O'REILLY®

Cloud Native Go

Building Reliable Services in Unreliable Environments

Free Chapters

compliments of

Cockroach Labs

Matthew A. Titmus

CockroachDB

Build what you dream.

Never worry about your database again.

```
/* Power your apps with the serverless SQL database built for developers.
   Elastic scale, zero operations and free forever. */
```

A hassle-free SQL database

Give your apps a distributed platform that's always available — and free your team from operational headaches.

Cost-effective, elastic and automated scale

Trust in infrastructure that grows alongside your apps. Pay only for what you use, when you use it.

Compatible with PostgreSQL

Build with familiar SQL for transactional consistency and relational schema efficiency.

Start instantly

www.cockroachlabs.com/serverless

TRUSTED BY INNOVATORS

COMCAST

Hard Rock DIGITAL

BOSE

orca security

rubrik

Five9

nubank

Cloud Native Go

Building Reliable Services in Unreliable Environments

This excerpt contains Chapters 5–7. The complete book is available on the O'Reilly Online Learning Platform and through other retailers.

Matthew A. Titmus

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

Cloud Native Go
by Matthew A. Titmus

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://oreilly.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Acquisitions Editor: Suzanne McQuade
Production Editor: Daniel Elfanbaum
Proofreader: nSight, Inc.
Interior Designer: David Futato
Illustrator: Kate Dullea

Development Editor: Amelia Blevins
Copyeditor: Piper Editorial Consulting, LLC
Indexer: nSight, Inc.
Cover Designer: Karen Montgomery

April 2021: First Edition

Revision History for the First Edition
2021-04-20: First Release

See *http://oreilly.com/catalog/errata.csp?isbn=9781492076339* for release details.

978-1-492-07633-9

[LSI]

For you, Dad.

Your gentleness, wisdom, and humility are dearly missed.

Also, you taught me to code, so any mistakes in this book are technically your fault.

Table of Contents

Foreword

In the early days of the app economy, scale was a challenge that only a select few really had to fret over. The term "web scale" was coined to address the early challenges of companies like Amazon, Google, Facebook, and Netflix to reach their massive volumes of users. Now, though, nearly every company sees massive concurrency and increasing demands on their applications and infrastructure. Even the smallest start-up must think about scale from day one.

Modern applications are overwhelmingly data-intensive and increasingly global in nature (like Spotify, Uber, or Square), and modern consumers expect highly dynamic and personalized experiences available instantaneously. Developers now need to balance both consistency and locality around ever more components—which themselves must be managed and scaled.

Worse, users now have zero tolerance for latency or downtime even under the heaviest workloads. In this context, your most successful moments (your application sees rapid growth or your content goes viral on social media) can instead quickly become your worst.

But if you plan, test, and continually optimize for scale from the start, unexpected and rapid growth can simply happen without you losing sleep or scrambling to patch or repair a failing stack. Author Matthew Titmus has been in the tech industry for decades and compiled his hard-learned lessons into *Cloud Native Go*.

As a database designed for cloud native elastic scale and resilience in the face of disasters both trivial and tremendous, CockroachDB is proud to sponsor three chapters ("Building a Cloud Native Service", "It's All About Dependability," and "Scalability") so you can see how to build your cloud native application the right way, right from the start.

CHAPTER 5

Building a Cloud Native Service

> Life was simple before World War II. After that, we had systems.[1]
>
> —Grace Hopper, OCLC Newsletter (1987)

In this chapter, our real work finally begins.

We'll weave together many of the materials discussed throughout Part 2 to create a service that will serve as the jumping-off point for the remainder of the book. As we go forward, we'll iterate on what we begin here, adding layers of functionality with each chapter until, at the conclusion, we have ourselves a true cloud native application.

Naturally, it won't be "production ready"—it will be missing important security features, for example—but it will provide a solid foundation for us to build upon.

But what do we build?

Let's Build a Service!

Okay. So. We need something to build.

It should be conceptually simple, straightforward enough to implement in its most basic form, but non-trivial and amenable to scaling and distributing. Something that we can iteratively refine over the remainder of the book. I put a lot of thought into this, considering different ideas for what our application would be, but in the end the answer was obvious.

We'll build ourselves a distributed key-value store.

1 Schieber, Philip. "The Wit and Wisdom of Grace Hopper." *OCLC Newsletter*, March/April, 1987, No. 167.

What's a Key-Value Store?

A key-value store is a kind of nonrelational database that stores data as a collection of key-value pairs. They're very different from the better-known relational databases, like Microsoft SQL Server or PostgreSQL, that we know and love.[2] Where relational databases structure their data among fixed tables with well-defined data types, key-value stores are far simpler, allowing users to associate a unique identifier (the key) with an arbitrary value.

In other words, at its heart, a key-value store is really just a map with a service endpoint, as shown in Figure 5-1. They're the simplest possible database.

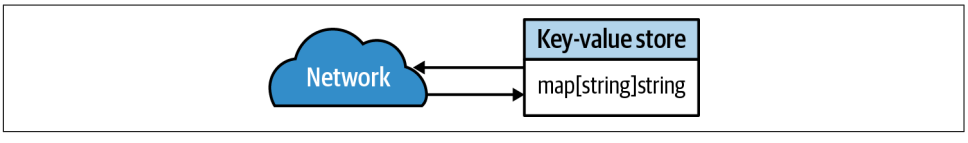

Figure 5-1. A key-value store is essentially a map with a service endpoint

Requirements

By the end of this chapter, we're going to have built a simple, nondistributed key-value store that can do all of the things that a (monolithic) key-value store should do.

- It must be able to store arbitrary key-value pairs.
- It must provide service endpoints that allow a user to put, get, and delete key-value pairs.
- It must be able to persistently store its data in some fashion.

Finally, we'd like the service to be idempotent. But why?

What Is Idempotence and Why Does It Matter?

The concept of *idempotence* has its origins in algebra, where it describes particular properties of certain mathematical operations. Fortunately, this isn't a math book. We're not going to talk about that (except in the sidebar at the end of this section).

In the programming world, an operation (such as a method or service call) is idempotent if calling it once has the same effect as calling it multiple times. For example, the assignment operation `x=1` is idempotent, because `x` will always be `1` no matter how many times you assign it. Similarly, an HTTP `PUT` method is idempotent because `PUT`-ting a resource in a place multiple times won't change anything: it won't get any

2 For some definition of "love."

more PUT the second time.[3] The operation x+=1, however, is not idempotent, because every time that it's called, a new state is produced.

Less discussed, but also important, is the related property of *nullipotence*, in which a function or operation has no side effect at all. For example, the x=1 assignment and an HTTP PUT are idempotent but not nullipotent because they trigger state changes. Assigning a value to itself, such as x=x, is nullipotent because no state has changed as a result of it. Similarly, simply reading data, as with an HTTP GET, usually has no side effects, so it's also nullipotent.

Of course, that's all very nice in theory, but why should we care in the real world? Well, as it turns out, designing your service methods to be idempotent provides a number of very real benefits:

Idempotent operations are safer
: What if you make a request to a service, but get no response? You'll probably try again. But what if it heard you the first time?[4] If the service method is idempotent, then no harm done. But if it's not, you could have a problem. This scenario is more common than you think. Networks are unreliable. Responses can be delayed; packets can get dropped.

Idempotent operations are often simpler
: Idempotent operations are more self-contained and easier to implement. Compare, for example, an idempotent PUT method that simply adds a key-value pair into a backing data store, and a similar but nonidempotent CREATE method that returns an error if the data store already contains the key. The PUT logic is simple: receive request, set value. The CREATE, on the other hand, requires additional layers of error checking and handling, and possibly even distributed locking and coordination among any service replicas, making its service harder to scale.

Idempotent operations are more declarative
: Building an idempotent API encourages the designer to focus on end-states, encouraging the production of methods that are more *declarative*: they allow users to tell a service *what needs to be done*, instead of telling it *how to do it*. This may seem to be a fine point, but declarative methods—as opposed to *imperative methods*—free users from having to deal with low-level constructs, allowing them to focus on their goals and minimizing potential side-effects.

In fact, idempotence provides such an advantage, particularly in a cloud native context, that some very smart people have even gone so far as to assert that it's a

3 If it does, something is very wrong.

4 Or, like my son, was only *pretending* not to hear you.

synonym for "cloud native."[5] I don't think that I'd go quite that far, but I *would* say that if your service aims to be cloud native, accepting any less than idempotence is asking for trouble.

The Mathematical Definition of Idempotence

The origin of idempotence is in mathematics, where it describes an operation that can be applied multiple times without changing the result beyond the initial application.

In purely mathematical terms: a function is idempotent if $f(f(x)) = f(x)$ for all x.

For example, taking the absolute value $abs(x)$ of an integer number x is an idempotent function because $abs(x) = abs(abs(x))$ is true for each real number x.

The Eventual Goal

These requirements are quite a lot to chew on, but they represent the absolute minimum for our key-value store to be usable. Later on we'll add some important basic functionality, like support for multiple users and data encryption in transit. More importantly, though, we'll introduce techniques and technologies that make the service more scalable, resilient, and generally capable of surviving and thriving in a cruel, uncertain universe.

Generation 0: The Core Functionality

Okay, let's get started. First things first. Without worrying about user requests and persistence, let's first build the core functions, which can be called later from whatever web framework we decide to use.

Storing arbitrary key-value pairs
: For now, we can implement this with a simple map, but what kind? For the sake of simplicity, we'll limit ourselves to keys and values that are simple strings, though we may choose to allow arbitrary types later. We'll just use a simple `map[string]string` as our core data structure.

Allow put, get, and delete of key-value pairs
: In this initial iteration, we'll create a simple Go API that we can call to perform the basic modification operations. Partitioning the functionality in this way will make it easier to test and easier to update in future iterations.

5 "Cloud native is not a synonym for microservices… if cloud native has to be a synonym for anything, it would be idempotent, which definitely needs a synonym." —Holly Cummins (Cloud Native London 2018).

Your Super Simple API

The first thing that we need to do is to create our map. The heart of our key-value store:

```
var store = make(map[string]string)
```

Isn't it a beauty? So simple. Don't worry, we'll make it more complicated later.

The first function that we'll create is, appropriately, `PUT`, which will be used to add records to the store. It does exactly what its name suggests: it accepts `key` and `value` strings, and puts them into `store`. `PUT`'s function signature includes an `error` return, which we'll need later:

```
func Put(key string, value string) error {
    store[key] = value

    return nil
}
```

Because we're making the conscious choice to create an idempotent service, `Put` doesn't check to see whether an existing key-value pair is being overwritten, so it will happily do so if asked. Multiple executions of `Put` with the same parameters will have the same result, regardless of any current state.

Now that we've established a basic pattern, writing the `Get` and `Delete` operations is just a matter of following through:

```
var ErrorNoSuchKey = errors.New("no such key")

func Get(key string) (string, error) {
    value, ok := store[key]

    if !ok {
        return "", ErrorNoSuchKey
    }

    return value, nil
}

func Delete(key string) error {
    delete(store, key)

    return nil
}
```

But look carefully: see how when `Get` returns an error, it doesn't use `errors.New`? Instead it returns the prebuilt `ErrorNoSuchKey` error value. But why? This is an example of a *sentinel error*, which allows the consuming service to determine exactly what type of error it's receiving and to respond accordingly. For example, it might do something like this:

```
    if errors.Is(err, ErrorNoSuchKey) {
        http.Error(w, err.Error(), http.StatusNotFound)
        return
    }
```

Now that you have your absolute minimal function set (really, really minimal), don't forget to write tests. We're not going to do that here, but if you're feeling anxious to move forward (or lazy—lazy works too) you can grab the code from the GitHub repository created for this book.

Generation 1: The Monolith

Now that we have a minimally functional key-value API, we can begin building a service around it. We have a few different options for how to do this. We could use something like GraphQL. There are some decent third-party packages out there that we could use, but we don't have the kind of complex data landscape to necessitate it. We could also use remote procedure call (RPC), which is supported by the standard `net/rpc` package, or even gRPC, but these require additional overhead for the client, and again our data just isn't complex enough to warrant it.

That leaves us with representational state transfer (REST). REST isn't a lot of people's favorite, but it *is* simple, and it's perfectly adequate for our needs.

Building an HTTP Server with net/http

Go doesn't have any web frameworks that are as sophisticated or historied as something like Django or Flask. What it does have, however, is a strong set of standard libraries that are perfectly adequate for 80% of use cases. Even better: they're designed to be extensible, so there *are* a number of Go web frameworks that extend them.

For now, let's take a look at the standard HTTP handler idiom in Go, in the form of a "Hello World" as implemented with `net/http`:

```
package main

import (
    "log"
    "net/http"
)

func helloGoHandler(w http.ResponseWriter, r *http.Request) {
    w.Write([]byte("Hello net/http!\n"))
}

func main() {
    http.HandleFunc("/", helloGoHandler)

    log.Fatal(http.ListenAndServe(":8080", nil))
}
```

In the previous example, we define a method, `helloGoHandler`, which satisfies the definition of a `http.HandlerFunc`:

```
type HandlerFunc func(http.ResponseWriter, *http.Request)
```

The `http.ResponseWriter` and a `*http.Request` parameters can be used to construct the HTTP response and retrieve the request, respectively. You can use the `http.HandleFunc` function to register `helloGoHandler` as the handler function for any request that matches a given pattern (the root path, in this example).

Once you've registered our handlers, you can call `ListenAndServe`, which listens on the address `addr`. It also accepts a second parameter, set to `nil` in our example.

You'll notice that `ListenAndServe` is also wrapped in a `log.Fatal` call. This is because `ListenAndServe` always stops the execution flow, only returning in the event of an error. Therefore, it always returns a non-nil `error`, which we always want to log.

The previous example is a complete program that can be compiled and run using `go run`:

```
$ go run .
```

Congratulations! You're now running the world's tiniest web service. Now go ahead and test it with `curl` or your favorite web browser:

```
$ curl http://localhost:8080
Hello net/http!
```

ListenAndServe, Handlers, and HTTP Request Multiplexers

The `http.ListenAndServe` function starts an HTTP server with a given address and handler. If the handler is `nil`, which it usually is when you're using only the standard `net/http` library, the `DefaultServeMux` value is used. But what's a handler? What is `DefaultServeMux`? *What's a "mux"?*

A `Handler` is any type that satisfies the `Handler` interface by providing a `ServeHTTP` method, defined in the following:

```
type Handler interface {
    ServeHTTP(ResponseWriter, *Request)
}
```

Most handler implementations, including the default handler, act as a "mux"—short for "multiplexer"—that can direct incoming signals to one of several possible outputs. When a request is received by a service that's been started by `ListenAndServe`, it's the job of a mux to compare the requested URL to the registered patterns and call the handler function associated with the one that matches most closely.

`DefaultServeMux` is a global value of type `ServeMux`, which implements the default HTTP multiplexer logic.

Building an HTTP Server with gorilla/mux

For many web services the `net/http` and `DefaultServeMux` will be perfectly sufficient. However, sometimes you'll need the additional functionality provided by a third-party web toolkit. A popular choice is Gorilla, which, while being relatively new and less fully developed and resource-rich than something like Django or Flask, does build on Go's standard `net/http` package to provide some excellent enhancements.

The `gorilla/mux` package—one of several packages provided as part of the Gorilla web toolkit—provides an HTTP request router and dispatcher that can fully replace `DefaultServeMux`, Go's default service handler, to add several very useful enhancements to request routing and handling. We're not going to make use of these features just yet, but they will come in handy going forward. If you're curious and/or impatient, however, you can take a look at the `gorilla/mux` documentation for more information.

Creating a minimal service

Once you've done so, making use of the minimal `gorilla/mux` router is a matter of adding an import and one line of code: the initialization of a new router, which can be passed to the `handler` parameter of `ListenAndServe`:

```
package main

import (
    "log"
    "net/http"

    "github.com/gorilla/mux"
)

func helloMuxHandler(w http.ResponseWriter, r *http.Request) {
    w.Write([]byte("Hello gorilla/mux!\n"))
}

func main() {
    r := mux.NewRouter()

    r.HandleFunc("/", helloMuxHandler)

    log.Fatal(http.ListenAndServe(":8080", r))
}
```

So you should be able to just run this now with `go run`, right? Give it a try:

```
$ go run .
main.go:7:5: cannot find package "github.com/gorilla/mux" in any of:
        /go/1.15.8/libexec/src/github.com/gorilla/mux (from $GOROOT)
        /go/src/github.com/gorilla/mux (from $GOPATH)
```

It turns out that you can't (yet). Since you're now using a third-party package—a package that lives outside the standard library—you're going to have to use Go modules.

Initializing your project with Go modules

Using a package from outside the standard library requires that you make use of Go modules, which were introduced in Go 1.12 to replace an essentially nonexistent dependency management system with one that's explicit and actually quite painless to use. All of the operations that you'll use for managing your dependencies will use one of a small handful of `go mod` commands.

The first thing you're going to have to do is initialize your project. Start by creating a new, empty directory, `cd` into it, and create (or move) the Go file for your service there. Your directory should now contain only a single Go file.

Next, use the `go mod init` command to initialize the project. Typically, if a project will be imported by other projects, it'll have to be initialized with its import path. This is less important for a standalone service like ours, though, so you can be a little more lax about the name you choose. I'll just use `example.com/gorilla`; you can use whatever you like:

```
$ go mod init example.com/gorilla
go: creating new go.mod: module example.com/gorilla
```

You'll now have an (almost) empty module file, `go.mod`, in your directory:[6]

```
$ cat go.mod
module example.com/gorilla

go 1.15
```

Next, we'll want to add our dependencies, which can be done automatically using `go mod tidy`:

```
$ go mod tidy
go: finding module for package github.com/gorilla/mux
go: found github.com/gorilla/mux in github.com/gorilla/mux v1.8.0
```

If you check your `go.mod` file, you'll see that the dependency (and a version number) have been added:

6 Isn't this exciting?

```
$ cat go.mod
module example.com/gorilla

go 1.15

require github.com/gorilla/mux v1.8.0
```

Believe it or not, that's all you need. If your required dependencies change in the future you need only run `go mod tidy` again to rebuild the file. Now try again to start your service:

```
$ go run .
```

Since the service runs in the foreground, your terminal should pause. Calling the endpoint with `curl` from another terminal or browsing to it with a browser should provide the expected response:

```
$ curl http://localhost:8080
Hello gorilla/mux!
```

Success! But surely you want your service to do more than print a simple string, right? Of course you do. Read on!

Variables in URI paths

The Gorilla web toolkit provides a wealth of additional functionality over the standard `net/http` package, but one feature is particularly interesting right now: the ability to create paths with variable segments, which can even optionally contain a regular expression pattern. Using the `gorilla/mux` package, a programmer can define variables using the format `{name}` or `{name:pattern}`, as follows:

```
r := mux.NewRouter()
r.HandleFunc("/products/{key}", ProductHandler)
r.HandleFunc("/articles/{category}/", ArticlesCategoryHandler)
r.HandleFunc("/articles/{category}/{id:[0-9]+}", ArticleHandler)
```

The `mux.Vars` function conveniently allows the handler function to retrieve the variable names and values as a `map[string]string`:

```
vars := mux.Vars(request)
category := vars["category"]
```

In the next section we'll use this ability to allow clients to perform operations on arbitrary keys.

So many matchers

Another feature provided by `gorilla/mux` is that it allows a variety of *matchers* to be added to routes that let the programmer add a variety of additional matching request criteria. These include (but aren't limited to) specific domains or subdomains, path

prefixes, schemes, headers, and even custom matching functions of your own creation.

Matchers can be applied by calling the appropriate function on the `*Route` value that's returned by Gorilla's `HandleFunc` implementation. Each matcher function returns the affected `*Route`, so they can be chained. For example:

```
r := mux.NewRouter()

r.HandleFunc("/products", ProductsHandler).
    Host("www.example.com").              // Only match a specific domain
    Methods("GET", "PUT").                // Only match GET+PUT methods
    Schemes("http")                       // Only match the http scheme
```

See the `gorilla/mux` documentation for an exhaustive list of available matcher functions.

Building a RESTful Service

Now that you know how to use Go's standard HTTP library, you can use it to create a RESTful service that a client can interact with to execute call to the API you built in "Your Super Simple API" on page 5. Once you've done this you'll have implemented the absolute minimal viable key-value store.

