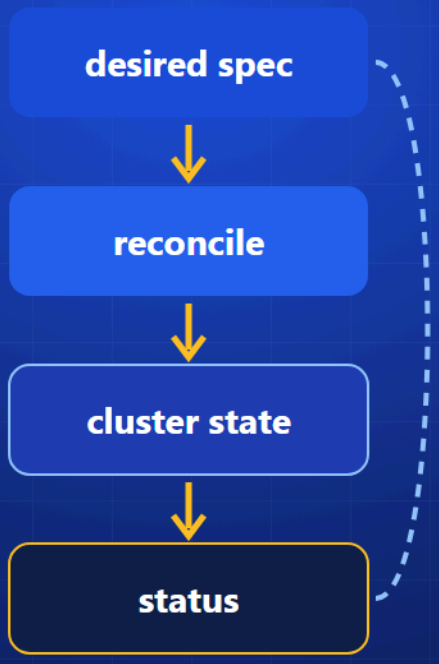
storefront_controller.go

```go
func (r *StorefrontReconciler)
  Reconcile(ctx, req) (ctrl.Result, error) {
  var sf platformv1.Storefront
  if err := r.Get(ctx, key, &sf); ...
  r.reconcileDeployment(ctx, &sf)
  r.reconcileService(ctx, &sf)
  setConditions(&sf, dep, routeErr)
  return ctrl.Result{}, r.Status().Patch(...)
}

// observe · compare · act · report
```




```
$ kubectl get storefronts -n shops
NAME            PLAN         REPLICAS   READY   URL
marlow-books    growth       3          True    https://marlow-books...
quarry-cycles   enterprise   5          True    https://quarry-cycles...
```

A PRACTICAL GUIDE FOR GO DEVELOPERS

Kubernetes Operators and Controllers in Go

Build production-ready operators with client-go, controller-runtime, CRDs, webhooks, and envtest

CRDs · CEL validation · reconcile loops · finalizers
webhooks · conversion · testing · RBAC · upgrades

API MACHINERY · CONTROLLERS · TESTING · PRODUCTION

THENEXBOOKS

Kubernetes Operators and Controllers in Go

Build Production-Ready Operators with client-go, controller-runtime, CRDs, Webhooks, and envtest
From the API machinery to tested, observable, upgradeable operators

First Edition

THENEXBOOKS

Kubernetes Operators and Controllers in Go
Build Production-Ready Operators with client-go, controller-runtime, CRDs, Webhooks, and envtest

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. Kubernetes, Go, and the libraries discussed in this book evolve quickly; always consult the official documentation for the versions you use, and test operators in a non-production cluster before deploying them.

Kubernetes is a registered trademark of The Linux Foundation. Go is a trademark of Google LLC. Prometheus and Helm are projects of the Cloud Native Computing Foundation. Docker is a trademark of Docker, 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 The Linux Foundation, the Cloud Native Computing Foundation, the Kubernetes project, Google, or the Go project.

Cobblestack Commerce, the company used as the running example in this book, is fictional, as are its people, merchants, internal services, and data. Domains such as cobblestack.example are reserved for examples and do not resolve.

Preface

Kubernetes is famous for running containers, but its most powerful idea is quieter: a database of desired state, watched by small programs that keep nudging the real world until it matches. Every Deployment, Service, and Job you have ever created is handled by one of those programs, a *controller*. The same machinery is open to you. You can teach the API server new types of objects, and you can write controllers that understand them. A controller that captures the operational knowledge of a specific application is called an *operator*.

Writing a controller that works in a demo is easy. Writing one that behaves correctly when the cache is stale, the API server returns a conflict, an external service is down, two replicas start at once, or a user deletes an object halfway through a rollout is a different skill. It requires understanding how the Kubernetes API actually behaves, how client-go caches and queues events, and what controller-runtime does on your behalf. This book teaches that understanding from the ground up and turns it into habits you can apply to any operator.

We do it by building one realistic operator from start to finish, testing it against a real API server, and preparing it for production: least-privilege RBAC, metrics and events, leader election, performance tuning, packaging, and safe upgrades.

Who This Book Is For

- **Go developers** who know Kubernetes as users and want to extend it with custom resources and controllers.
- **Platform engineers and SREs** who maintain internal platforms and want to replace scripts, cron jobs, and runbooks with reliable reconciliation.
- **Operator authors** who learned from scaffolding tools and now want to understand what the generated code does, why it is written that way, and how to test and operate it.

