precisely, with built-in benchmarks (Chapter 6) and the `pprof` profiler (Chapter 22).

The more important performance property for most teams is predictability. Go's performance characteristics are relatively easy to reason about. There is no just-in-time compiler warming up after deployment, and memory usage is mostly proportional to the data you actually hold. When something is slow, the profiler usually points to your code rather than to a mystery inside the runtime.

Tooling That Everyone Shares

The `go` command is a single entry point for building, testing, formatting, vetting, documenting, and managing dependencies. Every Go developer uses the same commands: `go build`, `go test`, `go vet`, `go mod tidy`. The test runner includes a race detector (`go test -race`) that finds data races at runtime, code coverage reporting, benchmarks, and fuzzing. The official language server, `gopls`, powers editor features in VS Code, GoLand, Neovim, and others.

Shared tooling has a social effect as well as a technical one. There are no arguments about formatting in code review, no need to agree on a test framework before writing the first test, and no build system to learn when you join a new team. Chapter 2 is a detailed tour of this toolchain.

The Compatibility Promise

When Go 1.0 was released, the Go team published the Go 1 compatibility promise: programs written to the Go 1 specification should continue to compile and run correctly with future Go 1.x releases, with narrow exceptions such as security fixes and bugs in the specification. In practice, upgrading to a new Go version is usually a matter of changing one line in your `go.mod` file and running your tests. For a service you expect to maintain for many years, this stability is enormously valuable.

> **TIP**
>
> Keep your services on a currently supported Go release. The Go project supports the two most recent major releases with security fixes, so falling far behind means missing patches. Because of the compatibility promise, staying current is usually cheap — make upgrading part of your regular maintenance rather than a yearly project.

Go Compared to the Languages You Know

Most readers of this book already build backend services in another language. The table below is a rough orientation rather than a verdict. Every row hides nuance, and every language on this list is used successfully for serious backend systems.

Topic	Go	What you may be used to
Deployment	One native binary	JVM, CLR, interpreter, or Node runtime
Concurrency	Goroutines and channels	Threads, thread pools, async/await, event loop
Error handling	Errors as returned values	Exceptions with try/catch
Type system	Static, structural interfaces	Nominal classes or dynamic typing
Code reuse	Composition and embedding	Inheritance hierarchies, mixins
Frameworks	Standard library first	Spring, ASP.NET, Django, Express, Laravel
Formatting	gofmt, not negotiable	Team-specific style configs

If you come from **Java or C#**, the biggest adjustment is the absence of a large framework and of class hierarchies. You will build services by wiring small structs together explicitly in `main`, and you will find that interfaces are defined by the code that uses them rather than the code that implements them. Chapter 3 explains this idea carefully, because it changes how you design packages.

If you come from **Python or PHP**, the biggest adjustment is static typing and compilation. The compiler will reject code you might have expected to work, and you will spend a little more time declaring types. In exchange, whole categories of runtime surprises disappear, and refactoring becomes much safer.

If you come from **Node.js**, the biggest adjustment is concurrency. In Node.js, a single event loop runs your JavaScript and you must avoid blocking it. In Go, many goroutines run in parallel across all CPU cores, and blocking a goroutine is normal. That power comes with responsibility: shared memory accessed by multiple goroutines must be synchronized, a topic we cover in Chapter 9.

Where Go Is Not the Best Fit

Choosing a language honestly means knowing its weak spots. Go is a general-purpose language, but some jobs are better served by other tools.

- **Data science and machine learning research.** The Python ecosystem for numerical computing, notebooks, and model training is far richer. Go is a good choice for the services *around* models — serving, orchestration, data pipelines — and Chapter 28 builds exactly that kind of backend. Training models in Go is uncommon.