Your RESTful methods

We're going to do our best to follow RESTful conventions, so our API will consider every key-value pair to be a distinct resource with a distinct URI that can be operated upon using the various HTTP methods. Each of our three basic operations—Put, Get, and Delete—will be requested using a different HTTP method that we summarize in Table 5-1.

The URI for your key-value pair resources will have the form `/v1/key/{key}`, where `{key}` is the unique key string. The `v1` segment indicates the API version. This convention is often used to manage API changes, and while this practice is by no means required or universal, it can be helpful for managing the impact of future changes that could break existing client integrations.

Table 5-1. Your RESTful methods

Functionality	Method	Possible Statuses
Put a key-value pair into the store	`PUT`	`201 (Created)`
Read a key-value pair from the store	`GET`	`200 (OK), 404 (Not Found)`
Delete a key-value pair	`DELETE`	`200 (OK)`

In "Variables in URI paths" on page 10, we discussed how to use the `gorilla/mux` package to register paths that contain variable segments, which will allow you to define a single variable path that handles *all* keys, mercifully freeing you from having to register every key independently. Then, in "So many matchers" on page 10, we discussed how to use route matchers to direct requests to specific handler functions based on various nonpath criteria, which you can use to create a separate handler function for each of the five HTTP methods that you'll be supporting.

Implementing the create function

Okay, you now have everything you need to get started! So, let's go ahead and implement the handler function for the creation of key-value pairs. This function has to be sure to satisfy several requirements:

- It must only match `PUT` requests for `/v1/key/{key}`.
- It must call the `Put` method from "Your Super Simple API" on page 5.
- It must respond with a `201 (Created)` when a key-value pair is created.
- It must respond to unexpected errors with a `500 (Internal Server Error)`.

All of the previous requirements are implemented in the `keyValuePutHandler` function. Note how the key's value is retrieved from the request body:

```
// keyValuePutHandler expects to be called with a PUT request for
// the "/v1/key/{key}" resource.
func keyValuePutHandler(w http.ResponseWriter, r *http.Request) {
    vars := mux.Vars(r)                    // Retrieve "key" from the request
    key := vars["key"]

    value, err := io.ReadAll(r.Body)       // The request body has our value
    defer r.Body.Close()

    if err != nil {                        // If we have an error, report it
        http.Error(w,
            err.Error(),
            http.StatusInternalServerError)
        return
    }
```

```
    err = Put(key, string(value))          // Store the value as a string
    if err != nil {                        // If we have an error, report it
        http.Error(w,
            err.Error(),
            http.StatusInternalServerError)
        return
    }

    w.WriteHeader(http.StatusCreated)      // All good! Return StatusCreated
}
```

Now that you have your "key-value create" handler function, you can register it with your Gorilla request router for the desired path and method:

```
func main() {
    r := mux.NewRouter()

    // Register keyValuePutHandler as the handler function for PUT
    // requests matching "/v1/{key}"
    r.HandleFunc("/v1/{key}", keyValuePutHandler).Methods("PUT")

    log.Fatal(http.ListenAndServe(":8080", r))
}
```

Now that you have your service put together, you can run it using `go run .` from the project root. Do that now, and send it some requests to see how it responds.

First, use our old friend `curl` to send a `PUT` containing a short snippet of text to the `/v1/key-a` endpoint to create a key named `key-a` with a value of `Hello, key-value store!`:

```
$ curl -X PUT -d 'Hello, key-value store!' -v http://localhost:8080/v1/key-a
```

Executing this command provides the following output. The complete output was quite wordy, so I've selected the relevant bits for readability:

```
> PUT /v1/key-a HTTP/1.1
< HTTP/1.1 201 Created
```

The first portion, prefixed with a greater-than symbol (`>`), shows some details about the request. The last portion, prefixed with a less-than symbol (`<`), gives details about the server response.

In this output you can see that you did in fact transmit a `PUT` to the `/v1/key-a` endpoint, and that the server responded with a `201 Created`—as expected.

What if you hit the `/v1/key-a` endpoint with an unsupported `GET` method? Assuming that the matcher function is working correctly, you should receive an error message:

```
$ curl -X GET -v http://localhost:8080/v1/key-a
> GET /v1/key-a HTTP/1.1
< HTTP/1.1 405 Method Not Allowed
```

Indeed, the server responds with a `405 Method Not Allowed` error. Everything seems to be working correctly.

Implementing the read function

Now that your service has a fully functioning `Put` method, it sure would be nice if you could read your data back! For our next trick, we're going to implement the `Get` functionality, which has the following requirements:

- It must only match `GET` requests for `/v1/key/{key}`.
- It must call the `Get` method from "Your Super Simple API" on page 5.
- It must respond with a `404 (Not Found)` when a requested key doesn't exist.
- It must respond with the requested value and a status `200` if the key exists.
- It must respond to unexpected errors with a `500 (Internal Server Error)`.

All of the previous requirements are implemented in the `keyValueGetHandler` function. Note how the value is written to `w`—the handler function's `http.ResponseWriter` parameter—after it's retrieved from the key-value API:

```
func keyValueGetHandler(w http.ResponseWriter, r *http.Request) {
    vars := mux.Vars(r)                    // Retrieve "key" from the request
    key := vars["key"]

    value, err := Get(key)                 // Get value for key
    if errors.Is(err, ErrorNoSuchKey) {
        http.Error(w,err.Error(), http.StatusNotFound)
        return
    }
    if err != nil {
        http.Error(w, err.Error(), http.StatusInternalServerError)
        return
    }

    w.Write([]byte(value))                 // Write the value to the response
}
```

And now that you have the "get" handler function, you can register it with the request router alongside the "put" handler:

```
func main() {
    r := mux.NewRouter()

    r.HandleFunc("/v1/{key}", keyValuePutHandler).Methods("PUT")
    r.HandleFunc("/v1/{key}", keyValueGetHandler).Methods("GET")

    log.Fatal(http.ListenAndServe(":8080", r))
}
```

Now let's fire up your newly improved service and see if it works:

```
$ curl -X PUT -d 'Hello, key-value store!' -v http://localhost:8080/v1/key-a
> PUT /v1/key-a HTTP/1.1
< HTTP/1.1 201 Created

$ curl -v http://localhost:8080/v1/key-a
> GET /v1/key-a HTTP/1.1
< HTTP/1.1 200 OK
Hello, key-value store!
```

It works! Now that you can get your values back, you're able to test for idempotence as well. Let's repeat the requests and make sure that you get the same results:

```
$ curl -X PUT -d 'Hello, key-value store!' -v http://localhost:8080/v1/key-a
> PUT /v1/key-a HTTP/1.1
< HTTP/1.1 201 Created

$ curl -v http://localhost:8080/v1/key-a
> GET /v1/key-a HTTP/1.1
< HTTP/1.1 200 OK
Hello, key-value store!
```

You do! But what if you want to overwrite the key with a new value? Will the subsequent `GET` have the new value? You can test that by changing the value sent by your `curl` slightly to be `Hello, again, key-value store!`:

```
$ curl -X PUT -d 'Hello, again, key-value store!' \
    -v http://localhost:8080/v1/key-a
> PUT /v1/key-a HTTP/1.1
< HTTP/1.1 201 Created

$ curl -v http://localhost:8080/v1/key-a
> GET /v1/key-a HTTP/1.1
< HTTP/1.1 200 OK
Hello, again, key-value store!
```

As expected, the `GET` responds back with a `200` status and your new value.

Finally, to complete your method set you'll just need to create a handler for the `DELETE` method. I'll leave that as an exercise, though. Enjoy!

Making Your Data Structure Concurrency-Safe

Maps in Go are not atomic and are not safe for concurrent use. Unfortunately, you now have a service designed to handle concurrent requests that's wrapped around exactly such a map.

So what do you do? Well, typically when a programmer has a data structure that needs to be read from and written to by concurrently executing goroutines, they'll use something like a mutex—also known as a lock—to act as a synchronization

mechanism. By using a mutex in this way, you can ensure that exactly one process has exclusive access to a particular resource.

Fortunately, you don't need to implement this yourself:[7] Go's `sync` package provides exactly what you need in the form of `sync.RWMutex`. The following statement uses the magic of composition to create an *anonymous struct* that contains your map and an embedded `sync.RWMutex`:

```
var myMap = struct{
    sync.RWMutex
    m map[string]string
}{m: make(map[string]string)}
```

The `myMap` struct has all of the methods from the embedded `sync.RWMutex`, allowing you to use the `Lock` method to take the write lock when you want to write to the `myMap` map:

```
myMap.Lock()                                // Take a write lock
myMap.m["some_key"] = "some_value"
myMap.Unlock()                              // Release the write lock
```

If another process has either a read or write lock, then `Lock` will block until that lock is released.

Similarly, to read from the map, you use the `RLock` method to take the read lock:

```
myMap.RLock()                               // Take a read lock
value := myMap.m["some_key"]
myMap.RUnlock()                             // Release the read lock

fmt.Println("some_key:", value)
```

Read locks are less restrictive than write locks in that any number of processes can simultaneously take read locks. However, `RLock` will block until any open write locks are released.

Integrating a read-write mutex into your application

Now that you know how to use a `sync.RWMutex` to implement a basic read-write mutex, you can go back and work it into the code you created for "Your Super Simple API" on page 5.

First, you'll want to refactor the `store` map.[8] You can construct it like `myMap`, i.e., as an anonymous struct that contains the map and an embedded `sync.RWMutex`:

7 It's a good thing too. Mutexes can be pretty tedious to implement correctly!

8 Didn't I tell you that we'd make it more complicated?

```
var store = struct{
    sync.RWMutex
    m map[string]string
}{m: make(map[string]string)}
```

Now that you have your `store` struct, you can update the `Get` and `Put` functions to establish the appropriate locks. Because `Get` only needs to *read* the `store` map, it'll use `RLock` to take a read lock only. `Put`, on the other hand, needs to *modify* the map, so it'll need to use `Lock` to take a write lock:

```
func Get(key string) (string, error) {
    store.RLock()
    value, ok := store.m[key]
    store.RUnlock()

    if !ok {
        return "", ErrorNoSuchKey
    }

    return value, nil
}

func Put(key string, value string) error {
    store.Lock()
    store.m[key] = value
    store.Unlock()

    return nil
}
```

The pattern here is clear: if a function needs to modify the map (`Put`, `Delete`), it'll use `Lock` to take a write lock. If it only needs to read existing data (`Get`), it'll use `RLock` to take a read lock. We leave the creation of the `Delete` function as an exercise for the reader.

Don't forget to release your locks, and make sure you're releasing the correct lock type!

Generation 2: Persisting Resource State

One of the stickiest challenges with distributed cloud native applications is how to handle state.

There are various techniques for distributing the state of application resources between multiple service instances, but for now we're just going to concern ourselves

with the minimum viable product and consider two ways of maintaining the state of our application:

- In "Storing State in a Transaction Log File" on page 20, you'll use a file-based *transaction log* to maintain a record of every time a resource is modified. If a service crashes, is restarted, or otherwise finds itself in an inconsistent state, a transaction log allows a service to reconstruct original state simply by replaying the transactions.
- In "Storing State in an External Database" on page 31, you'll use an external database instead of a file to store a transaction log. It might seem redundant to use a database given the nature of the application you're building, but externalizing data into another service designed specifically for that purpose is a common means of sharing state between service replicas and providing resilience.

You may be wondering why you're using a transaction log strategy to record the events when you could just use the database to store the values themselves. This makes sense when you intend to store your data in memory most of the time, only accessing your persistence mechanism in the background and at startup time.

This also affords you another opportunity: given that you're creating two different implementations of a similar functionality—a transaction log written both to a file and to a database—you can describe your functionality with an interface that both implementations can satisfy. This could come in quite handy, especially if you want to be able to choose the implementation according to your needs.

Application State Versus Resource State

The term "stateless" is used a lot in the context of cloud native architecture, and state is often regarded as a Very Bad Thing. But what is state, exactly, and why is it so bad? Does an application have to be completely devoid of any kind of state to be "cloud native"? The answer is… well, it's complicated.

First, it's important to draw a distinction between *application state* and *resource state*. These are very different things, but they're easily confused.

Application state
: Server-side data about the application or how it's being used by a client. A common example is client session tracking, such as to associate them with their access credentials or some other application context.

Resource state
: The current state of a resource within a service at any point of time. It's the same for every client, and has nothing to do with the interaction between client and server.

Any state introduces technical challenges, but application state is particularly problematic because it forces services to depend on *server affinity*—sending each of a user's requests to the same server where their session was initiated—resulting in a more complex application and making it hard to destroy or replace service replicas.

State and statelessness will be discussed in quite a bit more detail in "State and Statelessness" on page 87.

What's a Transaction Log?

In its simplest form, a *transaction log* is just a log file that maintains a history of mutating changes executed by the data store. If a service crashes, is restarted, or otherwise finds itself in an inconsistent state, a transaction log makes it possible to replay the transactions to reconstruct the service's functional state.

Transaction logs are commonly used by database management systems to provide a degree of data resilience against crashes or hardware failures. However, while this technique can get quite sophisticated, we'll be keeping ours pretty straightforward.

Your transaction log format

Before we get to the code, let's decide what the transaction log should contain.

We'll assume that your transaction log will be read only when your service is restarted or otherwise needs to recover its state, and that it'll be read from top to bottom, sequentially replaying each event. It follows that your transaction log will consist of an ordered list of mutating events. For speed and simplicity, a transaction log is also generally append-only, so when a record is deleted from your key-value store, for example, a `delete` is recorded in the log.

Given everything we've discussed so far, each recorded transaction event will need to include the following attributes:

Sequence number
: A unique record ID, in monotonically increasing order.

Event type
: A descriptor of the type of action taken; this can be `PUT` or `DELETE`.

Key
: A string containing the key affected by this transaction.

Value
: If the event is a `PUT`, the value of the transaction.

Nice and simple. Hopefully we can keep it that way.

Your transaction logger interface

The first thing we're going to do is define a `TransactionLogger` interface. For now, we're only going to define two methods: `WritePut` and `WriteDelete`, which will be used to write `PUT` and `DELETE` events, respectively, to a transaction log:

```
type TransactionLogger interface {
    WriteDelete(key string)
    WritePut(key, value string)
}
```

You'll no doubt want to add other methods later, but we'll cross that bridge when we come to it. For now, let's focus on the first implementation and add additional methods to the interface as we come across them.

Storing State in a Transaction Log File

The first approach we'll take is to use the most basic (and most common) form of transaction log, which is just an append-only log file that maintains a history of mutating changes executed by the data store. This file-based implementation has some tempting pros, but some pretty significant cons as well:

Pros:

No downstream dependency
: There's no dependency on an external service that could fail or that we can lose access to.

Technically straightforward
: The logic isn't especially sophisticated. We can be up and running quickly.

Cons:

Harder to scale
: You'll need some additional way to distribute your state between nodes when you want to scale.

Uncontrolled growth
: These logs have to be stored on disk, so you can't let them grow forever. You'll need some way of compacting them.

Prototyping your transaction logger

Before we get to the code, let's make some design decisions. First, for simplicity, the log will be written in plain text; a binary, compressed format might be more time- and space-efficient, but we can always optimize later. Second, each entry will be written on its own line; this will make it much easier to read the data later.

Finally, each transaction will include the four fields listed in "Your transaction log format" on page 19, delimited by tabs. Once again, these are:

Sequence number
: A unique record ID, in monotonically increasing order.

Event type
: A descriptor of the type of action taken; this can be `PUT` or `DELETE`.

Key
: A string containing the key affected by this transaction.

Value
: If the event is a `PUT`, the value of the transaction.

Now that we've established these fundamentals, let's go ahead and define a type, `FileTransactionLogger`, which will implicitly implement the `TransactionLogger` interface described in "Your transaction logger interface" on page 20 by defining `WritePut` and `WriteDelete` methods for writing `PUT` and `DELETE` events, respectively, to the transaction log:

```
type FileTransactionLogger struct {
    // Something, something, fields
}

func (l *FileTransactionLogger) WritePut(key, value string) {
    // Something, something, logic
}

func (l *FileTransactionLogger) WriteDelete(key string) {
    // Something, something, logic
}
```

Clearly these methods are a little light on detail, but we'll flesh them out soon!

Defining the event type

Thinking ahead, we probably want the `WritePut` and `WriteDelete` methods to operate asynchronously. You could implement that using some kind of `events` channel that some concurrent goroutine could read from and perform the log writes. That sounds like a nice idea, but if you're going to do that you'll need some kind of internal representation of an "event."

That shouldn't give you too much trouble. Incorporating all of the fields that we listed in "Your transaction log format" on page 19 gives something like the `Event` struct, in the following:

```
type Event struct {
    Sequence  uint64                // A unique record ID
    EventType EventType             // The action taken
```

```
    Key       string                // The key affected by this transaction
    Value     string                // The value of a PUT the transaction
}
```

Seems straightforward, right? `Sequence` is the sequence number, and `Key` and `Value` are self-explanatory. But…what's an `EventType`? Well, it's whatever we say it is, and we're going to say that it's a constant that we can use to refer to the different types of events, which we've already established will include one each for `PUT` and `DELETE` events.

One way to do this might be to just assign some constant `byte` values, like this:

```
const (
    EventDelete byte = 1
    EventPut    byte = 2
)
```

Sure, this would work, but Go actually provides a better (and more idiomatic) way: `iota`. `iota` is a predefined value that can be used in a `constant` declaration to construct a series of related constant values.

Declaring Constants with Iota

When used in a `constant` declaration, `iota` represents successive untyped integer constants that can be used to construct a set of related constants. Its value restarts at zero in each `constant` declaration and increments with each constant assignment (whether or not the `iota` identifier is actually referenced).

An `iota` can also be operated upon. We demonstrate this in the following by using in multiplication, left binary shift, and division operations:

```
const (
    a = 42 * iota           // iota == 0; a == 0
    b = 1 << iota           // iota == 1; b == 2
    c = 3                   // iota == 2; c == 3 (iota increments anyway!)
    d = iota / 2            // iota == 3; d == 1
)
```

Because `iota` is itself an untyped number, you can use it to make typed assignments without explicit type casts. You can even assign `iota` to a `float64` value:

```
const (
    u         = iota * 42   // iota == 0; u == 0 (untyped integer constant)
    v float64 = iota * 42   // iota == 1; v == 42.0 (float64 constant)
)
```

The `iota` keyword allows implicit repetition, which makes it trivial to create arbitrarily long sets of related constants, like we do in the following with the numbers of bytes in various digital units:

```
type ByteSize uint64

const (
    _           = iota                    // iota == 0; ignore the zero value
    KB ByteSize = 1 << (10 * iota)        // iota == 1; KB == 2^10
    MB                                    // iota == 2; MB == 2^20
    GB                                    // iota == 3; GB == 2^30
    TB                                    // iota == 4; TB == 2^40
    PB                                    // iota == 5; PB == 2^50
)
```

Using the `iota` technique, you don't have to manually assign values to constants. Instead, you can do something like the following:

```
type EventType byte

const (
    _                     = iota          // iota == 0; ignore the zero value
    EventDelete EventType = iota          // iota == 1
    EventPut                              // iota == 2; implicitly repeat
)
```

This might not be a big deal when you only have two constants like we have here, but it can come in handy when you have a number of related constants and don't want to be bothered manually keeping track of which value is assigned to what.

If you're using `iota` as enumerations in serializations (as we are here), take care to only *append* to the list, and don't reorder or insert values in the middle, or you won't be able to deserialize later.

We now have an idea of what the `TransactionLogger` will look like, as well as the two primary write methods. We've also defined a struct that describes a single event, and created a new `EventType` type and used `iota` to define its legal values. Now we're finally ready to get started.

Implementing your FileTransactionLogger

We've made some progress. We know we want a `TransactionLogger` implementation with methods for writing events, and we've created a description of an event in code. But what about the `FileTransactionLogger` itself?

The service will want to keep track of the physical location of the transaction log, so it makes sense to have an `os.File` attribute representing that. It'll also need to remember the last sequence number that was assigned so it can correctly set each event's

sequence number; that can be kept as an unsigned 64-bit integer attribute. That's great, but how will the `FileTransactionLogger` actually write the events?