You should be comfortable with Go (structs, interfaces, errors, `context`, goroutines, and tests) and with basic Kubernetes objects and `kubectl`. We do not re-teach either; we build on them.

What You Will Learn

- How the Kubernetes API is organized: groups, versions, kinds, resources, `resourceVersion`, `generation`, subresources, and optimistic concurrency.
- API machinery in Go: `runtime.Object`, schemes, `ObjectMeta`, unstructured objects, and serialization.
- client-go: clientsets, dynamic clients, informers, listers, indexers, and rate-limited workqueues, including a complete controller built with client-go alone.
- Custom Resource Definitions and API design: spec versus status, conditions, OpenAPI validation, defaults, and CEL validation rules.

- Go types, markers, and controller-gen for deepcopy code, CRDs, and RBAC in a Kubebuilder-style project.
- controller-runtime in depth: the manager, cached client, reconcile loops, owner references, status patching, server-side apply, finalizers, watches, predicates, and field indexes.
- Admission webhooks for defaulting and validation, multiple API versions, and conversion webhooks with lossless round trips.
- Testing with the fake client and with envtest against a real kube-apiserver and etcd.
- Production concerns: structured logging, events, Prometheus metrics, RBAC and pod security, leader election, performance at scale, packaging with manifests, Helm, and OLM bundles, and upgrading operators and CRDs safely.

How This Book Is Organized

Part I — The Kubernetes API and client-go (Chapters 1–5) explains the reconciliation pattern, the API model, API machinery in Go, client-go, and the informer and workqueue machinery underneath every controller. **Part II — Designing Custom Resources** (Chapters 6–10) designs the Storefront API, generates its CRD from Go types, adds validation and CEL rules, and lays out the operator project. **Part III — Building Controllers with controller-runtime** (Chapters 11–15) builds the reconciler: the manager and cache, the reconcile loop, owned objects, status and conditions, and finalizers. **Part IV — Watches, Webhooks, and API Evolution** (Chapters 16–19) covers watches and predicates, admission webhooks, API versioning, and conversion. **Part V — Testing and Observability** (Chapters 20–23) tests the operator with the fake client and envtest and makes it observable. **Part VI — Running Operators in Production** (Chapters 24–28) covers security, high availability, performance, packaging, and upgrades. The appendices provide a cheat sheet, templates, a production readiness checklist, and a glossary.

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

The Running Example: Cobblestack Commerce

Throughout the book we work with the platform team at **Cobblestack Commerce**, a fictional company that hosts online storefronts for independent shops such as Marlow Books, Fern & Fig, and Quarry Cycles. Every shop needs the same set of objects — a configuration, a web Deployment, a Service, and a route on the company's edge router — and the team is tired of copying YAML and running cleanup scripts by hand. Rosa leads the platform team and cares about clean, durable APIs. Kofi is the site reliability engineer who gets paged at night and asks what happens when things fail. Lin is a Go developer who is new to operators and learns alongside you.

Together they design a `Storefront` custom resource in the API group `platform.cobblestack.example` and build the **storefront operator** that turns each Storefront into running infrastructure, reports honest status, cleans up external routes with a finalizer, validates and defaults new objects, converts

an older `v1alpha1` API to `v1`, and ships with tests, metrics, and an upgrade plan.

Versions and Conventions

The reference project was built and tested with Go 1.27 (the module requires Go 1.26 or newer), the Kubernetes 1.36 client libraries (`k8s.io/api`, `k8s.io/apimachinery`, and `k8s.io/client-go` v0.36), controller-runtime v0.24, controller-tools (controller-gen) v0.21, and envtest control-plane binaries for Kubernetes 1.36. Recent versions of these libraries work the same way for everything in this book; where an API changed recently, we say so. The project follows the layout that Kubebuilder scaffolds, but we write it by hand so that every file is explained.

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

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

> **WARNING**
> A common mistake or something that can cause outages, data loss, or security problems.

Code appears in `monospace`. Long lines are wrapped to fit the page. File names above code blocks, such as `storefront-operator/internal/controller/storefront_controller.go`, tell you where the code lives. Output that requires a full cluster with running pods is introduced with "you should see something like"; everything else was produced by running the code.