- **Hard real-time or extremely latency-sensitive systems.** Go's garbage collector is designed for short pauses, but it is still a garbage collector. If you need strict guarantees about every microsecond, languages without a garbage collector, such as Rust, C, or C++, give you more control.
- **Rich domain modeling with advanced type features.** Go has no sum types, no pattern matching, and a deliberately limited generics system. If your domain benefits heavily from those features, languages such as Kotlin, Scala, F#, or Rust may express it more directly.
- **Rapid prototyping of CRUD applications with an admin UI.** Full-stack frameworks like Django, Rails, or Laravel generate a lot of functionality for free. Go asks you to assemble more of it yourself, which pays off as systems grow but may slow you down for a quick internal tool.

Go shines brightest in the space this book is about: network services, APIs, background workers, command-line tools, and infrastructure software. Much of the cloud-native ecosystem, including Docker, Kubernetes, Prometheus, and Terraform, is written in Go, and that is not an accident. The language's strengths line up closely with what that kind of software needs.

> **WARNING**
>
> Do not rewrite a working system in Go just because Go is well suited to backend work. Rewrites are expensive and risky. A more reliable path is to write a new, well-bounded service in Go — a notification worker or an internal API — and let the team build experience before making bigger decisions.

Meet Orbit Store

Examples in isolation teach syntax. Examples that connect to each other teach design. Throughout this book, we build pieces of **Orbit Store**, a fictional e-commerce marketplace where independent merchants sell products to customers. It is realistic enough to raise the problems you face in production — money, inventory, concurrency, failures between services — without drowning in business rules.

The Domain

Orbit Store's core concepts will be familiar if you have ever built or used an online shop:

- **User** and **Customer** — people who sign in and buy things.
- **Merchant** — a seller on the marketplace. Merchants are tenants: each one sees only its own products and orders.
- **Product**, **ProductVariant**, and **Category** — what is for sale. A variant is a specific SKU, such as a T-shirt in size M and color blue.
- **Review** — customer feedback on a product.
- **Cart**, **Order**, and **OrderLine** — what a customer intends to buy and has bought.

- **Payment** and **Shipment** — money moving and parcels moving.
- **Inventory** — how many units of each variant are available.
- **Notification** — emails and push messages sent to customers and merchants.

One rule applies everywhere: money is stored as an integer number of minor units — cents for US dollars — together with a currency code. We never use floating-point numbers for money, because binary floating point cannot represent most decimal fractions exactly, and small rounding errors add up across millions of orders.

The Services

Orbit Store is split into a handful of Go services. The split is intentionally modest; Chapter 26 discusses how to decide where service boundaries belong, and why fewer, well-structured services often beat many tiny ones.



- **gateway** is the public entry point. It authenticates requests and routes them to internal services.
- **catalog** owns products, variants, categories, and search.
- **orders** owns carts, checkout, and the order lifecycle.
- **payments** talks to a fictional external payment provider and records payment attempts.
- **inventory** tracks stock levels and reservations.
- **notifications** runs background workers that send emails and push notifications.

PostgreSQL is the main data store, Redis is used for caching, and a message broker carries events such as `order.placed` between services. We discuss the broker generically — the patterns apply whether you use Kafka, NATS, or a managed cloud offering. All code lives in a single Go module with the fictional path `github.com/orbitstore/orbit`.

A First Taste of Orbit Store Code

To close the loop between the ideas above and real code, here is a slightly larger slice of the catalog service: a product type, an in-memory store, and an HTTP handler that returns a product as JSON. Later chapters replace the in-memory store with PostgreSQL, add validation, logging, and tests, but the structure you see here will survive those changes.

internal/catalog/product.go

```
package catalog

import (
    "errors"
    "sync"
)

// ErrNotFound is returned when a product does not exist.
var ErrNotFound = errors.New("catalog: product not found")

// Item is a product as exposed by the catalog API.
type Item struct {
    ID         string `json:"id"`
    MerchantID string `json:"merchant_id"`
    Name       string `json:"name"`
    PriceCents int64  `json:"price_cents"`
    Currency   string `json:"currency"`
}

// MemoryStore keeps products in memory. It is safe for concurrent use.
type MemoryStore struct {
    mu    sync.RWMutex
    items map[string]Item
}

func NewMemoryStore(items ...Item) *MemoryStore {
    s := &MemoryStore{items: make(map[string]Item, len(items))}
    for _, it := range items {
        s.items[it.ID] = it
    }
    return s
}

func (s *MemoryStore) Get(id string) (Item, error) {
    s.mu.RLock()
    defer s.mu.RUnlock()
    it, ok := s.items[id]
    if !ok {
        return Item{}, ErrNotFound
```

```
    }
    return it, nil
}
```