One possible approach would be to keep an `io.Writer` that the `WritePut` and `WriteDelete` methods can operate on directly, but that would be a single-threaded approach, so unless you explicitly execute them in goroutines, you may find yourself spending more time in I/O than you'd like. Alternatively, you could create a buffer from a slice of `Event` values that are processed by a separate goroutine. Definitely warmer, but too complicated.

After all, why go through all of that work when we can just use standard buffered channels? Taking our own advice, we end up with a `FileTransactionLogger` and `Write` methods that look like the following:

```
type FileTransactionLogger struct {
    events       chan<- Event       // Write-only channel for sending events
    errors       <-chan error       // Read-only channel for receiving errors
    lastSequence uint64             // The last used event sequence number
    file         *os.File           // The location of the transaction log
}

func (l *FileTransactionLogger) WritePut(key, value string) {
    l.events <- Event{EventType: EventPut, Key: key, Value: value}
}

func (l *FileTransactionLogger) WriteDelete(key string) {
    l.events <- Event{EventType: EventDelete, Key: key}
}

func (l *FileTransactionLogger) Err() <-chan error {
    return l.errors
}
```

You now have your `FileTransactionLogger`, which has a `uint64` value that's used to track the last-used event sequence number, a write-only channel that receives `Event` values, and `WritePut` and `WriteDelete` methods that send `Event` values into that channel.

But it looks like there might be a part left over: there's an `Err` method there that returns a receive-only error channel. There's a good reason for that. We've already mentioned that writes to the transaction log will be done concurrently by a goroutine that receives events from the `events` channel. While that makes for a more efficient write, it also means that `WritePut` and `WriteDelete` can't simply return an `error` when they encounter a problem, so we provide a dedicated error channel to communicate errors instead.

Creating a new FileTransactionLogger

If you've followed along so far you may have noticed that none of the attributes in the `FileTransactionLogger` have been initialized. If you don't fix this issue, it's going to cause some problems. Go doesn't have constructors, though, so to solve this you need to define a construction function, which you'll call, for lack of a better name,[9] `NewFileTransactionLogger`:

```
func NewFileTransactionLogger(filename string) (TransactionLogger, error) {
    file, err := os.OpenFile(filename, os.O_RDWR|os.O_APPEND|os.O_CREATE, 0755)
    if err != nil {
        return nil, fmt.Errorf("cannot open transaction log file: %w", err)
    }

    return &FileTransactionLogger{file: file}, nil
}
```

See how `NewFileTransactionLogger` returns a pointer type, but its return list specifies the decidedly nonpointy `TransactionLogger` interface type?

The reason for this is tricksy: while Go allows pointer types to implement an interface, it doesn't allow pointers to interface types.

`NewFileTransactionLogger` calls the `os.OpenFile` function to open the file specified by the `filename` parameter. You'll notice it accepts several flags that have been binary `OR`-ed together to set its behavior:

`os.O_RDWR`
: Opens the file in read/write mode.

`os.O_APPEND`
: Any writes to this file will append, not overwrite.

`os.O_CREATE`
: If the file doesn't exist, creates it.

There are quite a few of these flags besides the three we use here. Take a look at the `os` package documentation for a full listing.

We now have a construction function that ensures that the transaction log file is correctly created. But what about the channels? We *could* create the channels and spawn a goroutine with `NewFileTransactionLogger`, but that feels like we'd be adding too much mysterious functionality. Instead, we'll create a `Run` method.

9 That's a lie. There are probably lots of better names.

Appending entries to the transaction log

As of yet, there's nothing reading from the `events` channel, which is less than ideal. What's worse, the channels aren't even initialized. Let's change this by creating a `Run` method, shown in the following:

```
func (l *FileTransactionLogger) Run() {
    events := make(chan Event, 16)              // Make an events channel
    l.events = events

    errors := make(chan error, 1)               // Make an errors channel
    l.errors = errors

    go func() {
        for e := range events {                 // Retrieve the next Event

            l.lastSequence++                    // Increment sequence number

            _, err := fmt.Fprintf(              // Write the event to the log
                l.file,
                "%d\t%d\t%s\t%s\n",
                l.lastSequence, e.EventType, e.Key, e.Value)

            if err != nil {
                errors <- err
                return
            }
        }
    }()
}
```

This implementation is incredibly basic. It won't even correctly handle entries with whitespace or multiple lines!

The `Run` function does several important things.

First, it creates a buffered `events` channel. Using a buffered channel in our `TransactionLogger` means that calls to `WritePut` and `WriteDelete` won't block as long as the buffer isn't full. This lets the consuming service handle short bursts of events without being slowed by disk I/O. If the buffer does fill up, then the write methods will block until the log writing goroutine catches up.

Second, it creates an `errors` channel, which is also buffered, that we'll use to signal any errors that arise in the goroutine that's responsible for concurrently writing events to the transaction log. The buffer value of `1` allows us to send an error in a nonblocking manner.

Finally, it starts a goroutine that retrieves `Event` values from our `events` channel and uses the `fmt.Fprintf` function to write them to the transaction log. If `fmt.Fprintf` returns an `error`, the goroutine sends the error to the `errors` channel and halts.

Using a bufio.Scanner to play back file transaction logs

Even the best transaction log is useless if it's never read.[10] But how do we do that?

You'll need to read the log from the beginning and parse each line; `io.ReadString` and `fmt.Sscanf` let you do this with minimal fuss.

Channels, our dependable friends, will let your service stream the results to a consumer as it retrieves them. This might be starting to feel routine, but stop for a second to appreciate it. In most other languages the path of least resistance here would be to read in the entire file, stash it in an array, and finally loop over that array to replay the events. Go's convenient concurrency primitives make it almost trivially easy to stream the data to the consumer in a much more space- and memory-efficient way.

The `ReadEvents` method[11] demonstrates this:

```
func (l *FileTransactionLogger) ReadEvents() (<-chan Event, <-chan error) {
    scanner := bufio.NewScanner(l.file)      // Create a Scanner for l.file
    outEvent := make(chan Event)             // An unbuffered Event channel
    outError := make(chan error, 1)          // A buffered error channel

    go func() {
        var e Event

        defer close(outEvent)                // Close the channels when the
        defer close(outError)                // goroutine ends

        for scanner.Scan() {
            line := scanner.Text()

            if err := fmt.Sscanf(line, "%d\t%d\t%s\t%s",
                &e.Sequence, &e.EventType, &e.Key, &e.Value); err != nil {

                outError <- fmt.Errorf("input parse error: %w", err)
                return
            }

            // Sanity check! Are the sequence numbers in increasing order?
            if l.lastSequence >= e.Sequence {
                outError <- fmt.Errorf("transaction numbers out of sequence")
```

10 What makes a transaction log "good" anyway?

11 Naming is hard.

```
                return
            }

            l.lastSequence = e.Sequence     // Update last used sequence #

            outEvent <- e                   // Send the event along
        }

        if err := scanner.Err(); err != nil {
            outError <- fmt.Errorf("transaction log read failure: %w", err)
            return
        }
    }()

    return outEvent, outError
}
```

The `ReadEvents` method can really be said to be two functions in one: the outer function initializes the file reader, and creates and returns the event and error channels. The inner function runs concurrently to ingest the file contents line by line and send the results to the channels.

Interestingly, the `file` attribute of `TransactionLogger` is of type `*os.File`, which has a `Read` method that satisfies the `io.Reader` interface. `Read` is fairly low-level, but, if you wanted to, you could actually use it to retrieve the data. The `bufio` package, however, gives us a better way: the `Scanner` interface, which provides a convenient means for reading newline-delimited lines of text. We can get a new `Scanner` value by passing an `io.Reader`—an `os.File` in this case—to `bufio.NewScanner`.

Each call to the `scanner.Scan` method advances it to the next line, returning `false` if there aren't any lines left. A subsequent call to `scanner.Text` returns the line.

Note the `defer` statements in the inner anonymous goroutine. These ensure that the output channels are always closed. Because `defer` is scoped to the function they're declared in, these get called at the end of the goroutine, not `ReadEvents`.

You may recall from Chapter 3 that the `fmt.Sscanf` function provides a simple (but sometimes simplistic) means of parsing simple strings. Like the other methods in the `fmt` package, the expected format is specified using a format string with various "verbs" embedded: two digits (`%d`) and two strings (`%s`), separated by tab characters (`\t`). Conveniently, `fmt.Sscanf` lets you pass in pointers to the target values for each verb, which it can update directly.[12]

12 After all this time, I still think that's pretty neat.

Go's format strings have a long history dating back to C's `printf` and `scanf`, but they've been adopted by many other languages over the years, including C++, Java, Perl, PHP, Ruby, and Scala. You may already be familiar with them, but if you're not, take a break now to look at the `fmt` package documentation.

At the end of each loop the last-used sequence number is updated to the value that was just read, and the event is sent on its merry way. A minor point: note how the same `Event` value is reused on each iteration rather than creating a new one. This is possible because the `outEvent` channel is sending struct values, not *pointers* to struct values, so it already provides copies of whatever value we send into it.

Finally, the function checks for `Scanner` errors. The `Scan` method returns only a single boolean value, which is really convenient for looping. Instead, when it encounters an error, `Scan` returns `false` and exposes the error via the `Err` method.

Your transaction logger interface (redux)

Now that you've implemented a fully functional `FileTransactionLogger`, it's time to look back and see which of the new methods we can use to incorporate into the `TransactionLogger` interface. It actually looks like there are quite few we might like to keep in any implementation, leaving us with the following final form for the `TransactionLogger` interface:

```
type TransactionLogger interface {
    WriteDelete(key string)
    WritePut(key, value string)
    Err() <-chan error

    ReadEvents() (<-chan Event, <-chan error)

    Run()
}
```

Now that that's settled, you can finally start integrating the transaction log into your key-value service.

Initializing the FileTransactionLogger in your web service

The `FileTransactionLogger` is now complete! All that's left to do now is to integrate it with your web service. The first step of this is to add a new function that can create a new `TransactionLogger` value, read in and replay any existing events, and call `Run`.

First, let's add a `TransactionLogger` reference to our `service.go`. You can call it `logger` because naming is hard:

```
var logger TransactionLogger
```

Now that you have that detail out of the way, you can define your initialization method, which can look like the following:

```
func initializeTransactionLog() error {
    var err error

    logger, err = NewFileTransactionLogger("transaction.log")
    if err != nil {
        return fmt.Errorf("failed to create event logger: %w", err)
    }

    events, errors := logger.ReadEvents()
    e, ok := Event{}, true

    for ok && err == nil {
        select {
        case err, ok = <-errors:               // Retrieve any errors
        case e, ok = <-events:
            switch e.EventType {
            case EventDelete:                  // Got a DELETE event!
                err = Delete(e.Key)
            case EventPut:                     // Got a PUT event!
                err = Put(e.Key, e.Value)
            }
        }
    }

    logger.Run()

    return err
}
```

This function starts as you'd expect: it calls `NewFileTransactionLogger` and assigns it to `logger`.

The next part is more interesting: it calls `logger.ReadEvents`, and replays the results based on the `Event` values received from it. This is done by looping over a `select` with cases for both the `events` and `errors` channels. Note how the cases in the `select` use the format `case foo, ok = <-ch`. The `bool` returned by a channel read in this way will be `false` if the channel in question has been closed, setting the value of `ok` and terminating the `for` loop.

If we get an `Event` value from the `events` channel, we call either `Delete` or `Put` as appropriate; if we get an error from the `errors` channel, `err` will be set to a non-`nil` value and the `for` loop will be terminated.

Integrating FileTransactionLogger with your web service

Now that the initialization logic is put together, all that's left to do to complete the integration of the `TransactionLogger` is add exactly three function calls into the web

service. This is fairly straightforward, so we won't walk through it here. But, briefly, you'll need to add the following:

- `initializeTransactionLog` to the `main` method
- `logger.WriteDelete` to `keyValueDeleteHandler`
- `logger.WritePut` to `keyValuePutHandler`

We'll leave the actual integration as an exercise for the reader.[13]

Future improvements

We may have completed a minimal viable implementation of our transaction logger, but it still has plenty of issues and opportunities for improvement, such as:

- There aren't any tests.
- There's no `Close` method to gracefully close the file.
- The service can close with events still in the write buffer: events can get lost.
- Keys and values aren't encoded in the transaction log: multiple lines or white-space will fail to parse correctly.
- The sizes of keys and values are unbound: huge keys or values can be added, filling the disk.
- The transaction log is written in plain text: it will take up more disk space than it probably needs to.
- The log retains records of deleted values forever: it will grow indefinitely.

All of these would be impediments in production. I encourage you to take the time to consider—or even implement—solutions to one or more of these points.

Storing State in an External Database

Databases, and data, are at the core of many, if not most, business and web applications, so it makes perfect sense that Go includes a standard interface for SQL (or SQL-like) databases in its core libraries.

But does it make sense to use a SQL database to back our key-value store? After all, isn't it redundant for our data store to just depend on another data store? Yes, certainly. But externalizing a service's data into another service designed specifically for that purpose—a database—is a common pattern that allows state to be shared

13 You're welcome.

between service replicas and provides data resilience. Besides, the point is to show how you might interact with a database, not to design the perfect application.

In this section, you'll be implementing a transaction log backed by an external database and satisfying the `TransactionLogger` interface, just as you did in "Storing State in a Transaction Log File" on page 20. This would certainly work, and even have some benefits as mentioned previously, but it comes with some tradeoffs:

Pros:

Externalizes application state
: Less need to worry about distributed state and closer to "cloud native."

Easier to scale
: Not having to share data between replicas makes scaling out *easier* (but not *easy*).

Cons:

Introduces a bottleneck
: What if you had to scale way up? What if all replicas had to read from the database at once?

Introduces an upstream dependency
: Creates a dependency on another resource that might fail.

Requires initialization
: What if the `Transactions` table doesn't exist?

Increases complexity
: Yet another thing to manage and configure.

Working with databases in Go

Databases, particularly SQL and SQL-like databases, are everywhere. You can try to avoid them, but if you're building applications with some kind of data component, you will at some point have to interact with one.

Fortunately for us, the creators of the Go standard library provided the `database/sql` package, which provides an idiomatic and lightweight interface around SQL (and SQL-like) databases. In this section we'll briefly demonstrate how to use this package, and point out some of the gotchas along the way.

Among the most ubiquitous members of the `database/sql` package is `sql.DB`: Go's primary database abstraction and entry point for creating statements and transactions, executing queries, and fetching results. While it doesn't, as its name might suggest, map to any particular concept of a database or schema, it does do quite a lot of things for you, including, but not limited to, negotiating connections with your database and managing a database connection pool.

We'll get into how you create your `sql.DB` in a bit. But first, we have to talk about database drivers.

Importing a database driver

While the `sql.DB` type provides a common interface for interacting with a SQL database, it depends on database drivers to implement the specifics for particular database types. At the time of this writing there are 45 drivers listed in the Go repository.

In the following section we'll be working with a Postgres database, so we'll use the third-party `lib/pq` Postgres driver implementation.

To load a database driver, anonymously import the driver package by aliasing its package qualifier to _. This triggers any initializers the package might have while also informing the compiler that you have no intention of directly using it:

```
import (
    "database/sql"
    _ "github.com/lib/pq"       // Anonymously import the driver package
)
```

Now that you've done this, you're finally ready to create your `sql.DB` value and access the database.

Implementing your PostgresTransactionLogger

Previously, we presented the `TransactionLogger` interface, which provides a standard definition for a generic transaction log. You might recall that it defined methods for starting the logger, as well as reading and writing events to the log, as detailed here:

```
type TransactionLogger interface {
    WriteDelete(key string)
    WritePut(key, value string)
    Err() <-chan error

    ReadEvents() (<-chan Event, <-chan error)

    Run()
}
```

Our goal now is to create a database-backed implementation of `TransactionLogger`. Fortunately, much of our work is already done for us. Looking back at "Implementing your FileTransactionLogger" on page 23 for guidance, it looks like we can create a `PostgresTransactionLogger` using very similar logic.

Starting with the `WritePut`, `WriteDelete`, and `Err` methods, you can do something like the following:

```
type PostgresTransactionLogger struct {
    events       chan<- Event       // Write-only channel for sending events
    errors       <-chan error       // Read-only channel for receiving errors
    db           *sql.DB            // The database access interface
}

func (l *PostgresTransactionLogger) WritePut(key, value string) {
    l.events <- Event{EventType: EventPut, Key: key, Value: value}
}

func (l *PostgresTransactionLogger) WriteDelete(key string) {
    l.events <- Event{EventType: EventDelete, Key: key}
}

func (l *PostgresTransactionLogger) Err() <-chan error {
    return l.errors
}
```

If you compare this to the `FileTransactionLogger` it's clear that the code is nearly identical. All we've really changed is:

- Renaming (obviously) the type to `PostgresTransactionLogger`
- Swapping the `*os.File` for a `*sql.DB`
- Removing `lastSequence`; you can let the database handle the sequencing

Creating a new PostgresTransactionLogger

That's all well and good, but we still haven't talked about how we create the `sql.DB`. I know how you must feel. The suspense is definitely killing me, too.

Much like we did in the `NewFileTransactionLogger` function, we're going to create a construction function for our `PostgresTransactionLogger`, which we'll call (quite predictably) `NewPostgresTransactionLogger`. However, instead of opening a file like `NewFileTransactionLogger`, it'll establish a connection with the database, returning an `error` if it fails.

There's a little bit of a wrinkle, though. Namely, that the setup for a Postgres connection takes a lot of parameters. At the bare minimum we need to know the host where the database lives, the name of the database, and the user name and password. One way to deal with this would be to create a function like the following, which simply accepts a bunch of string parameters:

```
func NewPostgresTransactionLogger(host, dbName, user, password string)
    (TransactionLogger, error) { ... }
```

This approach is pretty ugly, though. Plus, what if you wanted an additional parameter? Do you chunk it onto the end of the parameter list, breaking any code that's

already using this function? Maybe worse, the parameter order isn't clear without looking at the documentation.

There has to be a better way. So, instead of this potential horror show, you can create a small helper struct:

```
type PostgresDBParams struct {
    dbName   string
    host     string
    user     string
    password string
}
```

Unlike the big-bag-of-strings approach, this struct is small, readable, and easily extended. To use it, you can create a `PostgresDBParams` variable and pass it to your construction function. Here's what that looks like:

```
logger, err = NewPostgresTransactionLogger(PostgresDBParams{
    host:     "localhost",
    dbName:   "kvs",
    user:     "test",
    password: "hunter2"
})
```

The new construction function looks something like the following:

```
func NewPostgresTransactionLogger(config PostgresDBParams) (TransactionLogger,
    error) {

    connStr := fmt.Sprintf("host=%s dbname=%s user=%s password=%s",
        config.host, config.dbName, config.user, config.password)

    db, err := sql.Open("postgres", connStr)
    if err != nil {
        return nil, fmt.Errorf("failed to open db: %w", err)
    }

    err = db.Ping()                 // Test the database connection
    if err != nil {
        return nil, fmt.Errorf("failed to open db connection: %w", err)
    }

    logger := &PostgresTransactionLogger{db: db}

    exists, err := logger.verifyTableExists()
    if err != nil {
        return nil, fmt.Errorf("failed to verify table exists: %w", err)
    }
    if !exists {
        if err = logger.createTable(); err != nil {
            return nil, fmt.Errorf("failed to create table: %w", err)
        }
    }
```

```
    return logger, nil
}
```

This does quite a few things, but fundamentally it is not very different from `NewFileTransactionLogger`.

The first thing it does is to use `sql.Open` to retrieve a `*sql.DB` value. You'll note that the connection string passed to `sql.Open` contains several parameters; the `lib/pq` package supports many more than the ones listed here. See the package documentation for a complete listing.

Many drivers, including `lib/pq`, don't actually create a connection to the database immediately, so it uses `db.Ping` to force the driver to establish and test a connection.

Finally, it creates the `PostgresTransactionLogger` and uses that to verify that the `transactions` table exists, creating it if necessary. Without this step, the `PostgresTransactionLogger` will essentially assume that the table already exists, and will fail if it doesn't.

You may have noticed that the `verifyTableExists` and `createTable` methods aren't implemented here. This is entirely intentional. As an exercise, you're encouraged to dive into the `database/sql` docs and think about how you might go about doing that. If you'd prefer not to, you can find an implementation in the GitHub repository that comes with this book.