Contents

Preface

PART I — THE KUBERNETES API AND CLIENT-GO

1 Operators and Controllers: The Reconciliation Pattern

2 The Kubernetes API Model: Groups, Versions, Kinds, Resources, and Objects

3 API Machinery in Go: runtime.Object, Schemes, ObjectMeta, and Serialization

4 client-go Essentials: rest.Config, Clientsets, Dynamic Clients, and Typed Calls

5 Informers, Listers, Indexers, and Workqueues: A Controller with client-go Alone

PART II — DESIGNING CUSTOM RESOURCES

6 Custom Resource Definitions: Schemas, Scope, Subresources, and Printer Columns

7 Designing a Good Kubernetes API: Spec, Status, and the Storefront Resource

8 Go Types, Markers, and controller-gen: Deepcopy, CRDs, and RBAC from Code

9 Schema Validation and Defaulting: OpenAPI Constraints and CEL Rules

10 The Operator Project: A Kubebuilder-Style Layout Built by Hand

PART III — BUILDING CONTROLLERS WITH CONTROLLER-RUNTIME

11 Inside controller-runtime: Manager, Client, Cache, and Scheme

12 Writing the Reconcile Loop: Level-Triggered, Idempotent, and Retry-Safe

13 Owning Child Objects: CreateOrUpdate, Owner References, and Garbage Collection

14 Status, Conditions, and Patching: observedGeneration, Update, Patch, and Server-Side Apply

15 Finalizers and External Resources: Cleaning Up Safely

PART IV — WATCHES, WEBHOOKS, AND API EVOLUTION

16 Watches, Predicates, Event Handlers, and Field Indexes

17 Admission Webhooks: Defaulting and Validation in Go

18 Versioning a Custom Resource: Multiple Versions, Storage Versions, and Deprecation

19 Conversion Webhooks: Hub and Spoke and Lossless Round Trips

PART V — TESTING AND OBSERVABILITY

20 Unit Testing Reconcilers with the Fake Client

21 Integration Testing with envtest

22 Testing Webhooks, Conversion, and Failure Paths

23 Observability: Structured Logging, Events, and Prometheus Metrics

PART VI — RUNNING OPERATORS IN PRODUCTION

24 RBAC and Security: Least Privilege, Service Accounts, and Hardened Pods

25 Leader Election, High Availability, and Graceful Shutdown

26 Performance at Scale: Caches, Concurrency, Rate Limits, and API Server Load

27 Packaging and Distribution: Manifests, Helm Charts, and OLM Bundles

28 Upgrades and Day-2 Operations: Releasing New Versions Safely

APPENDICES

A Cheat Sheet: client-go, controller-runtime, Markers, and kubectl

B Templates and Reference: Manifests, Markers, Tests, and Build Targets

C Production Readiness Checklist for Operators

D Glossary

PART I

The Kubernetes API and client-go

How controllers think and how the API server really behaves: the reconciliation pattern, groups, versions, kinds, and resources, the API machinery types every Go client shares, client-go's clientsets and dynamic client, and the informers, listers, and workqueues underneath every controller.

1 · Operators and Controllers
2 · The Kubernetes API Model
3 · API Machinery in Go
4 · client-go Essentials
5 · Informers, Listers, and Workqueues

CHAPTER 1 · PART I — THE KUBERNETES API AND CLIENT-GO

Operators and Controllers: The Reconciliation Pattern

Every useful thing Kubernetes does for you is done by a loop that compares what you asked for with what exists and then closes the gap. In this chapter we take that loop apart, write a toy version in plain Go, meet the team at Cobblestack Commerce who need one of their own, and look at the finished storefront operator we will build across the rest of the book.

A Thermostat, Not a Script

Think about the difference between a thermostat and a script that turns on the heating. The script runs once: it switches the boiler on and exits. If someone opens a window an hour later, the script does not care. A thermostat works differently. You tell it the temperature you want, and it keeps measuring the room. When the room is too cold it heats; when the room is warm enough it stops; when someone opens a window it notices and heats again. The thermostat does not remember a list of things it did. It only knows two facts at any moment: the **desired state** (21 degrees) and the **observed state** (what the sensor says right now). Its whole job is to make the second match the first.