internal/catalog/http.go

```
package catalog

import (
    "encoding/json"
    "errors"
    "log"
    "net/http"
)

type Handler struct {
    Store *MemoryStore
}

func (h *Handler) Register(mux *http.ServeMux) {
    mux.HandleFunc("GET /products/{id}", h.getProduct)
}

func (h *Handler) getProduct(w http.ResponseWriter, r *http.Request) {
    id := r.PathValue("id")
    item, err := h.Store.Get(id)
    switch {
    case errors.Is(err, ErrNotFound):
        http.Error(w, "product not found", http.StatusNotFound)
        return
    case err != nil:
        log.Printf("get product %q: %v", id, err)
        http.Error(w, "internal error", http.StatusInternalServerError)
        return
    }
    w.Header().Set("Content-Type", "application/json")
    if err := json.NewEncoder(w).Encode(item); err != nil {
        log.Printf("encode product %q: %v", id, err)
    }
}
```

A few Go idioms appear here that we will study in detail later. `ErrNotFound` is a *sentinel error*, a package-level value that callers compare against with `errors.Is` (Chapter 4). The struct tags such as `` `json:"price_cents"` `` control how fields are named in JSON (Chapter 13). The `sync.RWMutex` protects the map because the HTTP server calls handlers from many goroutines at once, and Go maps are not safe for concurrent writes (Chapter 9). And `r.PathValue("id")` reads a wildcard from the route pattern, another feature of the enhanced `ServeMux` introduced in Go 1.22 (Chapter 11).

Notice also what is *not* here. There is no base controller class, no annotation declaring the route, no framework-managed lifecycle. The handler is a plain struct with a dependency, and `main` would wire it up in a few lines:

cmd/catalog/main.go (excerpt)

```
store := catalog.NewMemoryStore(
    catalog.Item{ID: "p-100", MerchantID: "m-1", Name: "Orbit Mug",
        PriceCents: 1299, Currency: "USD"},
)
h := &catalog.Handler{Store: store}

mux := http.NewServeMux()
h.Register(mux)
```

This explicit wiring is sometimes called "manual dependency injection". It feels verbose the first time, but it has a large benefit: you can read `main` and see exactly how your service is assembled. There is no hidden container deciding which implementation you get.

Mindset Shifts for Experienced Developers

Learning Go's syntax takes a few days. Learning to *think* in Go takes longer, because several habits that serve you well in other languages work against you here. These are the shifts we see experienced developers struggle with most often.

Errors Are Values, Not Exceptions

Go functions that can fail return an `error` as their last result, and the caller checks it. There is no `try` / `catch`. This feels repetitive at first, but it makes every failure path explicit and forces you to decide, at each step, whether to handle an error, add context to it, or pass it up. Go does have `panic`, but it is reserved for programming bugs and truly unrecoverable situations, not for ordinary failures like "product not found". Chapter 4 is devoted to doing this well.

Composition Instead of Inheritance

Go has no classes and no inheritance. You build types by composing smaller types, embedding one struct in another when it makes sense, and describing behavior with small interfaces. Instead of a `BaseRepository` class with twenty methods that subclasses override, you write small structs that each do one job and combine them. Chapter 3 shows how.

Small Interfaces, Defined by the Consumer

In Java or C#, a class declares which interfaces it implements. In Go, a type satisfies an interface automatically if it has the right methods. This lets the *consumer* of a dependency declare the minimal interface it needs. In the `LoadPage` example earlier, the catalog package declared `StockChecker` with a single method; the inventory client does not need to know that interface exists. This style leads to small, focused interfaces and code that is easy to test with fakes.