You now have a construction function that establishes a connection to the database and returns a newly created `TransactionLogger`. But, once again, you need to get things started. For that, you need to implement the `Run` method that will create the `events` and `errors` channels and spawn the event ingestion goroutine.

Using db.Exec to execute a SQL INSERT

For the `FileTransactionLogger`, you implemented a `Run` method that initialized the channels and created the go function responsible for writing to the transaction log.

The `PostgresTransactionLogger` is very similar. However, instead of appending a line to a file, the new logger uses `db.Exec` to execute an SQL `INSERT` to accomplish the same result:

```
func (l *PostgresTransactionLogger) Run() {
    events := make(chan Event, 16)              // Make an events channel
    l.events = events

    errors := make(chan error, 1)               // Make an errors channel
    l.errors = errors

    go func() {                                 // The INSERT query
        query := `INSERT INTO transactions
```

```
            (event_type, key, value)
            VALUES ($1, $2, $3)`

        for e := range events {                 // Retrieve the next Event

            _, err := l.db.Exec(                // Execute the INSERT query
                query,
                e.EventType, e.Key, e.Value)

            if err != nil {
                errors <- err
            }
        }
    }()
}
```

This implementation of the `Run` method does almost exactly what its `FileTransactionLogger` equivalent does: it creates the buffered `events` and `errors` channels, and it starts a goroutine that retrieves `Event` values from our `events` channel and writes them to the transaction log.

Unlike the `FileTransactionLogger`, which appends to a file, this goroutine uses `db.Exec` to execute a SQL query that appends a row to the `transactions` table. The numbered arguments (`$1, $2, $3`) in the query are placeholder query parameters, which must be satisfied when the `db.Exec` function is called.

Using db.Query to play back postgres transaction logs

In "Using a bufio.Scanner to play back file transaction logs" on page 27, you used a `bufio.Scanner` to read previously written transaction log entries.

The Postgres implementation won't be *quite* as straightforward, but the principle is the same: you point at the top of your data source and read until you hit the bottom:

```
func (l *PostgresTransactionLogger) ReadEvents() (<-chan Event, <-chan error) {
    outEvent := make(chan Event)               // An unbuffered events channel
    outError := make(chan error, 1)            // A buffered errors channel

    go func() {
        defer close(outEvent)                  // Close the channels when the
        defer close(outError)                  // goroutine ends

        query := `SELECT sequence, event_type, key, value FROM transactions
                  ORDER BY sequence`

        rows, err := db.Query(query)           // Run query; get result set
        if err != nil {
            outError <- fmt.Errorf("sql query error: %w", err)
            return
        }
```

```
        defer rows.Close()                      // This is important!

        e := Event{}                            // Create an empty Event

        for rows.Next() {                       // Iterate over the rows

            err = rows.Scan(                    // Read the values from the
                &e.Sequence, &e.EventType,      // row into the Event.
                &e.Key, &e.Value)

            if err != nil {
                outError <- fmt.Errorf("error reading row: %w", err)
                return
            }

            outEvent <- e                       // Send e to the channel
        }

        err = rows.Err()
        if err != nil {
            outError <- fmt.Errorf("transaction log read failure: %w", err)
        }
    }()

    return outEvent, outError
}
```

All of the interesting (or at least new) bits are happening in the goroutine. Let's break them down:

- `query` is a string that contains the SQL query. The query in this code requests four columns: `sequence`, `event_type`, `key`, and `value`.
- `db.Query` sends `query` to the database, and returns values of type `*sql.Rows` and `error`.
- We defer a call to `rows.Close`. Failing to do so can lead to connection leakage!
- `rows.Next` lets us iterate over the rows; it returns `false` if there are no more rows or if there's an error.
- `rows.Scan` copies the columns in the current row into the values we pointed at in the call.
- We send event `e` to the output channel.
- `Err` returns the error, if any, that may have caused `rows.Next` to return `false`.

Initializing the PostgresTransactionLogger in your web service

The `PostgresTransactionLogger` is pretty much complete. Now let's go ahead and integrate it into the web service.

Fortunately, since we already had the `FileTransactionLogger` in place, we only need to change one line:

```
logger, err = NewFileTransactionLogger("transaction.log")
```

which becomes…

```
logger, err = NewPostgresTransactionLogger("localhost")
```

Yup. That's it. Really.

Because this represents a complete implementation of the `TransactionLogger` interface, everything else stays exactly the same. You can interact with the `PostgresTransactionLogger` using exactly the same methods as before.

Future improvements

As with the `FileTransactionLogger`, the `PostgresTransactionLogger` represents a minimal viable implementation of a transaction logger and has lots of room for improvement. Some of the areas for improvement include, but are certainly not limited to:

- We assume that the database and table exist, and we'll get errors if they don't.
- The connection string is hard-coded. Even the password.
- There's still no `Close` method to clean up open connections.
- The service can close with events still in the write buffer: events can get lost.
- The log retains records of deleted values forever: it will grow indefinitely.

All of these would be (major) impediments in production. I encourage you to take the time to consider—or even implement—solutions to one or more of these points.

Generation 3: Implementing Transport Layer Security

Security. Love it or hate it, the simple fact is that security is a critical feature of *any* application, cloud native or otherwise. Sadly, security is often treated as an afterthought, with potentially catastrophic consequences.

There are rich tools and established security best practices for traditional environments, but this is less true of cloud native applications, which tend to take the form of several small, often ephemeral, microservices. While this architecture provides significant flexibility and scalability benefits, it also creates a distinct opportunity for would-be attackers: every communication between services is transmitted across a network, opening it up to eavesdropping and manipulation.

The subject of security can take up an entire book of its own,[14] so we'll focus on one common technique: encryption. Encrypting data "in transit" (or "on the wire") is commonly used to guard against eavesdropping and message manipulation, and any language worth its salt—including, and especially, Go—will make it relatively low-lift to implement.

Transport Layer Security

Transport Layer Security (TLS) is a cryptographic protocol that's designed to provide communications security over a computer network. Its use is ubiquitous and widespread, being applicable to virtually any Internet communications. You're most likely familiar with it (and perhaps using it right now) in the form of HTTPS—also known as HTTP over TLS—which uses TLS to encrypt exchanges over HTTP.

TLS encrypts messages using *public-key cryptography*, in which both parties possess their own *key pair*, which includes a *public key* that's freely given out, and a *private key* that's known only to its owner, illustrated in Figure 5-2. Anybody can use a public key to encrypt a message, but it can only be decrypted with the corresponding private key. Using this protocol, two parties that wish to communicate privately can exchange their public keys, which can then be used to secure all subsequent communications in a way that can only be read by the owner of the intended recipient, who holds the corresponding private key.[15]

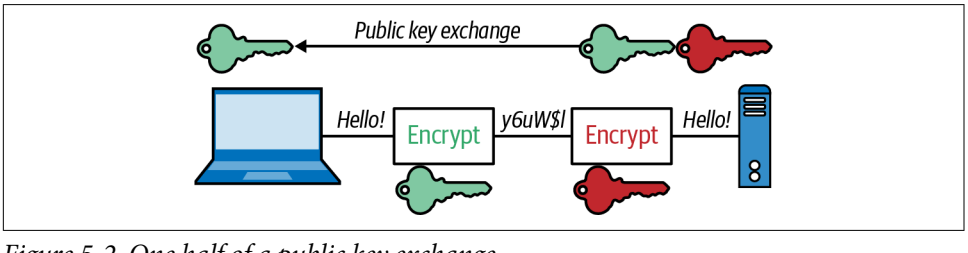

Figure 5-2. One half of a public key exchange

Certificates, certificate authorities, and trust

If TLS had a motto, it would be "trust but verify." Actually, scratch the trust part. Verify everything.

It's not enough for a service to simply provide a public key.[16] Instead, every public key is associated with a *digital certificate*, an electronic document used to prove the key's

14 Ideally written by somebody who knows more than I do about security.

15 This is a gross over-simplification, but it'll do for our purposes. I encourage you to learn more about this and correct me, though.

16 You don't know where that key has been.

ownership. A certificate shows that the owner of the public key is, in fact, the named subject (owner) of the certificate, and describes how the key may be used. This allows the recipient to compare the certificate against various "trusts" to decide whether it will accept it as valid.

First, the certificate must be digitally signed and authenticated by a *certificate authority*, a trusted entity that issues digital certificates.

Second, the subject of the certificate has to match the domain name of the service the client is trying to connect to. Among other things, this helps to ensure that the certificates you're receiving are valid and haven't been swapped out by a man-in-the-middle.

Only then will your conversation proceed.

Web browsers or other tools will usually allow you to choose to proceed if a certificate can't be validated. If you're using self-signed certificates for development, for example, that might make sense. But generally speaking, heed the warnings.

Private Key and Certificate Files

TLS (and its predecessor, SSL) has been around long enough[17] that you'd think that we'd have settled on a single key container format, but you'd be wrong. Web searches for "key file format" will return a virtual zoo of file extensions: *.csr*, *.key*, *.pkcs12*, *.der*, and *.pem* just to name a few.

Of these, however, *.pem* seems to be the most common. It also happens to be the format that's most easily supported by Go's `net/http` package, so that's what we'll be using.

Privacy enhanced mail (PEM) file format

Privacy enhanced mail (PEM) is a common certificate container format, usually stored in *.pem* files, but *.cer* or *.crt* (for certificates) and *.key* (for public or private keys) are common too. Conveniently, PEM is also base64 encoded and therefore viewable in a text editor, and even safe to paste into (for example) the body of an email message.[18]

17 SSL 2.0 was released in 1995 and TLS 1.0 was released in 1999. Interestingly, SSL 1.0 had some pretty profound security flaws and was never publicly released.

18 Public keys only, please.

Often, *.pem* files will come in a pair, representing a complete key pair:

cert.pem
: The server certificate (including the CA-signed public key).

key.pem
: A private key, not to be shared.

Going forward, we'll assume that your keys are in this configuration. If you don't yet have any keys and need to generate some for development purposes, instructions are available in multiple places online. If you already have a key file in some other format, converting it is beyond the scope of this book. However, the Internet is a magical place, and there are plenty of tutorials online for converting between common key formats.

Securing Your Web Service with HTTPS

So, now that we've established that security should be taken seriously, and that communication via TLS is a bare-minimum first step towards securing our communications, how do we go about doing that?

One way might be to put a reverse proxy in front of our service that can handle HTTPS requests and forward them to our key-value service as HTTP, but unless the two are co-located on the same server, we're still sending unencrypted messages over a network. Plus, the additional service adds some architectural complexity that we might prefer to avoid. Perhaps we can have our key-value service serve HTTPS?

Actually, we can. Going all the way back to "Building an HTTP Server with net/http" on page 6, you might recall that the `net/http` package contains a function, `ListenAndServe`, which, in its most basic form, looks something like the following:

```
func main() {
    http.HandleFunc("/", helloGoHandler)            // Add a root path handler

    http.ListenAndServe(":8080", nil)               // Start the HTTP server
}
```

In this example, we call `HandleFunc` to add a handler function for the root path, followed by `ListenAndServe` to start the service listening and serving. For the sake of simplicity, we ignore any errors returned by `ListenAndServe`.

There aren't a lot of moving parts here, which is kind of nice. In keeping with that philosophy, the designers of `net/http` kindly provided a TLS-enabled variant of the `ListenAndServe` function that we're familiar with:

```
func ListenAndServeTLS(addr, certFile, keyFile string, handler Handler) error
```

As you can see, `ListenAndServeTLS` looks and feels almost exactly like `ListenAndServe` except that it has two extra parameters: `certFile` and `keyFile`. If you happen to have certificate and private key PEM files, then service HTTPS-encrypted connections is just a matter of passing the names of those files to `ListenAndServeTLS`:

```
http.ListenAndServeTLS(":8080", "cert.pem", "key.pem", nil)
```

This sure looks super convenient, but does it work? Let's fire up our service (using self-signed certificates) and find out.

Dusting off our old friend `curl`, let's try inserting a key/value pair. Note that we use the `https` scheme in our URL instead of `http`:

```
$ curl -X PUT -d 'Hello, key-value store!' -v https://localhost:8080/v1/key-a
* SSL certificate problem: self signed certificate
curl: (60) SSL certificate problem: self signed certificate
```

Well, that didn't go as planned. As we mentioned in "Certificates, certificate authorities, and trust" on page 40, TLS expects any certificates to be signed by a certificate authority. It doesn't like self-signed certificates.

Fortunately, we can turn this safety check off in `curl` with the appropriately named `--insecure` flag:

```
$ curl -X PUT -d 'Hello, key-value store!' --insecure -v \
    https://localhost:8080/v1/key-a
* SSL certificate verify result: self signed certificate (18), continuing anyway.
> PUT /v1/key-a HTTP/2
< HTTP/2 201
```

We got a sternly worded warning, but it worked!

Transport Layer Summary

We've covered quite a lot in just a few pages. The topic of security is vast, and there's no way we're going to do it justice, but we were able to at least introduce TLS, and how it can serve as one relatively low-cost, high-return component of a larger security strategy.

We were also able to demonstrate how to implement TLS in an Go `net/http` web service, and saw how—as long as we have valid certificates—to secure a service's communications without a great deal of effort.

Containerizing Your Key-Value Store

A *container* is a lightweight operating-system-level virtualization[19] abstraction that provides processes with a degree of isolation, both from their host and from other containers. The concept of the container has been around since at least 2000, but it was the introduction of Docker in 2013 that made containers accessible to the masses and brought containerization into the mainstream.

Importantly, containers are not virtual machines:[20] they don't use hypervisors, and they share the host's kernel rather than carrying their own guest operating system. Instead, their isolation is provided by a clever application of several Linux kernel features, including `chroot`, cgroups, and kernel namespaces. In fact, it can be reasonably argued that containers are nothing more than a convenient abstraction, and that there's actually no such thing as a container.

Even though they're not virtual machines,[21] containers do provide some virtual-machine-like benefits. The most obvious of which is that they allow an application, its dependencies, and much of its environment to be packaged within a single distributable artifact—a container image—that can be executed on any suitable host.

The benefits don't stop there, however. In case you need them, here's a few more:

Agility
: Unlike virtual machines that are saddled with an entire operating system and a colossal memory footprint, containers boast image sizes in the megabyte range and startup times that measure in milliseconds. This is particularly true of Go applications, whose binaries have few, if any, dependencies.

Isolation
: This was hinted at previously, but bears repeating. Containers virtualize CPU, memory, storage, and network resources at the operating-system-level, providing developers with a sandboxed view of the OS that is logically isolated from other applications.

Standardization and productivity
: Containers let you package an application alongside its dependencies, such as specific versions of language runtimes and libraries, as a single distributable binary, making your deployments reproducible, predictable, and versionable.

19 Containers are not virtual machines. They virtualize the operating system instead of hardware.

20 Repetition intended. This is an important point.

21 Yup. I said it. Again.

Orchestration
: Sophisticated container orchestration systems like Kubernetes provide a huge number of benefits. By containerizing your application(s) you're taking the first step towards being able to take advantage of them.

There are just four (very) motivating arguments.[22] In other words, containerization is super, super useful.

For this book, we'll be using Docker to build our container images. Alternative build tools exist, but Docker is the most common containerization tool in use today, and the syntax for its build file—termed a *Dockerfile*—lets you use familiar shell scripting commands and utilities.

That being said, this isn't a book about Docker or containerization, so our discussion will mostly be limited to the bare basics of using Docker with Go. If you're interested in learning more, I suggest picking up a copy of *Docker: Up & Running: Shipping Reliable Containers in Production* by Sean P. Kane and Karl Matthias (O'Reilly).

Docker (Absolute) Basics

Before we continue, it's important to draw a distinction between container images and the containers themselves. A *container image* is essentially an executable binary that contains your application runtime and its dependencies. When an image is run, the resulting process is the *container*. An image can be run many times to create multiple (essentially) identical containers.

Over the next few pages we'll create a simple Dockerfile and build and execute an image. If you haven't already, please take a moment and install the Docker Community Edition (CE).

The Dockerfile

Dockerfiles are essentially build files that describe the steps required to build an image. A very minimal—but complete—example is demonstrated in the following:

```
# The parent image. At build time, this image will be pulled and
# subsequent instructions run against it.
FROM ubuntu:20.04

# Update apt cache and install nginx without an approval prompt.
RUN apt-get update && apt-get install --yes nginx

# Tell Docker this image's containers will use port 80.
EXPOSE 80
```

22 The initial draft had several more, but this chapter is already pretty lengthy.

```
# Run Nginx in the foreground. This is important: without a
# foreground process the container will automatically stop.
CMD ["nginx", "-g", "daemon off;"]
```

As you can see, this Dockerfile includes four different commands:

`FROM`
: Specifies a *base image* that this build will extend, and will typically be a common Linux distribution, such as `ubuntu` or `alpine`. At build time this image is pulled and run, and the subsequent commands applied to it.

`RUN`
: Will execute any commands on top of the current image. The result will be used for the next step in the Dockerfile.

`EXPOSE`
: Tells Docker which port(s) the container will use. See "What's the Difference Between Exposing and Publishing Ports?" on page 48 for more information on exposing ports.

`CMD`
: The command to execute when the container is executed. There can only be one `CMD` in a Dockerfile.

These are four of the most common Dockerfile instructions of many available. For a complete listing see the official Dockerfile reference.

As you may have inferred, the previous example starts with an existing Linux distribution image (Ubuntu 20.04) and installs Nginx, which is executed when the container is started.

By convention, the file name of a Dockerfile is *Dockerfile*. Go ahead and create a new file named *Dockerfile* and paste the previous example into it.

Building your container image

Now that you have a simple Dockerfile, you can build it! Make sure that you're in the same directory as your Dockerfile and enter the following:

```
$ docker build --tag my-nginx .
```

This will instruct Docker to begin the build process. If everything works correctly (and why wouldn't it?) you'll see the output as Docker downloads the parent image, and runs the `apt` commands. This will probably take a minute or two the first time you run it.

At the end, you'll see a line that looks something like the following: `Successfully tagged my-nginx:latest`.

If you do, you can use the `docker images` command to verify that your image is now present. You should see something like the following:

```
$ docker images
REPOSITORY      TAG         IMAGE ID          CREATED              SIZE
my-nginx        latest      64ea3e21a388      29 seconds ago       159MB
ubuntu          20.04       f63181f19b2f      3 weeks ago          72.9MB
```

If all has gone as planned, you'll see at least two images listed: our parent image `ubuntu:20.04`, and your own `my-nginx:latest` image. Next step: running the service container!

What Does latest Mean?

Note the name of the image. What's `latest`? That's a simple question with a complicated answer. Docker images have two name components: a `repository` and a `tag`.

The repository name component can include the domain name of a host where the image is stored (or will be stored). For example, the repository name for an image hosted by FooCorp may be named something like `docker.foo.com/ubuntu`. If no repository URL is evident, then the image is either 100% local (like the image we just built) or lives in the Docker Hub.

The `tag` component is intended as a unique label for a particular version of an image, and often takes the form of a version number. The `latest` tag is a default tag name that's added by many `docker` operations if no tag is specified.

Using `latest` in production is generally considered a bad practice, however, because its contents can change—sometimes significantly—with unfortunate consequences.

Running your container image

Now that you've built your image, you can run it. For that, you'll use the `docker run` command:

```
$ docker run --detach --publish 8080:80 --name nginx my-nginx
61bb4d01017236f6261ede5749b421e4f65d43cb67e8e7aa8439dc0f06afe0f3
```

This instructs Docker to run a container using your `my-nginx` image. The `--detach` flag will cause the container to be run in the background. Using `--publish 8080:80` instructs Docker to publish port 8080 on the host bridged to port 80 in the container, so any connections to `localhost:8080` will be forwarded to the container's port 80. Finally, the `--name nginx` flag specifies a name for the container; without this, a randomly generated name will be assigned instead.

You'll notice that running this command presents us with a very cryptic line containing 65 very cryptic hexadecimal characters. This is the *container ID*, which can be used to refer to the container in lieu of its name.

What's the Difference Between Exposing and Publishing Ports?

The difference between "exposing" and "publishing" container ports can be confusing, but there's actually an important distinction:

- *Exposing* ports is a way of clearly documenting—both to users and to Docker—which ports a container uses. It does not map or open any ports on the host. Ports can be exposed using the `EXPOSE` keyword in the Dockerfile or the `--expose` flag to `docker run`.
- *Publishing* ports tells Docker which ports to open on the container's network interface. Ports can be published using the `--publish` or `--publish-all` flag to `docker run`, which create firewall rules that map a container port to a port on the host.