Kubernetes is built out of thermostats. When you run `kubectl apply` with a Deployment that asks for three replicas, nothing in the command itself starts a container. The command stores a record in the API server that says "three replicas of this pod template should exist." Then a set of independent programs, each watching a small part of that record, notice the gap between what the record says and what actually runs, and they act. If a node dies and takes a pod with it, nobody needs to re-run the command: the same programs notice the new gap and close it again.

Those programs are called **controllers**, and the act of comparing desired with observed state and taking a step to close the difference is called **reconciliation**. This book is about writing controllers in Go for your own kinds of objects. Before we write any Kubernetes code, it is worth being precise about what a controller is and what it is not, because most of the hard bugs in operators come from treating a controller like a script.

The three ideas behind every controller

1. **Desired state is data.** The user writes intent into an object stored by the API server, usually in a field called `spec`. The user does not call a function; they change a document.
2. **Observed state is measured, not remembered.** On every pass, the controller looks at the world as it is now: the child objects in the API, the status of pods, the answer from an external system. It does not trust a private memory of what it did last time.

3. **Actions are small, repeatable steps.** The controller takes the next step toward the desired state and then lets the loop run again. A step that has already been taken must be safe to take again, because the loop will run many times for the same object.

These three ideas let controllers recover from almost anything without a human: a crashed process, a lost packet, a half-finished change, or a hand edit all leave the world different from the desired state, and the next pass moves it back. Kofi, the on-call engineer you will meet shortly, likes to put it this way: the best answer to "what happens at 3 a.m. when the operator crashes halfway through?" is "it starts again and finishes the job."

Controllers Inside Kubernetes

The cleanest way to understand a controller is to look at the ones Kubernetes already ships. The `kube-controller-manager` process runs dozens of them. Each one owns a narrow responsibility and talks to the rest of the system only through the API server. None of them call each other directly.

Follow what happens when you create a Deployment:

```
you: kubectl apply -f deployment.yaml
          |
          v
+--------------------+   stores    +------------------------------+
|   kube-apiserver   | ----------> | etcd: Deployment marlow-web  |
+--------------------+             +------------------------------+
   ^  watch / write
   |
   |  Deployment controller: "this Deployment needs a ReplicaSet
   |  for its current pod template" -> creates ReplicaSet
   |
   |  ReplicaSet controller: "this ReplicaSet wants 3 pods, I see 0"
   |  -> creates 3 Pod objects
   |
   |  Scheduler: "these Pods have no node" -> sets spec.nodeName
   |
   |  kubelet on each node: "a Pod is bound to me" -> starts the
   |  containers and reports status back to the API server
```

Notice the shape. The Deployment controller does not start containers; it only manages ReplicaSets. The ReplicaSet controller does not pick nodes; it only creates and deletes Pod objects. Every step is one controller reading some objects, writing others, and trusting the next controller in the chain to react.

Two details of this chain matter for everything we build later. First, the link between parent and child is an **owner reference**: each ReplicaSet records which Deployment owns it, and each Pod records its ReplicaSet. That lets controllers find "my" children, and it lets the garbage collector delete children

when the owner goes away. Second, each controller reports what it observed in the object's `status`. The Deployment's `status.readyReplicas` is not something you set. A controller measured it and wrote it down, so that people and other controllers can read it.

Level-triggered, not edge-triggered

Electronics engineers distinguish between circuits that react to an *edge* (the moment a signal changes) and circuits that react to a *level* (the value the signal has right now). Kubernetes controllers are designed to be **level-triggered**. They do receive notifications when objects change, but they use those notifications only as a hint that "something about this object may need attention." The decision about what to do is always made by reading the current state, never by interpreting the notification itself.

The difference shows up when things go wrong. An edge-triggered design says "a pod was deleted, so create one pod"; lose a notification and the count drifts. A level-triggered design says "I want three, I see one, so create two," and it is right no matter how many notifications it missed. Chapter 12 makes this the central rule for writing the `Reconcile` method.

> **NOTE**
>
> Controllers are eventually consistent by design. Between the moment you change a spec and the moment the world matches it, there is a window where the two disagree. A good controller reports that window honestly in its status ("2 of 3 replicas ready") instead of pretending the change happened instantly.