Running your container image

To verify that your container is running and is doing what you expect, you can use the `docker ps` command to list all running containers. This should look something like the following:

```
$ docker ps
CONTAINER ID    IMAGE       STATUS          PORTS                   NAMES
4cce9201f484    my-nginx    Up 4 minutes    0.0.0.0:8080->80/tcp    nginx
```

The preceding output has been edited for brevity (you may notice that it's missing the `COMMAND` and `CREATED` columns). Your output should include seven columns:

`CONTAINER ID`
: The first 12 characters of the container ID. You'll notice it matches the output of your `docker run`.

`IMAGE`
: The name (and tag, if specified) of this container's source image. No tag implies `latest`.

`COMMAND` *(not shown)*
: The command running inside the container. Unless overridden in the `docker run` this will be the same as the `CMD` instruction in the Dockerfile. In our case this will be `nginx -g 'daemon off;'`.

`CREATED` *(not shown)*
: How long ago the container was created.

`STATUS`
: The current state of the container (`up`, `exited`, `restarting`, etc) and how long it's been in that state. If the state changed, then the time will differ from `CREATED`.

`PORTS`
: Lists all exposed and published ports (see "What's the Difference Between Exposing and Publishing Ports?" on page 48). In our case, we've published `0.0.0.0:8080` on the host and mapped it to `80` on the container, so that all requests to host port 8080 are forwarded to container port 80.

`NAMES`
: The name of the container. Docker will randomly set this if it's not explicitly defined. Two containers with the same name, regardless of state, cannot exist on the same host at the same time. To reuse a name, you'll first have to `delete` the unwanted container.

Issuing a request to a published container port

If you've gotten this far, then your `docker ps` output should show a container named `nginx` that appears to have port 8080 published and forwarding to the container's port 80. If so, then you're ready to send a request to your running container. But which port should you query?

Well, the Nginx container is listening on port 80. Can you reach that? Actually, no. That port won't be accessible because it wasn't published to any network interface during the `docker run`. Any attempt to connect to an unpublished container port is doomed to failure:

```
$ curl localhost:80
curl: (7) Failed to connect to localhost port 80: Connection refused
```

You haven't published to port 80, but you *have* published port 8080 and forwarded it to the container's port 80. You can verify this with our old friend `curl` or by browsing to `localhost:8080`. If everything is working correctly you'll be greeted with the familiar Nginx "Welcome" page illustrated in Figure 5-3.

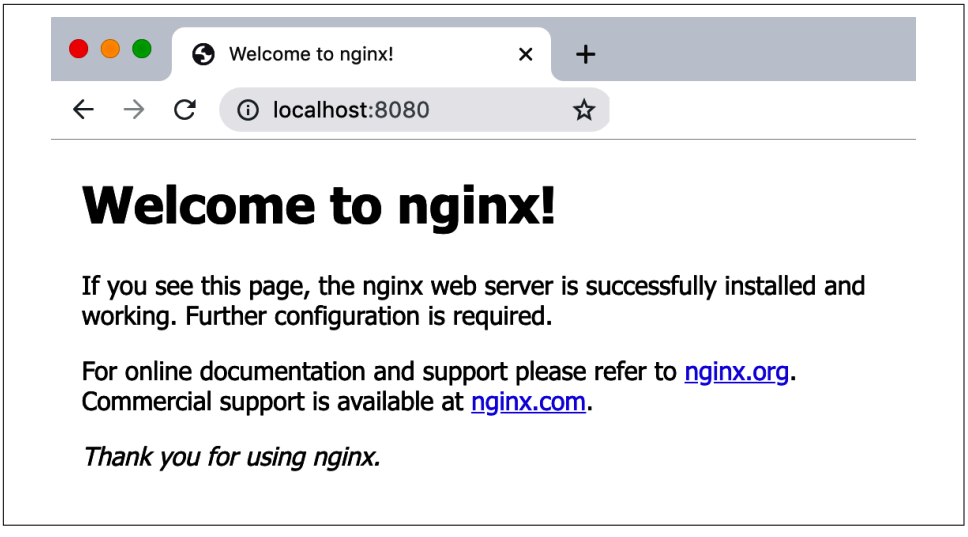

Figure 5-3. Welcome to nginx!

Running multiple containers

One of the "killer features" of containerization is this: because all of the containers on a host are isolated from one another, it's possible to run quite a lot of them—even ones that contain different technologies and stacks—on the same host, with each listening on a different published port. For example, if you wanted to run an `httpd` container alongside your already-running `my-nginx` container, you could do exactly that.

"But," you might say, "both of those containers expose port 80! Won't they collide?"

Great question, to which the answer is, happily, no. In fact, you can actually have as many containers as you want that *expose* the same port—even multiple instances of the same image—as long as they don't attempt to *publish* the same port on the same network interface.

For example, if you want to run the stock `httpd` image, you can run it by using the `docker run` command again, as long as you take care to publish to a different port (8081, in this case):

```
$ docker run --detach --publish 8081:80 --name httpd httpd
```

If all goes as planned, this will spawn a new container listening on the host at port 8081. Go ahead: use `docker ps` and `curl` to test:

```
$ curl localhost:8081
<html><body><h1>It works!</h1></body></html>
```

Stopping and deleting your containers

Now you've successfully run your container, you'll probably need to stop and delete it at some point, particularly if you want to rerun a new container using the same name.

To stop a running container, you can use the `docker stop` command, passing it either the container name or the first few characters of its container ID (how many characters doesn't matter, as long they can be used to uniquely identify the desired container). Using the container ID to stop our `nginx` container looks like this:

```
$ docker stop 4cce      # "docker stop nginx" will work too
4cce
```

The output of a successful `docker stop` is just the name or ID that we passed into the command. You can verify that your container has actually been stopped using `docker ps --all`, which will show *all* containers, not just the running ones:

```
$ docker ps
CONTAINER ID    IMAGE       STATUS                      PORTS    NAMES
4cce9201f484    my-nginx    Exited (0) 3 minutes ago             nginx
```

If you ran the `httpd` container, it will also be displayed with a status of `Up`. You will probably want to stop it as well.

As you can see, the status of our `nginx` container has changed to `Exited`, followed by its exit code—an exit status of 0 indicates that we were able to execute a graceful shutdown—and how long ago the container entered its current status.

Now that you've stopped your container you can freely delete it.

You can't delete a running container or an image that's used by a running container.

To do this, you use the `docker rm` (or the newer `docker container rm`) command to remove your container, again passing it either the container name or the first few characters of the ID of the container you want to delete:

```
$ docker rm 4cce            # "docker rm nginx" will work too
4cce
```

As before, the output name or ID indicates success. If you were to go ahead and run `docker ps --all` again, you shouldn't see the container listed anymore.

Building Your Key-Value Store Container

Now that you have the basics down, you can start applying them to containerizing our key-value service.

Fortunately, Go's ability to compile into statically linked binaries makes it especially well suited for containerization. While most other languages have to be built into a parent image that contains the language runtime, like the 486MB `openjdk:15` for Java or the 885MB `python:3.9` for Python,[23] Go binaries need no runtime at all. They can be placed into a "scratch" image: an image, with no parent at all.

Iteration 1: adding your binary to a FROM scratch image

To do this, you'll need a Dockerfile. The following example is a pretty typical example of a Dockerfile for a containerized Go binary:

```
# We use a "scratch" image, which contains no distribution files. The
# resulting image and containers will have only the service binary.
FROM scratch

# Copy the existing binary from the host.
COPY kvs .

# Copy in your PEM files.
COPY *.pem .

# Tell Docker we'll be using port 8080.
EXPOSE 8080

# Tell Docker to execute this command on a `docker run`.
CMD ["/kvs"]
```

This Dockerfile is fairly similar to the previous one, except that instead of using `apt` to install an application from a repository, it uses `COPY` to retrieve a compiled binary from the filesystem it's being built on. In this case, it assumes the presence of a binary named `kvs`. For this to work, we'll need to build the binary first.

In order for your binary to be usable inside a container, it has to meet a few criteria:

- It has to be compiled (or cross-compiled) for Linux.
- It has to be statically linked.
- It has to be named `kvs` (because that's what the Dockerfile is expecting).

We can do all of these things in one command, as follows:

23 To be fair, these images are "only" 240MB and 337MB compressed, respectively.

```
$ CGO_ENABLED=0 GOOS=linux go build -a -o kvs
```

Let's walk through what this does:

- `CGO_ENABLED=0` tells the compiler to disable `cgo` and statically link any C bindings. We won't go into what this is, other than that it enforces static linking, but I encourage you to look at the cgo documentation if you're curious.
- `GOOS=linux` instructs the compiler to generate a Linux binary, cross-compiling if necessary.
- `-a` forces the compiler to rebuild any packages that are already up to date.
- `-o kvs` specifies that the binary will be named `kvs`.

Executing the command should yield a statically linked Linux binary. This can be verified using the `file` command:

```
$ file kvs
kvs: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), statically linked,
not stripped
```

Linux binaries will run in a Linux container, even one running in Docker for MacOS or Windows, but won't run on MacOS or Windows otherwise.

Great! Now let's build the container image, and see what comes out:

```
$ docker build --tag kvs .
...output omitted.

$ docker images
REPOSITORY     TAG        IMAGE ID           CREATED              SIZE
kvs            latest     7b1fb6fa93e3       About a minute ago   6.88MB
node           15         ebcfbb59a4bd       7 days ago           936MB
python         3.9        2a93c239d591       8 days ago           885MB
openjdk        15         7666c92f41b0       2 weeks ago          486MB
```

Less than 7MB! That's roughly two orders of magnitude smaller than the relatively massive images for other languages' runtimes. This can come in quite handy when you're operating at scale and have to pull your image onto a couple hundred nodes a few times a day.

But does it run? Let's find out:

```
$ docker run --detach --publish 8080:8080 kvs
4a05617539125f7f28357d3310759c2ef388f456b07ea0763350a78da661afd3

$ curl -X PUT -d 'Hello, key-value store!' -v http://localhost:8080/v1/key-a
> PUT /v1/key-a HTTP/1.1
```

```
< HTTP/1.1 201 Created

$ curl http://localhost:8080/v1/key-a
Hello, key-value store!
```

Looks like it works!

So now you have a nice, simple Dockerfile that builds an image using a precompiled binary. Unfortunately, that means that you have to make sure that you (or your CI system) rebuilds the binary fresh for each Docker build. That's not *too* terrible, but it does mean that you need to have Go installed on your build workers. Again, not terrible, but we can certainly do better.

Iteration 2: using a multi-stage build

In the last section, you created a simple Dockerfile that would take an existing Linux binary and wrap it in a bare-bones "scratch" image. But what if you could perform the *entire* image build—Go compilation and all—in Docker?

One approach might be to use the `golang` image as our parent image. If you did that, your Dockerfile could compile your Go code and run the resulting binary at deploy time. This could build on hosts that don't have the Go compiler installed, but the resulting image would be saddled with an additional 862MB (the size of the `golang:1.16` image) of entirely unnecessary build machinery.

Another approach might be to use two Dockerfiles: one for building the binary, and another that containerizes the output of the first build. This is a lot closer to where you want to be, but it requires two distinct Dockerfiles that need be sequentially built or managed by a separate script.

A better way became available with the introduction of multistage Docker builds, which allow multiple distinct builds—even with entirely different base images—to be chained together so that artifacts from one stage can be selectively copied into another, leaving behind everything you don't want in the final image. To use this approach, you define a build with two stages: a "build" stage that generates the Go binary, and an "image" stage that uses that binary to produce the final image.

To do this, you use multiple `FROM` statements in our Dockerfile, each defining the start of a new stage. Each stage can be arbitrarily named. For example, you might name your build stage `build`, as follows:

```
FROM golang:1.16 as build
```

Once you have stages with names, you can use the `COPY` instruction in your Dockerfile to copy any artifact *from any previous stage*. Your final stage might have an instruction like the following, which copies the file `/src/kvs` from the `build` stage to the current working directory:

```
COPY --from=build /src/kvs .
```

Putting these things together yields a complete, two-stage Dockerfile:

```
# Stage 1: Compile the binary in a containerized Golang environment
#
FROM golang:1.16 as build

# Copy the source files from the host
COPY . /src

# Set the working directory to the same place we copied the code
WORKDIR /src

# Build the binary!
RUN CGO_ENABLED=0 GOOS=linux go build -o kvs


# Stage 2: Build the Key-Value Store image proper
#
# Use a "scratch" image, which contains no distribution files
FROM scratch

# Copy the binary from the build container
COPY --from=build /src/kvs .

# If you're using TLS, copy the .pem files too
COPY --from=build /src/*.pem .

# Tell Docker we'll be using port 8080
EXPOSE 8080

# Tell Docker to execute this command on a "docker run"
CMD ["/kvs"]
```

Now that you have your complete Dockerfile, you can build it in precisely the same way as before. We'll tag it as `multipart` this time, though, so that you can compare the two images:

```
$ docker build --tag kvs:multipart .
...output omitted.

$ docker images
REPOSITORY     TAG           IMAGE ID          CREATED              SIZE
kvs            latest        7b1fb6fa93e3      2 hours ago          6.88MB
kvs            multipart     b83b9e479ae7      4 minutes ago        6.56MB
```

This is encouraging! You now have a single Dockerfile that can compile your Go code—regardless of whether or not the Go compiler is even installed on the build worker—and that drops the resulting statically linked executable binary into a `FROM scratch` base to produce a very, very small image containing nothing except your key-value store service.

You don't need to stop there, though. If you wanted to, you could add other stages as well, such as a `test` stage that runs any unit tests prior to the build step. We won't go through that exercise now, however, since it's more of the same thing, but I encourage you to try it for yourself.

Externalizing Container Data

Containers are intended to be ephemeral, and any container should be designed and run with the understanding that it can (and will) be destroyed and recreated at any time, taking all of its data with it. To be clear, this is a feature, and is very intentional, but sometimes you might *want* your data to outlive your containers.

For example, the ability to mount externally managed files directly into the filesystem of an otherwise general-purpose container can decouple configurations from images so you don't have to rebuild them whenever just to change their settings. This is a very powerful strategy, and is probably the most common use-case for container data externalization. So common in fact that Kubernetes even provides a resource type—`ConfigMap`—dedicated to it.

Similarly, you might want data generated in a container to exist beyond the lifetime of the container. Storing data on the host can be an excellent strategy for warming caches, for example. It's important to keep in mind, however, one of the realities of cloud native infrastructure: nothing is permanent, not even servers. Don't store anything on the host that you don't mind possibly losing forever.

Fortunately, while "pure" Docker limits you to externalizing data directly onto local disk,[24] container orchestration systems like Kubernetes provides various abstractions that allows data to survive the loss of a host.

Unfortunately, this is supposed to be a book about Go, so we really can't cover Kubernetes in detail here. But if you haven't already, I strongly encourage you to take a long look at the excellent Kubernetes documentation, and equally excellent *Kubernetes: Up and Running* by Brendan Burns, Joe Beda, and Kelsey Hightower (O'Reilly).

24 I'm intentionally ignoring solutions like Amazon's Elastic Block Store, which can help, but have issues of their own.

Summary

This was a long chapter, and we touched on a lot of different topics. Consider how much we've accomplished!

- Starting from first principles, we designed and implemented a simple monolithic key-value store, using `net/http` and `gorilla/mux` to build a RESTful service around functionality provided by a small, independent, and easily testable Go library.
- We leveraged Go's powerful interface capabilities to produce two completely different transaction logger implementations, one based on local files and using `os.File` and the `fmt` and `bufio` packages; the other backed by a Postgres database and using the `database/sql` and `github.com/lib/pq` Postgres driver packages.
- We discussed the importance of security in general, covered some of the basics of TLS as one part of a larger security strategy, and implemented HTTPS in our service.
- Finally, we covered containerization, one of the core cloud native technologies, including how to build images and how to run and manage containers. We even containerized not only our application, but we even containerized its build process.

Going forward, we'll be extending on our key-value service in various ways when we introduce new concepts, so stay tuned. Things are about to get even more interesting.

CHAPTER 6
It's All About Dependability

> The most important property of a program is whether it accomplishes the intention of its user.[1]
>
> —C.A.R. Hoare, Communications of the ACM (October 1969)

Professor Sir Charles Antony Richard (Tony) Hoare is a brilliant guy. He invented quicksort, authored Hoare Logic for reasoning about the correctness of computer programs, and created the formal language "communicating sequential processes" (CSP) that inspired Go's beloved concurrency model. Oh, and he developed the structured programming paradigm[2] that forms the foundation of all modern programming languages in common use today. He also invented the null reference. Please don't hold that against him, though. He publicly apologized[3] for it in 2009, calling it his "billion-dollar mistake."

Tony Hoare literally invented programming as we know it. So when he says that the single most important property of a program is whether it accomplishes the intention of its user, you can take that on some authority. Think about this for a second: Hoare specifically (and quite rightly) points out that it's the intention of a program's *users*—not its *creators*—that dictates whether a program is performing correctly. How inconvenient that the intentions of a program's users aren't always the same as those of its creator!

1 Hoare, C.A.R. "An Axiomatic Basis for Computer Programming.". *Communications of the ACM*, vol. 12, no. 10, October 1969, pp. 576–583. *https://oreil.ly/jOwO9*.

2 When Edsger W. Dijkstra coined the expression "GOTO considered harmful," he was referencing Hoare's work in structured programming.

3 Hoare, Tony. "Null References: The Billion Dollar Mistake." *InfoQ.com*. 25 August 2009. *https://oreil.ly/4QWS8*.

Given this assertion, it stands to reason that a user's first expectation about a program is that *the program works*. But when is a program "working"? This is actually a pretty big question, one that lies at the heart of cloud native design. The first goal of this chapter is to explore that very idea, and in the process, introduce concepts like "dependability" and "reliability" that we can use to better describe (and meet) user expectations. Finally, we'll briefly review a number of practices commonly used in cloud native development to ensure that services meet the expectations of its users. We'll discuss each of these in-depth throughout the remainder of this book.

What's the Point of Cloud Native?

In Chapter 1 we spent a few pages defining "cloud native," starting with the Cloud Native Computing Foundation's definition and working forward to the properties of an ideal cloud native service. We spent a few more pages talking about the pressures that have driven cloud native to be a thing in the first place.

What we didn't spend so much time on, however, was the *why* of cloud native. Why does the concept of cloud native even exist? Why would we even want our systems to be cloud native? What's its purpose? What makes it so special? Why should I care?

So, why *does* cloud native exist? The answer is actually pretty straightforward: it's all about dependability. In the first part of this chapter, we'll dig into the concept of dependability, what it is, why it's important, and how it underlies all the patterns and techniques that we call cloud native.

It's All About Dependability

Holly Cummins, the worldwide development community practice lead for the IBM Garage, famously said that "if cloud native has to be a synonym for anything, it would be idempotent."[4] Cummins is absolutely brilliant, and has said a lot of absolutely brilliant things,[5] but I think she only has half of the picture on this one. I think that idempotence is very important—perhaps even necessary for cloud native—but not sufficient. I'll elaborate.

The history of software, particularly the network-based kind, has been one of struggling to meet the expectations of increasingly sophisticated users. Long gone are the days when a service could go down at night "for maintenance." Users today rely heavily on the services they use, and they expect those services to be available and to respond promptly to their requests. Remember the last time you tried to start a Netflix movie and it took the longest five seconds of your life? Yeah, that.

4 *Cloud Native Is About Culture, Not Containers*. Cummins, Holly. Cloud Native London 2018.

5 If you ever have a chance to see her speak, I strongly recommend you take it.

Users don't care that your services have to be maintained. They won't wait patiently while you hunt down that mysterious source of latency. They just want to finish binge-watching the second season of Breaking Bad.[6]

All of the patterns and techniques that we associate with cloud native—*every single one*—exist to allow services to be deployed, operated, and maintained at scale in unreliable environments, driven by the need to produce dependable services that keep users happy.

In other words, I think that if "cloud native" has to be a synonym for anything, it would be "dependability."

What Is Dependability and Why Is It So Important?

I didn't choose the word "dependability" arbitrarily. It's actually a core concept in the field of *systems engineering*, which is full of some very smart people who say some very smart things about the design and management of complex systems. The concept of dependability in a computing context was first rigorously defined by Jean-Claude Laprie about 35 years ago,[7] who defined a system's dependability according to the expectations of its users. Laprie's original definition has been tweaked and extended over the years by various authors, but here's my favorite:

> The dependability of a computer system is its ability to avoid failures that are more frequent or more severe, and outage durations that are longer, than is acceptable to the user(s).[8]
>
> —Fundamental Concepts of Computer System Dependability (2001)

In other words, a dependable system consistently does what its users expect and can be quickly fixed when it doesn't.

By this definition, a system is dependable only when it can *justifiably* be trusted. Obviously, a system can't be considered dependable if it falls over any time one of its components glitch, or if it requires hours to recover from a failure. Even if it's been running for months without interruption, an undependable system may still be one bad day away from catastrophe: lucky isn't dependable.

Unfortunately, it's hard to objectively gauge "user expectations." For this reason, as illustrated in Figure 6-1, dependability is an umbrella concept encompassing several

6 Remember what Walt did to Jane that time? That was so messed up.

7 Laprie, J-C. "Dependable Computing and Fault Tolerance: Concepts and Terminology." *FTCS-15 The 15th Int'l Symposium on Fault-Tolerant Computing*, June 1985, pp. 2–11. *https://oreil.ly/UZFFY*.

8 A. Avižienis, J. Laprie, and B. Randell. "Fundamental Concepts of Computer System Dependability." *Research Report No. 1145, LAAS-CNRS*, April 2001. *https://oreil.ly/4YXd1*.

more specific and quantifiable attributes—availability, reliability, and maintainability—all of which are subject to similar threats that may be overcome by similar means.

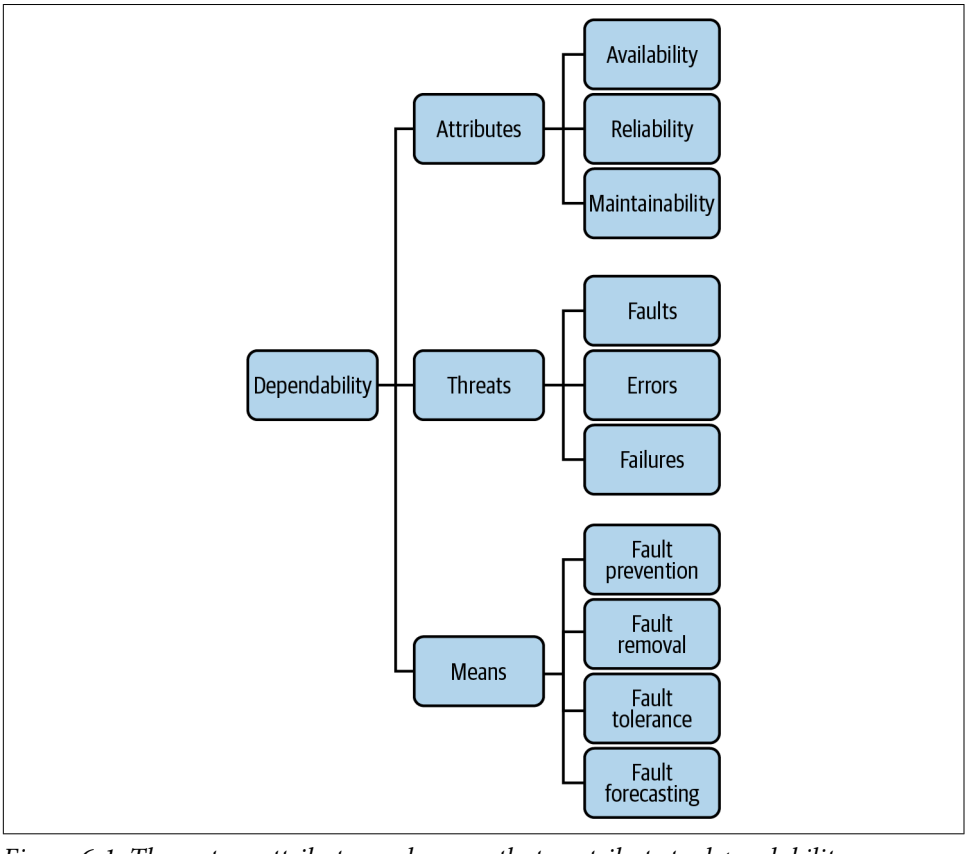

Figure 6-1. The system attributes and means that contribute to dependability

So while the concept of "dependability" alone might be a little squishy and subjective, the attributes that contribute to it are quantitative and measurable enough to be useful:

Availability
: The ability of a system to perform its intended function at a random moment in time. This is usually expressed as the probability that a request made of the system will be successful, defined as uptime divided by total time.

Reliability
: The ability of a system to perform its intended function for a given time interval. This is often expressed as either the mean time between failures (MTBF: total time divided by the number of failures) or failure rate (number of failures divided by total time).

Maintainability
: The ability of a system to undergo modifications and repairs. There are a variety of indirect measures for maintainability, ranging from calculations of cyclomatic complexity to tracking the amount of time required to change a system's behavior to meet new requirements or to restore it to a functional state.

Later authors extended Laprie's definition of dependability to include several security-related properties, including safety, confidentiality, and integrity. I've reluctantly omitted these, not because security isn't important (it's *SO* important!), but for brevity. A worthy discussion of security would require an entire book of its own.

Dependability Is Not Reliability

If you've read any of O'Reilly's Site Reliability Engineering (SRE) books[9] you've already heard quite a lot about reliability. However, as illustrated in Figure 6-1, reliability is just one property that contributes to overall dependability.

If that's true, though, then why has reliability become the standard metric for service functionality? Why are there "site reliability engineers" but no "site dependability engineers"?

There are probably several answers to these questions, but perhaps the most definitive is that the definition of "dependability" is purely qualitative. There's no measure for it, and when you can't measure something it's very hard to construct a set of rules around it.

Reliability, on the other hand, is quantitative. Given a robust definition[10] for what it means for a system to provide "correct" service, it becomes relatively straightforward to calculate that system's "reliability," making it a powerful (if indirect) measure of user experience.

Dependability: It's Not Just for Ops Anymore

Since the introduction of networked services, it's been the job of developers to build services, and of systems administrators ("operations") to deploy those services onto servers and keep them running. This worked well enough for a time, but it had the unfortunate side-effect of incentivizing developers to prioritize feature development at the expense of stability and operations.

9 If you haven't, start with *Site Reliability Engineering: How Google Runs Production Systems*. It really is very good.

10 Many organizations use service-level objectives (SLOs) for precisely this purpose.

Fortunately, over the past decade or so—coinciding with the DevOps movement—a new wave of technologies has become available with the potential to completely change the way technologists of all kinds do their jobs.

On the operations side, with the availability of infrastructure and platforms as a service (IaaS/PaaS) and tools like Terraform and Ansible, working with infrastructure has never been more like writing software.

On the development side, the popularization of technologies like containers and serverless functions has given developers an entire new set of "operations-like" capabilities, particularly around virtualization and deployment.

As a result, the once-stark line between software and infrastructure is getting increasingly blurry. One could even argue that with the growing advancement and adoption of infrastructure abstractions like virtualization, container orchestration frameworks like Kubernetes, and software-defined behavior like service meshes, we may even be at the point where they could be said to have merged. Everything is software now.

The ever-increasing demand for service dependability has driven the creation of a whole new generation of cloud native technologies. The effects of these new technologies and the capabilities they provide has been considerable, and the traditional developer and operations roles are changing to suit them. At long last, the silos are crumbling, and, increasingly, the rapid production of dependable, high-quality services is a fully collaborative effort of all of its designers, implementors, and maintainers.

Achieving Dependability

This is where the rubber meets the road. If you've made it this far, congratulations.

So far we've discussed Laprie's definition of "dependability," which can be (very) loosely paraphrased as "happy users," and we've discussed the attributes—availability, reliability, and maintainability—that contribute to it. This is all well and good, but without actionable advice for how to achieve dependability the entire discussion is purely academic.

Laprie thought so too, and defined four broad categories of techniques that can be used together to improve a system's dependability (or which, by their absence, can reduce it):

Fault prevention
: Fault prevention techniques are used during system construction to prevent the occurrence or introduction of faults.

Fault tolerance
: Fault tolerance techniques are used during system design and implementation to prevent service failures in the presence of faults.

Fault removal
: Fault removal techniques are used to reduce the number and severity of faults.

Fault forecasting
: Fault forecasting techniques are used to identify the presence, the creation, and the consequences of faults.

Interestingly, as illustrated in Figure 6-2, these four categories correspond surprisingly well to the five cloud native attributes that we introduced all the way back in Chapter 1.

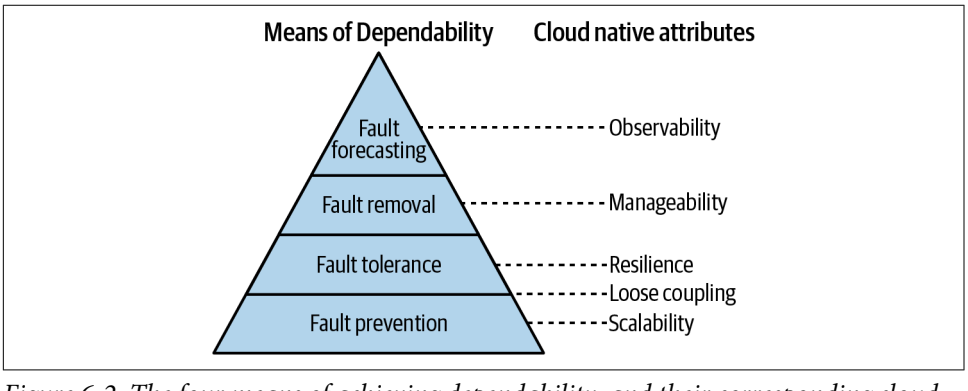

Figure 6-2. The four means of achieving dependability, and their corresponding cloud native attributes

Fault prevention and fault tolerance make up the bottom two layers of the pyramid, corresponding with scalability, loose coupling, and resilience. Designing a system for scalability prevents a variety of faults common among cloud native applications, and resiliency techniques allow a system to tolerate faults when they do inevitably arise. Techniques for loose coupling can be said to fall into both categories, preventing and enhancing a service's fault tolerance. Together these can be said to contribute to what Laprie terms *dependability procurement*: the means by which a system is provided with the ability to perform its designated function.

Techniques and designs that contribute to manageability are intended to produce a system that can be easily modified, simplifying the process of removing faults when they're identified. Similarly, observability naturally contributes to the ability to forecast faults in a system. Together fault removal and forecasting techniques contribute to what Laprie termed *dependability validation*: the means by which confidence is gained in a system's ability to perform its designated function.

Consider the implications of this relationship: what was a purely academic exercise 35 years ago has essentially been rediscovered—apparently independently—as a natural consequence of years of accumulated experience building reliable production systems. Dependability has come full-circle.

In the subsequent sections we'll explore these relationships more fully and preview later chapters, in which we discuss exactly how these two apparently disparate systems actually correspond quite closely.

Fault Prevention

At the base of our "Means of Dependability" pyramid are techniques that focus on preventing the occurrence or introduction of faults. As veteran programmers can attest, many—if not most—classes of errors and faults can be predicted and prevented during the earliest phases of development. As such, many fault prevention techniques come into play during the design and implementation of a service.

Good programming practices

Fault prevention is one of the primary goals of software engineering in general, and is the explicit goal of any development methodology, from pair programming to test-driven development and code review practices. Many such techniques can really be grouped into what might be considered to be "good programming practice," about which innumerable excellent books and articles have already been written, so we won't explicitly cover it here.

Language features

Your choice of language can also greatly affect your ability to prevent or fix faults. Many language features that some programmers have sometimes come to expect, such as dynamic typing, pointer arithmetic, manual memory management, and thrown exceptions (to name a few) can easily introduce unintended behaviors that are difficult to find and fix, and may even be maliciously exploitable.

These kinds of features strongly motivated many of the design decisions for Go, resulting in the strongly typed garbage-collected language we have today. For a refresher for why Go is particularly well suited for the development of cloud native services, take a look back at Chapter 2.

Scalability

We briefly introduced the concept of scalability way back in Chapter 1, where it was defined as the ability of a system to continue to provide correct service in the face of significant changes in demand.

In that section we introduced two different approaches to scaling—vertical scaling (scaling up) by resizing existing resources, and horizontal scaling (scaling out) by adding (or removing) service instances—and some of the pros and cons of each.

We'll go quite a bit deeper into each of these in Chapter 7, especially into the gotchas and downsides. We'll also talk a lot about the problems posed by state.[11] For now, though, it'll suffice to say that having to scale your service adds quite a bit of overhead, including but not limited to cost, complexity, and debugging.

While scaling resources is eventually often inevitable, it's often better (and cheaper!) to resist the temptation to throw hardware at the problem and postpone scaling events as long as possible by considering runtime efficiency and algorithmic scaling. As such, we'll cover a number of Go features and tooling that allow us to identify and fix common problems like memory leaks and lock contention that tend to plague systems at scale.

Loose coupling

Loose coupling, which we first defined in Chapter 1, is the system property and design strategy of ensuring that a system's components have as little knowledge of other components as possible. The degree of coupling between services can have an enormous—and too often under-appreciated—impact on a system's ability to scale and to isolate and tolerate failures.

Since the beginning of microservices there have been dissenters who point to the difficulty of deploying and maintaining microservice-based systems as evidence that such architectures are just too complex to be viable. I don't agree, but I can see where they're coming from, given how incredibly easy it is to build a *distributed monolith*. The hallmark of a distributed monolith is the tight coupling between its components, which results in an application saddled with all of the complexity of microservices plus the all of the tangled dependencies of the typical monolith. If you have to deploy most of your services together, or if a failed health check sends cascading failures through your entire system, you probably have a distributed monolith.

Building a loosely coupled system is easier said than done, but is possible with a little discipline and reasonable boundaries. In Chapter 8 we'll cover how to use data exchange contracts to establish those boundaries, and different synchronous and asynchronous communication models and architectural patterns and packages used to implement them and avoid the dreaded distributed monolith.

11 Application state is hard, and when done wrong it's poison to scalability.

Fault Tolerance

Fault tolerance has a number of synonyms—self-repair, self-healing, resilience—that all describe a system's ability to detect errors and prevent them from cascading into a full-blown failure. Typically, this consists of two parts: *error detection*, in which an error is discovered during normal service, and *recovery*, in which the system is returned to a state where it can be activated again.

Perhaps the most common strategy for providing resilience is redundancy: the duplication of critical components (having multiple service replicas) or functions (retrying service requests). This is a broad and very interesting field with a number of subtle gotchas that we'll dig into in Chapter 9.

Fault Removal

Fault removal, the third of the four dependability means, is the process of reducing the number and severity of faults—latent software flaws that can cause errors—before they manifest as errors.

Even under ideal conditions, there are plenty of ways that a system can error or otherwise misbehave. It might fail to perform an expected action, or perform the wrong action entirely, perhaps maliciously. Just to make things even more complicated, conditions aren't always—or often—ideal.

Many faults can be identified by testing, which allows you to verify that the system (or at least its components) behaves as expected under known test conditions.

But what about unknown conditions? Requirements change, and the real world doesn't care about your test conditions. Fortunately, with effort, a system can be designed to be manageable enough that its behavior can often be adjusted to keep it secure, running smoothly, and compliant with changing requirements.

We'll briefly discuss these next.

Verification and testing

There are exactly four ways of finding latent software faults in your code: testing, testing, testing, and bad luck.

Yes, I joke, but that's not so far from the truth: if you don't find your software faults, your users will. If you're lucky. If you're not, then they'll be found by bad actors seeking to take advantage of them.

Bad jokes aside, there are two common approaches to finding software faults in development:

Static analysis
: Automated, rule-based code analysis performed without actually executing programs. Static analysis is useful for providing early feedback, enforcing consistent practices, and finding common errors and security holes without depending on human knowledge or effort.

Dynamic analysis
: Verifying the correctness of a system or subsystem by executing it under controlled conditions and evaluating its behavior. More commonly referred to simply as "testing."

Key to software testing is having software that's *designed for testability* by minimizing the *degrees of freedom*—the range of possible states—of its components. Highly testable functions have a single purpose, with well-defined inputs and outputs and few or no *side effects*; that is, they don't modify variables outside of their scope. If you'll forgive the nerdiness, this approach minimizes the *search space*—the set of all possible solutions—of each function.

Testing is a critical step in software development that's all too often neglected. The Go creators understood this and baked unit testing and benchmarking into the language itself in the form of the `go test` command and the testing package. Unfortunately, a deep dive into testing theory is well beyond the scope of this book, but we'll do our best to scratch the surface in Chapter 9.

Manageability

Faults exist when your system doesn't behave according to requirements. But what happens when those requirements change?

Designing for *manageability*, first introduced back in Chapter 1, allows a system's behavior to be adjusted without code changes. A manageable system essentially has "knobs" that allow real-time control to keep your system secure, running smoothly, and compliant with changing requirements.

Manageability can take a variety of forms, including (but not limited to!) adjusting and configuring resource consumption, applying on-the-fly security remediations, *feature flags* that can turn features on or off, or even loading plug-in-defined behaviors.

Clearly, manageability is a broad topic. We'll review a few of the mechanisms Go provides for it in Chapter 10.

Fault Forecasting

At the peak of our "Means of Dependability" pyramid (Figure 6-2) is *fault forecasting*, which builds on the knowledge gained and solutions implemented in the levels below it to attempt to estimate the present number, the future incidence, and the likely consequence of faults.

Too often this consists of guesswork and gut feelings instead, generally resulting in unexpected failures when a starting assumption stops being true. More systematic approaches include Failure Mode and Effects Analysis and stress testing, which are very useful for understanding a system's possible failure modes.

In a system designed for *observability*, which we'll discuss in depth in Chapter 11, failure mode indicators can be tracked so that they can be forecast and corrected before they manifest as errors. Furthermore, when unexpected failures occur—as they inevitably will—observable systems allow the underlying faults to be quickly identified, isolated, and corrected.

The Continuing Relevance of the Twelve-Factor App

In the early 2010s, developers at Heroku, a platform as a service (PaaS) company and early cloud pioneer, realized that they were seeing web applications being developed again and again with the same fundamental flaws.

Motivated by what they felt were systemic problems in modern application development, they drafted *The Twelve-Factor App*. This was a set of twelve rules and guidelines constituting a development methodology for building web applications, and by extension, cloud native applications (although "cloud native" wasn't a commonly used term at the time). The methodology was for building web applications that: [12]

- Use declarative formats for setup automation, to minimize time and cost for new developers joining the project
- Have a clean contract with the underlying operating system, offering maximum portability between execution environments
- Are suitable for deployment on modern cloud platforms, obviating the need for servers and systems administration

12 Wiggins, Adam. *The Twelve-Factor App*. 2011. *https://12factor.net*.

- Minimize divergence between development and production, enabling continuous deployment for maximum agility
- Can scale up without significant changes to tooling, architecture, or development practices

While not fully appreciated when it was first published in 2011, as the complexities of cloud native development have become more widely understood (and felt), *The Twelve Factor App* and the properties it advocates have started to be cited as the bare minimum for any service to be cloud native.

I. Codebase

> One codebase tracked in revision control, many deploys.
>
> —The Twelve-Factor App

For any given service, there should be exactly one codebase that's used to produce any number of immutable releases for multiple deployments to multiple environments. These environments typically include a production site, and one or more staging and development sites.

Having multiple services sharing the same code tends to lead to a blurring of the lines between modules, trending in time to something like a monolith, making it harder to make changes in one part of the service without affecting another part (or another service!) in unexpected ways. Instead, shared code should be refactored into libraries that can be individually versioned and included through a dependency manager.

Having a single service spread across multiple repositories, however, makes it nearly impossible to automatically apply the build and deploy phases of your service's life cycle.

II. Dependencies

> Explicitly declare and isolate (code) dependencies.
>
> —The Twelve-Factor App

For any given version of the codebase, `go build`, `go test`, and `go run` should be deterministic: they should have the same result, however they're run, and the product should always respond the same way to the same inputs.