A Toy Reconcile Loop in Plain Go

Before we touch the Kubernetes libraries, let us build the smallest possible control loop and watch it recover from trouble. There is no API server here: the "cluster" is a Go map of pod names, and the desired state is a struct. The point is the shape of the code, which you will see again in every controller in this book. The `examples/` folders in Part I are small programs you create next to the operator project; they are not part of the operator itself.

examples/toyloop/main.go

```
// Command toyloop simulates a control loop without Kubernetes.
package main

import (
    "fmt"
    "sort"
)

// Desired is what the user asked for (the "spec").
type Desired struct {
    Shop     string
    Replicas int
}
```

```
// World is the simulated cluster: the pods that actually exist.
type World struct {
    pods map[string]bool
    seq  int
}

// Observe returns the current pod names, sorted (the "status").
func (w *World) Observe() []string {
    names := make([]string, 0, len(w.pods))
    for n := range w.pods {
        names = append(names, n)
    }
    sort.Strings(names)
    return names
}

// reconcile compares desired and observed state and takes one step
// toward the desired state. It never looks at what happened before.
func reconcile(d Desired, w *World) string {
    have := w.Observe()
    switch {
    case len(have) < d.Replicas:
        w.seq++
        name := fmt.Sprintf("%s-web-%d", d.Shop, w.seq)
        w.pods[name] = true
        return "created " + name
    case len(have) > d.Replicas:
        victim := have[len(have)-1]
        delete(w.pods, victim)
        return "deleted " + victim
    default:
        return "in sync"
    }
}

func main() {
    desired := Desired{Shop: "marlow-books", Replicas: 3}
    world := &World{pods: map[string]bool{}}

    for tick := 1; tick <= 9; tick++ {
        switch tick {
        case 5: // something outside the loop breaks the world
            delete(world.pods, "marlow-books-web-2")
            fmt.Println("        !! pod marlow-books-web-2 disappeared")
        case 7: // the user edits the spec
            desired.Replicas = 2
            fmt.Println("        !! spec.replicas changed to 2")
        }
        have := len(world.Observe())
        action := reconcile(desired, world)
        fmt.Printf("tick %d: want %d, have %d -> %s\n",
            tick, desired.Replicas, have, action)
    }
}
```

Run it with `go run ./examples/toyloop`. This is the real output:

```
tick 1: want 3, have 0 -> created marlow-books-web-1
tick 2: want 3, have 1 -> created marlow-books-web-2
tick 3: want 3, have 2 -> created marlow-books-web-3
tick 4: want 3, have 3 -> in sync
        !! pod marlow-books-web-2 disappeared
tick 5: want 3, have 2 -> created marlow-books-web-4
tick 6: want 3, have 3 -> in sync
        !! spec.replicas changed to 2
tick 7: want 2, have 3 -> deleted marlow-books-web-4
tick 8: want 2, have 2 -> in sync
tick 9: want 2, have 2 -> in sync
```

Reading the output

The first three ticks are the loop converging from nothing: it wants three pods, sees fewer, and creates one per pass. It deliberately takes *one* step at a time. Real controllers often create several children in one pass, but the principle is the same: do some work, then let the loop look again. On tick 4 the world matches, and the most important line in the output appears: `in sync`. A controller that has nothing to do must do nothing. If your controller writes to the API on every pass even when nothing changed, it will generate traffic and trigger itself again, which is the root of the "hot loop" problems we discuss in Chapter 26.

Before tick 5, something outside the loop deletes a pod. The loop was never told which pod died or why. It does not need to know: it compares two with three and creates a replacement. That is level-triggered behavior in miniature. Before tick 7, the user changes the desired state to two replicas, and the same function removes the extra pod. The user changed the document, and the loop derived the action.

The toy hides what real controllers must handle: stale observations, failed actions, conflicting writers, and cleanup outside Kubernetes. Each gets its own chapter in Parts III and IV.

> **TIP**
>
> When you design any controller, write the "in sync" case first. Ask: "If I run this pass twice in a row with nothing changing in between, does the second pass do anything?" If the answer is yes, find out why before you write any more code.

From Controllers to Operators

The built-in controllers know about built-in objects: Deployments, Jobs, Services, and so on. An **operator** applies exactly the same pattern to an object type that you define yourself. The term comes from the idea of encoding what a human operator of some software would do — install it, configure it, scale it, back it up, upgrade it, clean up after it — as a controller that runs all the time.