But what if a dependency—an imported code package or installed system tool beyond the programmer's control—changes in such a way that it breaks the build, introduces a bug, or becomes incompatible with the service?

Most programming languages offer a packaging system for distributing support libraries, and Go is no different.[13] By using Go modules to declare all dependencies, completely and exactly, you can ensure that imported packages won't change out from under you and break your build in unexpected ways.

To extend this somewhat, services should generally try to avoid using the `os/exec` package's `Command` function to shell out to external tools like ImageMagick or `curl`.

Yes, your target tool might be available on all (or most) systems, but there's no way to *guarantee* that they both exist and are fully compatible with the service everywhere that it might run in the present or future. Ideally, if your service requires an external tool, that tool should be *vendored* into the service by including it in the service's repository.

III. Configuration

> Store configuration in the environment.
>
> —The Twelve-Factor App

Configuration—anything that's likely to vary between environments (staging, production, developer environments, etc)—should always be cleanly separated from the code. Under no circumstances should an application's configuration be baked into the code.

Configuration items may include, but certainly aren't limited to:

- URLs or other resource handles to a database or other upstream service dependencies—even if it's not likely to change any time soon.
- Secrets of *any* kind, such as passwords or credentials for external services.
- Per-environment values, such as the canonical hostname for the deploy.

A common means of extracting configuration from code is by *externalizing* them into some configuration file—often YAML[14]—which may or may not be checked into the repository alongside the code. This is certainly an improvement over configuration-in-code, but it's also less than ideal.

First, if your configuration file lives outside of the repository, it's all too easy to accidentally check it in. What's more, such files tend to proliferate, with different versions for different environments living in different places, making it hard to see and manage configurations with any consistency.

13 Although it was for too long!

14 The world's worst configuration language (except for all the other ones).

Alternatively, you *could* have different versions of your configurations for each environment in the repository, but this can be unwieldy and tends to lead to some awkward repository acrobatics.

There Are No Partially Compromised Secrets

It's worth emphasizing that while configuration values should never be in code, passwords or other sensitive secrets should *absolutely, never ever* be in code. It's all too easy for those secrets, in a moment of forgetfulness, to get shared with the whole world.

Once a secret is out, it's out. There are no partially compromised secrets.

Always treat your repository—and the code it contains—as if it can be made public at any time. Which, of course, it can.

Instead of configurations as code or even as external configurations, *The Twelve Factor App* recommends that configurations be stored as *environment variables*. Using environment variables in this way actually has a lot of advantages:

- They are standard and largely OS and language agnostic.
- That are easy to change between deploys without changing any code.
- They're very easy to inject into containers.

Go has several tools for doing this.

The first—and most basic—is the `os` package, which provides the `os.Getenv` function for this purpose:

```
name := os.Getenv("NAME")
place := os.Getenv("CITY")

fmt.Printf("%s lives in %s.\n", name, place)
```

For more sophisticated configuration options, there are several excellent packages available. Of these, `spf13/viper` seems to be particularly popular. A snippet of Viper in action might look like the following:

```
viper.BindEnv("id")             // Will be uppercased automatically
viper.SetDefault("id", "13")    // Default value is "13"

id1 := viper.GetInt("id")
fmt.Println(id1)                // 13

os.Setenv("ID", "50")           // Typically done outside of the app!
```

```
id2 := viper.GetInt("id")
fmt.Println(id2)                // 50
```

Additionally, Viper provides a number of features that the standard packages do not, such as default values, typed variables, and reading from command-line flags, variously formatted configuration files, and even remote configuration systems like etcd and Consul.

We'll dive more deeply into Viper and other configuration topics in Chapter 10.

IV. Backing Services

> Treat backing services as attached resources.
>
> —The Twelve-Factor App

A backing service is any downstream dependency that a service consumes across the network as part of its normal operation (see Chapter 1). A service should make no distinction between backing services of the same type. Whether it's an internal service that's managed within the same organization or a remote service managed by a third party should make no difference.

To the service, each distinct upstream service should be treated as just another resource, each addressable by a configurable URL or some other resource handle, as illustrated oi Figure 6-3. All resources should be treated as equally subject to the *Fallacies of Distributed Computing* (see Chapter 4 for a refresher, if necessary).

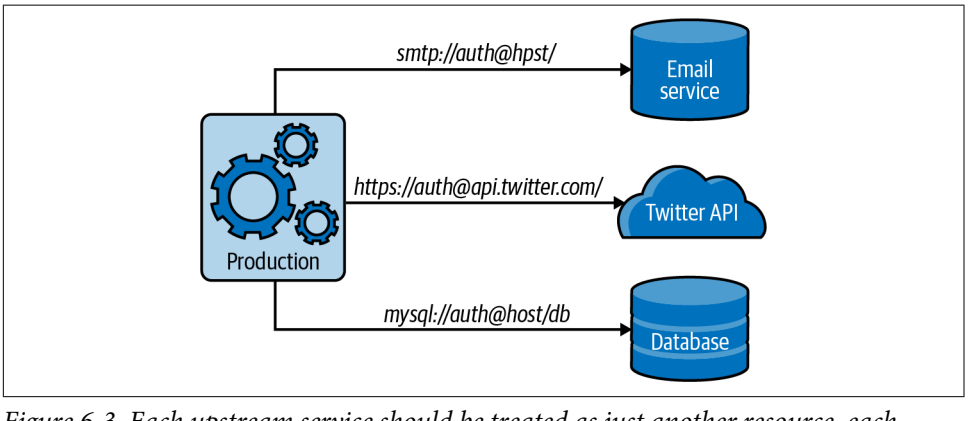

Figure 6-3. Each upstream service should be treated as just another resource, each addressable by a configurable URL or some other resource handle, each equally subject to the Fallacies of Distributed Computing

In other words, a MySQL database run by your own team's sysadmins should be treated no differently than an AWS-managed RDS instance. The same goes for *any*

upstream service, whether it's running in a data center in another hemisphere or in a Docker container on the same server.

A service that's able to swap out any resource at will with another one of the same kind—internally managed or otherwise—just by changing a configuration value can be more easily deployed to different environments, can be more easily tested, and more easily maintained.

V. Build, Release, Run

> Strictly separate build and run stages.
>
> —The Twelve-Factor App

Each (nondevelopment) deployment—the union of a specific version of the built code and a configuration—should be immutable and uniquely labeled. It should be possible, if necessary, to precisely recreate a deployment if (heaven forbid) it is necessary to roll a deployment back to an earlier version.

Typically, this is accomplished in three distinct stages, illustrated in Figure 6-4 and described in the following:

Build
: In the build stage, an automated process retrieves a specific version of the code, fetches dependencies, and compiles an executable artifact we call a *build*. Every build should always have a unique identifier, typically a timestamp or an incrementing build number.

Release
: In the release stage, a specific build is combined with a configuration specific to the target deployment. The resulting *release* is ready for immediate execution in the execution environment. Like builds, releases should also have a unique identifier. Importantly, producing releases with same version of a build shouldn't involve a rebuild of the code: to ensure environment parity, each environment-specific configuration should use the same build artifact.

Run
: In the run stage, the release is delivered to the deployment environment and executed by launching the service's processes.

Ideally, a new versioned build will be automatically produced whenever new code is deployed.

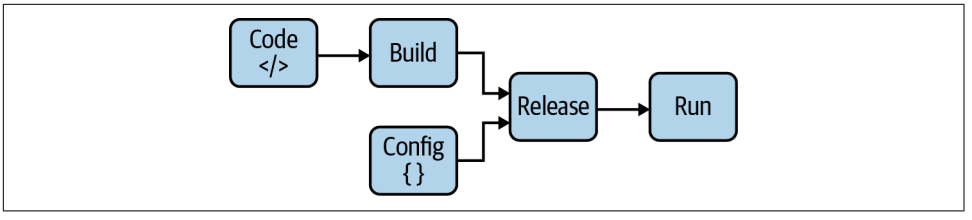

Figure 6-4. The process of deploying a codebase to a (nondevelopment) environment should be performed in distinct build, release, and run stages

VI. Processes

> Execute the app as one or more stateless processes.
>
> —The Twelve-Factor App

Service processes should be stateless and share nothing. Any data that has to be persisted should be stored in a stateful backing service, typically a database or external cache.

We've already spent some time talking about statelessness—and we'll spend more in the next chapter—so we won't dive into this point any further.

However, if you're interested in reading ahead, feel free to take a look at "State and Statelessness" on page 87.

VII. Data Isolation

> Each service manages its own data.
>
> —Cloud Native, Data Isolation

Each service should be entirely *self-contained*. That is, it should manage its own data, and make its data accessible only via an API designed for that purpose. If this sounds familiar to you, good! This is actually one of the core principles of microservices, which we'll discuss more in "The Microservices System Architecture" on page 103.

Very often this will be implemented as a request-response service like a RESTful API or RPC protocol that's exported by listening to requests coming in on a port, but this can also take the form of an asynchronous, event-based service using a publish-subscribe messaging pattern. Both of these patterns will be described in more detail in Chapter 8.

Historical Note

The actual title of the seventh section of *The Twelve Factor App* is "Port Binding," and is summarized as "export services via port binding."[15]

At the time, this advice certainly made sense, but this title obscures its main point: that a service should encapsulate and manage its own data, and only share that data via an API.

While many (or even most) web applications do, in fact, expose their APIs via ports, the increasing popularity of functions as a service (FaaS) and event-driven architectures means this is no longer necessarily always the case.

So, instead of the original text, I've decided to use the more up-to-date (and true-to-intent) summary provided by Boris Scholl, Trent Swanson, and Peter Jausovec in *Cloud Native: Using Containers, Functions, and Data to Build Next-Generation Applications* (O'Reilly).

And finally, although this is something you don't see in the Go world, some languages and frameworks allow the runtime injection of an application server into the execution environment to create a web-facing service. This practice limits testability and portability by breaking data isolation and environment agnosticism, and is *very strongly* discouraged.

VIII. Scalability

> Scale out via the process model.
>
> —The Twelve-Factor App

Services should be able to scale horizontally by adding more instances.

We talk about scalability quite a bit in this book. We even dedicated all of Chapter 7 to it. With good reason: the importance of scalability can't be understated.

Sure, it's certainly convenient to just beef up the one server your service is running on—and that's fine in the (very) short term—but vertical scaling is a losing strategy in the long run. If you're lucky, you'll eventually hit a point where you simply can't scale up any more. It's more likely that your single server will either suffer load spikes faster than you can scale up, or just die without warning and without a redundant failover.[16] Both scenarios end with a lot of unhappy users. We'll discuss scalability quite a bit more in Chapter 7.

15 Wiggins, Adam. "Port Binding." *The Twelve-Factor App*. 2011. *https://oreil.ly/bp8lC*.

16 Probably at three in the morning.

IX. Disposability

> Maximize robustness with fast startup and graceful shutdown.
>
> —The Twelve-Factor App

Cloud environments are fickle: provisioned servers have a funny way of disappearing at odd times. Services should account for this by being *disposable*: service instances should be able to be started or stopped—intentionally or not—at any time.

Services should strive to minimize the time it takes to start up to reduce the time it takes for the service to be deployed (or redeployed) to elastically scale. Go, having no virtual machine or other significant overhead, is especially good at this.

Containers provide fast startup time and are also very useful for this, but care must be taken to keep image sizes small to minimize the data transfer overhead incurred with each initial deploy of a new image. This is another area in which Go excels: its self-sufficient binaries can generally be installed into `SCRATCH` images, without requiring an external language runtime or other external dependencies. We demonstrated this in the previous chapter, in "Containerizing Your Key-Value Store" on page 44.

Services should also be capable of shutting down when they receive a `SIGTERM` signal by saving all data that needs to be saved, closing open network connections, or finishing any in-progress work that's left or by returning the current job to the work queue.

X. Development/Production Parity

> Keep development, staging, and production as similar as possible.
>
> —The Twelve-Factor App

Any possible differences between development and production should be kept as small as possible. This includes code differences, of course, but it extends well beyond that:

Code divergence
: Development branches should be small and short-lived, and should be tested and deployed into production as quickly as possible. This minimizes functional differences between environments and reduces the risk of both deploys and rollbacks.

Stack divergence
: Rather than having different components for development and production (say, SQLite on OS X versus MySQL on Linux), environments should remain as similar as possible. Lightweight containers are an excellent tool for this. This minimizes the possibility that inconvenient differences between almost-but-not-quite-the-same implementations will emerge to ruin your day.

Personnel divergence

Once it was common to have programmers who wrote code and operators who deployed code, but that arrangement created conflicting incentives and counterproductive adversarial relationships. Keeping code authors involved in deploying their work and responsible for its behavior in production helps break down development/operations silos and aligns incentives around stability and velocity.

Taken together, these approaches help to keep the gap between development and production small, which in turn encourages rapid, automated, continuous deployment.

XI. Logs

> Treat logs as event streams.
>
> —The Twelve-Factor App

Logs—a service's never-ending stream of consciousness—are incredibly useful things, particularly in a distributed environment. By providing visibility into the behavior of a running application, good logging can greatly simplify the task of locating and diagnosing misbehavior.

Traditionally, services wrote log events to a file on the local disk. At cloud scale, however, this just makes valuable information awkward to find, inconvenient to access, and impossible to aggregate. In dynamic, ephemeral environments like Kubernetes your service instances (and their log files) may not even exist by the time you get around to viewing them.

Instead, a cloud native service should treat log information as nothing more than a stream of events, writing each event, unbuffered, directly to `stdout`. It shouldn't concern itself with implementation trivialities like routing or storage of its log events, and allow the executor to decide what happens to them.

Though seemingly simple (and perhaps somewhat counterintuitive), this small change provides a great deal of freedom.

During local development, a programmer can watch the event stream in a terminal to observe the service's behavior. In deployment, the output stream can be captured by the execution environment and forwarded to one or more destinations, such as a log indexing system like Elasticsearch, Logstash, and Kibana (ELK) or Splunk for review and analysis, or a data warehouse for long-term storage.

We'll discuss logs and logging, in the context of observability, in more detail in Chapter 11.

XII. Administrative Processes

> Run administrative/management tasks as one-off processes.
>
> —The Twelve-Factor App

Of all of the original Twelve Factors, this is the one that most shows its age. For one thing, it explicitly advocates shelling into an environment to manually execute tasks.

To be clear: *making manual changes to a server instance creates snowflakes. This is a bad thing.* See "Special Snowflakes" on page 80.

Assuming you even have an environment that you can shell into, you should assume that it can (and eventually will) be destroyed and re-created any moment.

Ignoring all of that for a moment, let's distill the point to its original intent: administrative and management tasks should be run as one-off processes. This could be interpreted in two ways, each requiring its own approach:

- If your task is an administrative process, like a data repair job or database migration, it should be run as a short-lived process. Containers and functions are excellent vehicles for such purposes.
- If your change is an update to your service or execution environment, you should instead modify your service or environment construction/configuration scripts, respectively.

Special Snowflakes

Keeping servers healthy can be a challenge. At 3 a.m., when things aren't working quite right, it's really tempting to make a quick change and go back to bed. Congratulations, you've just created a *snowflake*: a special server instance with manual changes that give it unique, usually undocumented, behaviors.

Even minor, seemingly harmless changes can lead to significant problems. Even if the changes are documented—which is rarely the case—snowflake servers are hard to reproduce exactly, particularly if you need to keep an entire cluster in sync. This can lead to a bad time when you have to redeploy your service onto new hardware and can't figure out why it's not working.

Furthermore, because your testing environment no longer matches production, you can no longer trust your development environments to reliably reproduce your production deployment.

Instead, servers and containers should be treated as *immutable*. If something needs to be updated, fixed, or modified in any way, changes should be made by updating the

appropriate build scripts, baking[17] a new common image, and provisioning new server or container instances to replace the old ones.

As the expression goes, instances should be treated as "cattle, not pets."

Summary

In this chapter we considered the question "what's the point of cloud native?" The common answer is "a computer system that works in the cloud." But "work" can mean anything. Surely we can do better.

So we went back to thinkers like Tony Hoare and J-C Laprie, who provided the first part of the answer: *dependability*. That is, to paraphrase, computer systems that behave in ways that users find acceptable, despite living in a fundamentally unreliable environment.

Obviously, that's more easily said than done, so we reviewed three schools of thought regarding how to achieve it:

- Laprie's academic "means of dependability," which include preventing, tolerating, removing, and forecasting faults
- Adam Wiggins' *Twelve Factor App*, which took a more prescriptive (and slightly dated, in spots) approach
- Our own "cloud native attributes," based on the Cloud Native Computing Foundation's definition of "cloud native," that we introduced in Chapter 1 and organized this entire book around

Although this chapter was essentially a short survey of theory, there's a lot of important, foundational information here that describes the motivations and means used to achieve what we call "cloud native."

17 "Baking" is a term sometimes used to refer to the process of creating a new container or server image.

CHAPTER 7
Scalability

> Some of the best programming is done on paper, really. Putting it into the computer is just a minor detail.[1]
>
> —Max Kanat-Alexander, Code Simplicity: The Fundamentals of Software

In the summer of 2016, I joined a small company that digitized the kind of forms and miscellaneous paperwork that state and local governments are known and loved for. The state of their core application was pretty typical of early-stage startups, so we got to work and, by that fall, had managed to containerize it, describe its infrastructure in code, and fully automate its deployment.

One of our clients was a small coastal city in southeastern Virginia, so when Hurricane Matthew—the first Category 5 Atlantic hurricane in nearly a decade—was forecast to make landfall not far from there, the local officials dutifully declared a state of emergency and used our system to create the necessary paperwork for citizens to fill out. Then they posted it to social media, and half a million people all logged in at the same time.

When the pager went off, the on-call checked the metrics and found that aggregated CPU for the servers was pegged at 100%, and that hundreds of thousands of requests were timing out.

So, we added a zero to the desired server count, created a "to-do" task to implement autoscaling, and went back to our day. Within 24 hours, the rush had passed, so we scaled the servers in.

What did we learn from this, other than the benefits of autoscaling?[2]

1 Kanat-Alexander, Max. *Code Simplicity: The Science of Software Design*. O'Reilly Media, 23 March 2012.

2 Honestly, if we had autoscaling in place I probably wouldn't even remember that this happened.

First of all, it underscored the fact that without the ability to scale, our system would have certainly suffered extended downtime. But being able to add resources on demand meant that we could serve our users even under load far beyond what we had ever anticipated. As an added benefit, if any one server failed, its work could have been divided among the survivors.

Second, having far more resources than necessary isn't just wasteful, it's expensive. The ability to scale our instances back in when demand ebbed meant that we were only paying for the resources that we needed. A major plus for a startup on a budget.

Unfortunately, because unscalable services can seem to function perfectly well under initial conditions, scalability isn't always a consideration during service design. While this might be perfectly adequate in the short term, services that aren't capable of growing much beyond their original expectations also have a limited lifetime value. What's more, it's often fiendishly difficult to refactor a service for scalability, so building with it in mind can save both time and money in the long run.

First and foremost, this is meant to be a Go book, or at least more of a Go book than an infrastructure or architecture book. While we will discuss things like scalable architecture and messaging patterns, much of this chapter will focus on demonstrating how Go can be used to produce services that lean on the other (non-infrastructure) part of the scalability equation: efficiency.[3]

What Is Scalability?

You may recall that concept of scalability was first introduced way back in Chapter 1, where it was defined as the ability of a system to continue to provide correct service in the face of significant changes in demand. By this definition, a system can be considered to be scalable if it doesn't need to be redesigned to perform its intended function during steep increases in load.

Note that this definition[4] doesn't actually say anything at all about adding physical resources. Rather, it calls out a system's ability to handle large swings in demand. The thing being "scaled" here is the magnitude of the demand. While adding resources is one perfectly acceptable means of achieving scalability, it isn't exactly the same as being scalable. To make things just a little more confusing, the word "scaling" can also be applied to a system, in which case it *does* mean a change in the amount of dedicated resources.

3 If you want to know more about cloud native infrastructure and architecture, a bunch of excellent books on the subject have already been written. I particularly recommend *Cloud Native Infrastructure* by Justin Garrison and Kris Nova, and *Cloud Native Transformation* by Pini Reznik, Jamie Dobson, and Michelle Gienow (both O'Reilly Media).