An operator has two halves:

- **A custom resource**: a new API type, registered with the API server through a **CustomResourceDefinition** (CRD). Once the CRD is installed, users can create, list, watch, and delete objects of your kind with `kubectl` and every other Kubernetes tool, exactly like built-in objects. The custom resource is the user interface of your operator.
- **A controller**: a program, usually running as a Deployment inside the cluster, that watches those custom objects and reconciles the world to match them. It creates built-in objects (Deployments, Services, ConfigMaps), calls external systems, and writes what it observed back into the custom object's status.

What makes an operator valuable is not the Kubernetes plumbing, which is mostly the same for every operator, but the **domain knowledge** encoded in the reconcile logic. A database operator knows that a replica must be promoted before the old primary is removed. A certificate operator knows when a certificate is close to expiry. Our storefront operator will know which plans allow how many replicas, how to roll pods when a shop's configuration changes, and how to remove a shop's public route before the shop disappears. Without the operator, that knowledge lives in runbooks, wiki pages, and the memory of whoever is on call.

In this book a **controller** is any reconcile loop, and an **operator** is a controller packaged with the custom resources it manages.

When Not to Write an Operator

Operators are powerful, and they are also software you have to maintain, test, secure, upgrade, and be paged for. Rosa, who leads the platform team in our running example, asks one question before approving any new operator: "What would we lose if we did this with something simpler?" Often the honest answer is "nothing." Here are the common cases where a simpler tool does the job:

What you need	Simpler tool	Why it is enough
Install the same set of manifests with a few parameters	A Helm chart or Kustomize overlay	No ongoing reaction to change is needed after install
Run a task on a schedule (backups, reports)	A CronJob	Kubernetes already reconciles the schedule
Run a task once after deploy	A Job, or a CI/CD step	One-shot work does not need a loop
Keep many clusters in sync with Git	A GitOps tool	Someone already wrote that controller
Reject bad objects at admission	CEL rules or ValidatingAdmissionPolicy	Declarative rules, no server to run

An operator starts to pay for itself when at least one of these is true: the application needs **continuous reaction** to changes in the cluster or in the outside world; the correct action depends on **observed state**, not just on input parameters; the lifecycle has **ordered steps** that a human would otherwise follow from a runbook (especially on deletion and upgrade); or many teams need a **small, stable API** that hides a larger set of objects. If none of these apply, a chart and a CronJob are cheaper.

> **WARNING**
> Do not build an operator just to template YAML. If your reconcile function only renders a few objects from parameters and never reads status, never reacts to outside changes, and never cleans anything up, you have rebuilt a templating tool with extra failure modes — and you now own an always-running process with cluster-wide permissions.

Meet Cobblestack Commerce

Throughout the book we follow one fictional company. **Cobblestack Commerce** hosts online storefronts for independent shops: bookshops, tea merchants, bicycle repairers, and hundreds more. Each merchant gets a public site at an address like `marlow-books.shops.cobblestack.example`. The company runs everything on Kubernetes, and its **platform team** is responsible for the machinery that turns "a new merchant signed up" into "a working storefront on the internet."

Three people on that team will appear in the chapters ahead:

- **Rosa** leads the platform team. She cares most about API design and safe upgrades. When a field name is ambiguous or a change could break existing users, Rosa is the one who notices.
- **Kofi** is a site reliability engineer and spends one week in four on call. His questions always start with "what happens at 3 a.m. when…", and he is the reason our operator reports honest status, emits events, and exposes metrics.
- **Lin** is a Go developer who joined the team recently. Lin knows Go and has used Kubernetes as a developer, but has never written a controller. Lin is learning alongside you, and asks the questions you are likely to ask.

The problem

Every storefront at Cobblestack is made of the same four pieces:

1. A **ConfigMap** with the shop's settings: its hostname, plan, and theme.
2. A **Deployment** running the storefront web image (`registry.cobblestack.example/storefront-web:2.4.1`) with the right number of replicas for the merchant's plan.
3. A **Service** in front of those pods.
4. A **route on the company's edge router**, an internal HTTP API outside Kubernetes that maps the public hostname to the Service.

Today a shell script and YAML templates create these pieces. That worked for twenty shops; with hundreds, it hurts. Lin cataloged the recurring problems:

- After a theme change, running pods keep the old settings until somebody restarts them.
- When a merchant leaves, the edge route sometimes stays behind, pointing at a deleted Service.
- Starter-plan shops end up with more replicas than allowed, because the limit lives in a wiki page.
- Nobody can answer "which shops are unhealthy right now?" without several commands per shop.
- Kofi's night pages are mostly half-finished script runs that left a shop partly created.

Every item on that list is a gap between desired and observed state that no loop is watching. That is the signature of a problem a controller solves well. Rosa approved the project with one condition: the new API must be small, versioned, and safe to evolve, because merchants' automation will depend on it.

The plan: a Storefront resource

The team decides to introduce a custom resource called `Storefront` in the API group `platform.cobblestack.example`. One Storefront object describes one merchant's shop. Here is the one for Marlow Books, a growth-plan bookshop with three replicas and a checkout Secret:

storefront-operator/config/samples/platform_v1_storefront_marlow-books.yaml

```
apiVersion: platform.cobblestack.example/v1
kind: Storefront
metadata:
  name: marlow-books
  namespace: shops
spec:
  image: registry.cobblestack.example/storefront-web:2.4.1
  hostname: marlow-books.shops.cobblestack.example
  plan: growth
  replicas: 3
  theme: paper
  checkoutSecretRef:
    name: marlow-books-checkout
```

That is the whole user interface. The merchant onboarding system writes one of these; the operator does everything else. Fern and Fig, a starter-plan shop, relies on defaults for plan, replicas, and theme:

storefront-operator/config/samples/platform_v1_storefront_fern-and-fig.yaml

```
apiVersion: platform.cobblestack.example/v1
kind: Storefront
metadata:
  name: fern-and-fig
  namespace: shops
spec:
  image: registry.cobblestack.example/storefront-web:2.4.1
  hostname: fern-and-fig.shops.cobblestack.example
  # plan defaults to starter, replicas to 1, theme to classic
```

A third sample, Quarry Cycles, is an enterprise-plan shop with five replicas and the `bold` theme; we use it for scale and deletion demonstrations. A fourth merchant, Lumen Tea, still uses an older `v1alpha1` shape of the API, and it will star in Chapters 18 and 19 when we discuss versioning and conversion.

A Preview of the Finished Operator

It helps to see where we are going. The following output is real: we started a local Kubernetes control plane (an API server and etcd, but no nodes), installed the Storefront CRD, applied the three v1 samples, and ran the finished operator from this book's reference project against it for a few seconds. Then we asked for the storefronts:

```
$ kubectl get storefronts -n shops       # URL column cut to fit the page
NAME            PLAN         REPLICAS   READY   AGE
fern-and-fig    starter                 False   22s
marlow-books    growth       3          False   10s
quarry-cycles   enterprise   5          False   10s

$ kubectl get sf -n shops -o custom-columns='NAME:.metadata.name,PLAN:.spec.plan,
    READY:.status.conditions[?(@.type=="Ready")].status,URL:.status.url'
NAME            PLAN         READY   URL
fern-and-fig    starter      False   https://fern-and-fig.shops.cobblestack.example
marlow-books    growth       False   https://marlow-books.shops.cobblestack.example
quarry-cycles   enterprise   False   https://quarry-cycles.shops.cobblestack.example
```

(The custom-columns argument is one string; we wrapped it for the page.) And the children the operator created:

```
$ kubectl get deploy,svc,cm -n shops
NAME                                READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/fern-and-fig-web    0/1     0            0           22s
deployment.apps/marlow-books-web    0/3     0            0           22s
deployment.apps/quarry-cycles-web   0/5     0            0           22s

NAME                        TYPE        CLUSTER-IP   EXTERNAL-IP   PORT(S)   AGE
service/fern-and-fig-web    ClusterIP   10.0.0.162   <none>        80/TCP    22s
service/marlow-books-web    ClusterIP   10.0.0.248   <none>        80/TCP    22s
service/quarry-cycles-web   ClusterIP   10.0.0.200   <none>        80/TCP    22s

NAME                              DATA   AGE
configmap/fern-and-fig-config     3      22s
configmap/marlow-books-config     3      22s
configmap/quarry-cycles-config    3      22s
```

Reading the output

Start with the columns of the first listing. `PLAN`, `REPLICAS`, `READY`, and `URL` are not built into `kubectl`; they are **printer columns** declared in the CRD, which we design in Chapter 6 and generate from Go markers in Chapter 8. `PLAN` and `REPLICAS` come from the spec the user wrote. `READY` and `URL` come from the status the operator wrote. Looking at one line, you can see both what was asked for and what was observed, which is exactly the point.

The `REPLICAS` column is empty for Fern and Fig because the sample omits `spec.replicas` and we ran this preview with the admission webhooks switched off. With the defaulting webhook from Chapter 17 enabled, the field is filled in with the plan's default and the column shows `1`. The operator still created a one-replica Deployment, because the reconcile code applies the same default when the field is missing — you can see `0/1` in the second listing.

`READY` is `False` for every shop, and that is honest rather than broken. Our local control plane has no scheduler, no nodes, and no kubelet, so the Deployments are stored but no pod ever starts; the Deployments show `0/3` and `0/5` ready. The operator measured that and reported it. On a real cluster with healthy pods you should see `True` in that column once the rollouts finish. The `URL` column is filled in because the operator successfully created each shop's edge route — in this run against an in-memory fake of the edge router, which is the default when no router address is configured.

Behind the `READY` column is a richer status. Here is part of what the operator wrote for Marlow Books (real output, trimmed):

```
status:
  conditions:
  - message: 0/3 replicas ready
    observedGeneration: 3
    reason: DeploymentPending
    status: "False"
    type: Ready
  - message: waiting for the Deployment rollout
    observedGeneration: 3
    reason: RolloutInProgress
    status: "True"
    type: Progressing
  - message: no problems detected
    observedGeneration: 3
    reason: AsExpected
    status: "False"
    type: Degraded
  observedGeneration: 3
  routeID: rt-marlow-books.shops.cobblestack.example
  url: https://marlow-books.shops.cobblestack.example
```

Three **conditions** — `Ready`, `Progressing`, and `Degraded` — each with a machine-readable reason and a human-readable message. Kofi can tell at a glance that the shop is not ready because the Deployment is still rolling out, and that nothing is actually broken. The `observedGeneration` fields tell him which

version of the spec the operator last acted on (generation 3: we had edited the spec twice). We explain generations in Chapter 2 and design these conditions in Chapters 7 and 14.

Finally, every child the operator creates carries an owner reference back to its Storefront. This is the Deployment's metadata (real output, trimmed):

```
metadata:
  labels:
    app.kubernetes.io/instance: marlow-books
    app.kubernetes.io/managed-by: storefront-operator
    app.kubernetes.io/name: storefront
    platform.cobblestack.example/plan: growth
  name: marlow-books-web
  namespace: shops
  ownerReferences:
  - apiVersion: platform.cobblestack.example/v1
    blockOwnerDeletion: true
    controller: true
    kind: Storefront
    name: marlow-books
    uid: b840eead-f41b-4165-80ca-562a672aecd9
```

That owner reference is the same mechanism that links a ReplicaSet to its Deployment. On a real cluster, deleting the Storefront lets the garbage collector remove the ConfigMap, Deployment, and Service automatically. The edge route lives outside Kubernetes, so the garbage collector cannot see it; the operator removes it itself, using a **finalizer**, before the Storefront is allowed to disappear. That is Lin's "route left behind" problem solved, and it is the subject of Chapter 15.

What the operator does on every pass

Stripped of detail, the reconcile loop for one Storefront looks like this. Do not worry about the Go yet; this is the outline we fill in from Chapter 12 onward.

```
Reconcile(namespace/name):
  read the Storefront            (gone? nothing to do)
  being deleted?                 remove edge route, drop finalizer, stop
  ensure our finalizer is set
  ensure ConfigMap  <name>-config   (hostname, plan, theme)
  ensure Deployment <name>-web      (replicas, image, config hash)
  ensure Service    <name>-web      (port 80 -> http)
  ensure edge route for the hostname
  write status: conditions, URL, route ID, observedGeneration
  route failed? return the error so the pass is retried later
```

Every "ensure" step is idempotent: it creates the object if it is missing, updates it if it differs, and does nothing if it already matches. That is the toy loop's `in sync` case, applied to four kinds of object. The **config hash** on the Deployment's pod template deserves a word now, because it fixes the stale theme