4 This is my definition. I acknowledge that it diverges from other common definitions.

So how do we handle high demand without adding resources? As we'll discuss in "Scaling Postponed: Efficiency" on page 89, systems built with *efficiency* in mind are inherently more scalable by virtue of their ability to gracefully absorb high levels of demand, without immediately having to resort to adding hardware in response to every dramatic swing in demand, and without having to massively over-provision "just in case."

Different Forms of Scaling

Unfortunately, even the most efficient of efficiency strategies has its limit, and eventually you'll find yourself needing to scale your service to provide additional resources. There are two different ways that this can be done (see Figure 7-1), each with its own associated pros and cons:

Vertical scaling
: A system can be *vertically scaled* (or *scaled up*) by increasing its resource allocations. In a public cloud, an existing server can be vertically scaled fairly easily just by changing its instance size, but only until you run out of larger instance types (or money).

Horizontal scaling
: A system can be *horizontally scaled* (or *scaled out*) by duplicating the system or service to limit the burden on any individual server. Systems using this strategy can typically scale to handle greater amounts of load, but as you'll see in "State and Statelessness" on page 87, the presence of state can make this strategy difficult or impossible for some systems.

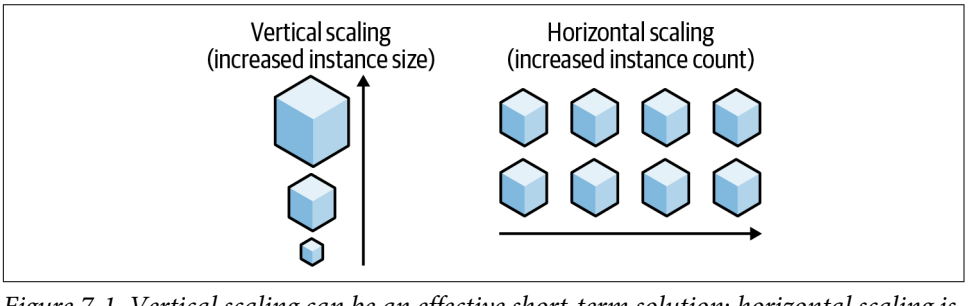

Figure 7-1. Vertical scaling can be an effective short-term solution; horizontal scaling is more technically challenging but may be a better long-term strategy

These two terms are used to describe the most common way of thinking about scaling: taking an entire system, and just making *more* of it. There are a variety of other scaling strategies in use, however.

Perhaps the most common of these is *functional partitioning*, which you're no doubt already familiar with, if not by name. Functional partitioning involves decomposing

complex systems into smaller functional units that can be independently optimized, managed, and scaled. You might recognize this as a generalization of a number of best practices ranging from basic program design to advanced distributed systems design.

Another approach common in systems with large amounts of data—particularly databases—is *sharding*. Systems that use this strategy distribute load by dividing their data into partitions called *shards*, each of which holds a specific subset of the larger dataset. A basic example of this is presented in "Minimizing locking with sharding" on page 97.

The Four Common Bottlenecks

As the demands on a system increase, there will inevitably come a point at which one resource just isn't able to keep pace, effectively stalling any further efforts to scale. That resource has become a *bottleneck*.

Returning the system to operable performance levels requires identifying and addressing the bottleneck. This can be done in the short-term by increasing the bottlenecked component—vertically scaling—by adding more memory or up-sizing the CPU, for instance. As you might recall from the discussion in "Different Forms of Scaling" on page 85, this approach isn't always possible (or cost-effective), and it can never be relied upon forever.

However, it's often possible to address a bottleneck by enhancing or reducing the burden on the affected component by utilizing another resource that the system still has in abundance. A database might avoid disk I/O bottlenecking by caching data in RAM; conversely a memory-hungry service could page data to disk. Horizontally scaling doesn't make a system immune: adding more instances can mean more communication overhead, which puts additional strain on the network. Even highly-concurrent systems can become victims of their own inner workings as the demand on them increases, and phenomena like lock contention come into play. Using resources effectively often means making tradeoffs.

Of course, fixing a bottleneck requires that you first identify the constrained component, and while there are many different resources that can emerge as targets for scaling efforts—whether by actually scaling the resource or by using it more efficiently—such efforts tend to focus on just four resources:

CPU
: The number of operations per unit of time that can be performed by a system's central processor and a common bottleneck for many systems. Scaling strategies for CPU include caching the results of expensive deterministic operations (at the expense of memory), or simply increasing the size or number of processors (at the expense of network I/O if scaling out).

Memory
: The amount of data that can be stored in main memory. While today's systems can store incredible amounts of data on the order of tens or hundreds of gigabytes, even this can fall short, particularly for data-intensive systems that lean on memory to circumvent disk I/O speed limits. Scaling strategies include offloading data from memory to disk (at the expense of disk I/O) or an external dedicated cache (at the expense of network I/O), or simply increasing the amount of available memory.

Disk I/O
: The speed at which data can be read from and written to a hard disk or other persistent storage medium. Disk I/O is a common bottleneck on highly parallel systems that read and write heavily to disk, such as databases. Scaling strategies include caching data in RAM (at the expense of memory) or using an external dedicated cache (at the expense of network I/O).

Network I/O
: The speed at which data can be sent across a network, either from a particular point or in aggregate. Network I/O translates directly into *how much* data the network can transmit per unit of time. Scaling strategies for network I/O are often limited,[5] but network I/O is particularly amenable to various optimization strategies that we'll discuss shortly.

As the demand on a system increases, it'll almost certainly find itself bottlenecked by one of these, and while there are efficiency strategies that can be applied, those tend to come at the expense of one or more other resources, so you'll eventually find your system being bottlenecked *again* by another resource.

State and Statelessness

We briefly touched on statelessness in "Application State Versus Resource State" on page 18, where we described application state—server-side data about the application or how it's being used by a client—as something to be avoided if at all possible. But this time, let's spend a little more time discussing what state is, why it can be problematic, and what we can do about it.

It turns out that "state" is strangely difficult to define, so I'll do my best on my own. For the purposes of this book I'll define state as the set of an application's variables which, if changed, affect the behavior of the application.[6]

5 Some cloud providers impose lower network I/O limits on smaller instances. Increasing the size of the instance may increase these limits in some cases.

6 If you have a better definition, let me know. I'm already thinking about the second edition.

Application State Versus Resource State

Most applications have some form of state, but not all state is created equal. It comes in two kinds, one of which is far less desirable than the other.

First, there's *application state*, which exists any time an application needs to remember an event locally. Whenever somebody talks about a *stateful* application, they're usually talking about an application that's designed to use this kind of local state. "Local" is an operative word here.

Second, there's *resource state*, which is the same for every client and which has nothing to do with the actions of clients, like data stored in external data store or managed by configuration management. It's misleading, but saying that an application is *stateless* doesn't mean that it doesn't have any data, just that it's been designed in such a way that it's free of any local persistent data. Its only state is resource state, often because all of its state is stored in some external data store.

To illustrate the difference between the two, imagine an application that tracks client sessions, associating them with some application context. If users' session data was maintained locally by the application, that would be considered "application state." But if the data was stored in an external database, then it could be treated as a remote resource, and it would be "resource state."

Application state is something of the "anti-scalability." Multiple instances of a stateful service will quickly find their individual states diverging due to different inputs being received by each replica. Server affinity provides a workaround to this specific condition by ensuring that each of a client's requests are made to the same server, but this strategy poses a considerable data risk, since the failure of any single server is likely to result in a loss of data.

Advantages of Statelessness

So far, we've discussed the differences between application state and resource state, and we've even suggested—without much evidence (yet)—that application state is bad. However, statelessness provides some very noticeable advantages:

Scalability
: The most visible and most often cited benefit is that stateless applications can handle each request or interaction independent of previous requests. This means that any service replica can handle any request, allowing applications to grow, shrink, or be restarted without losing data required to handle any in-flight sessions or requests. This is especially important when autoscaling your service, because the instances, nodes, or pods hosting the service can (and usually will) be created and destroyed unexpectedly.

Durability
: Data that lives in exactly one place (such as a single service replica) can (and, at some point, *will*) get lost when that replica goes away for any reason. Remember: everything in "the cloud" evaporates eventually.

Simplicity
: Without any application state, stateless services are freed from the need to… well… manage their state.[7] Not being burdened with having to maintain service-side state synchronization, consistency, and recovery logic[8] makes stateless APIs less complex, and therefore easier to design, build, and maintain.

Cacheability
: APIs provided by stateless services are relatively easy to design for cacheability. If a service knows that the result of a particular request will always be the same, regardless of who's making it or when, the result can be safely set aside for easy retrieval later, increasing efficiency and reducing response time.

These might seem like four different things, but there's overlap with respect to what they provide. Specifically, statelessness makes services both simpler and safer to build, deploy, and maintain.

Scaling Postponed: Efficiency

In the context of cloud computing, we usually think of scalability in terms of the ability of a system to add network and computing resources. Often neglected, however, is the role of *efficiency* in scalability. Specifically, the ability for a system to handle changes in demand *without* having to add (or greatly over-provision) dedicated resources.

While it can be argued that most people don't care about program efficiency most of the time, this starts to become less true as demand on a service increases. If a language has a relatively high per-process concurrency overhead—often the case with dynamically typed languages—it will consume all available memory or compute resources much more quickly than a lighter-weight language, and consequently require resources and more scaling events to support the same demand.

This was a major consideration in the design of Go's concurrency model, whose goroutines aren't threads at all but lightweight routines multiplexed onto multiple OS threads. Each costs little more than the allocation of stack space, allowing potentially millions of concurrently executing routines to be created.

7 I know I said the word "state" a bunch of times there. Writing is hard.

8 See also: idempotence.

As such, in this section we'll cover a selection of Go features and tooling that allow us to avoid common scaling problems, such as memory leaks and lock contention, and to identify and fix them when they do arise.

Efficient Caching Using an LRU Cache

Caching to memory is a very flexible efficiency strategy that can be used to relieve pressure on anything from CPU to disk I/O or network I/O, or even just to reduce latency associated with remote or otherwise slow-running operations.

The concept of caching certainly *seems* straightforward. You have something you want to remember the value of—like the result of an expensive (but deterministic) calculation—and you put it in a map for later. Right?

Well, you could do that, but you'll soon start running into problems. What happens as the number of cores and goroutines increases? Since you didn't consider concurrency, you'll soon find your modifications stepping on one another, leading to some unpleasant results. Also, since we forgot to remove anything from our map, it'll continue growing indefinitely until it consumes all of our memory.

What we need is a cache that:

- Supports concurrent read, write, and delete operations
- Scales well as the number of cores and goroutines increase
- Won't grow without limit to consume all available memory

One common solution to this dilemma is an LRU (Least Recently Used) cache: a particularly lovely data structure that tracks how recently each of its keys have been "used" (read or written). When a value is added to the cache such that it exceeds a predefined capacity, the cache is able to "evict" (delete) its least recently used value.

A detailed discussion of how to implement an LRU cache is beyond the scope of this book, but I will say that it's quite clever. As illustrated on Figure 7-2, an LRU cache contains a doubly linked list (which actually contains the values), and a map that associates each key to a node in the linked list. Whenever a key is read or written, the appropriate node is moved to the bottom of the list, such that the least recently used node is always at the top.

There are a couple of Go LRU cache implementations available, though none in the core libraries (yet). Perhaps the most common can be found as part of the golang/groupcache library. However, I prefer HashiCorp's open source extension to `groupcache`, `hashicorp/golang-lru`, which is better documented and includes `sync.RWMutexes` for concurrency safety.

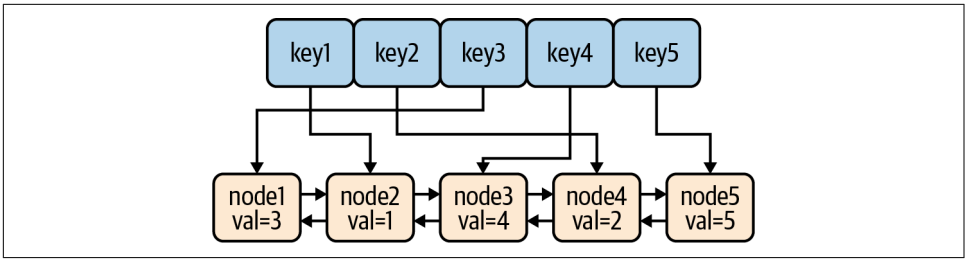

Figure 7-2. An LRU cache contains a map and a doubly linked list, which allows it to discard stale items when it exceeds its capacity

HashiCorp's library contains two construction functions, each of which returns a pointer of type `*Cache` and an `error`:

```
// New creates an LRU cache with the given capacity.
func New(size int) (*Cache, error)

// NewWithEvict creates an LRU cache with the given capacity, and also accepts
// an "eviction callback" function that's called when an eviction occurs.
func NewWithEvict(size int,
    onEvicted func(key interface{}, value interface{})) (*Cache, error)
```

The `*Cache` struct has a number of attached methods, the most useful of which are as follows:

```
// Add adds a value to the cache and returns true if an eviction occurred.
func (c *Cache) Add(key, value interface{}) (evicted bool)

// Check if a key is in the cache (without updating the recent-ness).
func (c *Cache) Contains(key interface{}) bool

// Get looks up a key's value and returns (value, true) if it exists.
// If the value doesn't exist, it returns (nil, false).
func (c *Cache) Get(key interface{}) (value interface{}, ok bool)

// Len returns the number of items in the cache.
func (c *Cache) Len() int

// Remove removes the provided key from the cache.
func (c *Cache) Remove(key interface{}) (present bool)
```

There are several other methods as well. Take a look at the GoDocs for a complete list.

In the following example, we create and use an LRU cache with a capacity of two. To better highlight evictions, we include a callback function that prints some output to `stdout` whenever an eviction occurs. Note that we've decided to initialize the `cache` variable in an `init` function, a special function that's automatically called before the `main` function and after the variable declarations have evaluated their initializers:

```
package main

import (
    "fmt"
    lru "github.com/hashicorp/golang-lru"
)

var cache *lru.Cache

func init() {
    cache, _ = lru.NewWithEvict(2,
        func(key interface{}, value interface{}) {
            fmt.Printf("Evicted: key=%v value=%v\n", key, value)
        },
    )
}

func main() {
    cache.Add(1, "a")          // adds 1
    cache.Add(2, "b")          // adds 2; cache is now at capacity

    fmt.Println(cache.Get(1))  // "a true"; 1 now most recently used

    cache.Add(3, "c")          // adds 3, evicts key 2

    fmt.Println(cache.Get(2))  // "<nil> false" (not found)
}
```

In the preceding program, we create a `cache` with a capacity of two, which means that the addition of a third value will force the eviction of the least recently used value.

After adding the values `{1:"a"}` and `{2:"b"}` to the cache, we call `cache.Get(1)`, which makes `{1:"a"}` more recently used than `{2:"b"}`. So, when we add `{3:"c"}` in the next step, `{2:"b"}` is evicted, so the next `cache.Get(2)` shouldn't return a value.

If we run this program we'll be able to see this in action. We'll expect the following output:

```
$ go run lru.go
a true
Evicted: key=2 value=b
<nil> false
```

The LRU cache is an excellent data structure to use as a global cache for most use cases, but it does have a limitation: at very high levels of concurrency—on the order of several million operations per second—it will start to experience some contention.

Unfortunately, at the time of this writing, Go still doesn't seem to have a *very* high throughput cache implementation.[9]

Efficient Synchronization

A commonly repeated Go proverb is "don't communicate by sharing memory; share memory by communicating." In other words, channels are generally preferred over shared data structures.

This is a pretty powerful concept. After all, Go's concurrency primitives—goroutines and channels—provide a powerful and expressive synchronization mechanism, such that a set of goroutines using channels to exchange references to data structures can often allow locks to be dispensed with altogether.

(If you're a bit fuzzy on the details of channels and goroutines, don't stress. Take a moment to flip back to Chapter 3. It's okay. I'll wait.)

That being said, Go *does* provide more traditional locking mechanisms by way of the `sync` package. But if channels are so great, why would we want to use something like a `sync.Mutex`, and when would we use it?

Well, as it turns out, channels *are* spectacularly useful, but they're not the solution to every problem. Channels shine when you're working with many discrete values, and are the better choice for passing ownership of data, distributing units of work, or communicating asynchronous results. Mutexes, on the other hand, are ideal for synchronizing access to caches or other large stateful structures.

At the end of the day, no tool solves every problem. Ultimately, the best option is to use whichever is most expressive and/or most simple.

Share memory by communicating

Threading is easy; locking is hard.

In this section we're going to use a classic example—originally presented in Andrew Gerrand's classic *Go Blog* article "Share Memory By Communicating"[10]—to demonstrate this truism and show how Go channels can make concurrency safer and easier to reason about.

9 However, if you're interested in learning more about high-performance caching in Go, take a look at Manish Rai Jain's excellent post on the subject, "The State of Caching in Go," on the *Dgraph Blog*.

10 Gerrand, Andrew. "Share Memory By Communicating." *The Go Blog*, 13 July 2010. *https://oreil.ly/GTURp* Portions of this section are modifications based on work created and shared by Google and used according to terms described in the Creative Commons 4.0 Attribution License.

Imagine, if you will, a hypothetical program that polls a list of URLs by sending it a GET request and waiting for the response. The catch is that each request can spend quite a bit of time waiting for the service to respond: anywhere from milliseconds to seconds (or more), depending on the service. Exactly the kind of operation that can benefit from a bit of concurrency, isn't it?

In a traditional threading environment that depends on locking for synchronization you might structure its data something like the following:

```
type Resource struct {
    url        string
    polling    bool
    lastPolled int64
}

type Resources struct {
    data []*Resource
    lock *sync.Mutex
}
```

As you can see, instead of having a slice of URL strings, we have two structs—`Resource` and `Resources`—each of which is already saddled with a number of synchronization structures beyond the URL strings we really care about.

To multithread the polling process in the traditional way, you might have a `Poller` function like the following running in multiple threads:

```
func Poller(res *Resources) {
    for {
        // Get the least recently polled Resource and mark it as being polled
        res.lock.Lock()

        var r *Resource

        for _, v := range res.data {
            if v.polling {
                continue
            }
            if r == nil || v.lastPolled < r.lastPolled {
                r = v
            }
        }

        if r != nil {
            r.polling = true
        }

        res.lock.Unlock()

        if r == nil {
            continue
```

```
            }

            // Poll the URL

            // Update the Resource's polling and lastPolled
            res.lock.Lock()
            r.polling = false
            r.lastPolled = time.Nanoseconds()
            res.lock.Unlock()
        }
    }
```

This does the job, but it has a lot of room for improvement. It's about a page long, hard to read, hard to reason about, and doesn't even include the URL polling logic or gracefully handle exhaustion of the `Resources` pool.

Now let's take a look at the same functionality implemented using Go channels. In this example, `Resource` has been reduced to its essential component (the URL string), and `Poller` is a function that receives `Resource` values from an input channel, and sends them to an output channel when they're done:

```
type Resource string

func Poller(in, out chan *Resource) {
    for r := range in {
        // Poll the URL

        // Send the processed Resource to out
        out <- r
    }
}
```

It's so…simple. We've completely shed the clockwork locking logic in `Poller`, and our `Resource` data structure no longer contains bookkeeping data. In fact, all that's left are the important parts.

But what if we wanted more than one `Poller` process? Isn't that what we were trying to do in the first place? The answer is, once again, gloriously simple: goroutines. Take a look at the following:

```
for i := 0; i < numPollers; i++ {
    go Poller(in, out)
}
```

By executing `numPollers` goroutines, we're creating `numPollers` concurrent processes, each reading from and writing to the same channels.

A lot has been omitted from the previous examples to highlight the relevant bits. For a walkthrough of a complete, idiomatic Go program that uses these ideas, see the "Share Memory By Communicating" Codewalk.

Reduce blocking with buffered channels

At some point in this chapter you've probably thought to yourself, "sure, channels are great, but writing to channels still blocks." After all, every send operation on a channel blocks until there's a corresponding receive, right? Well, as it turns out, this is only *mostly* true. At least, it's true of default, unbuffered channels.

However, as we first describe in Chapter 3, it's possible to create channels that have an internal message buffer. Send operations on such buffered channels only block when the buffer is full and receives from a channel only block when the buffer is empty.

You may recall that buffered channels can be created by passing an additional capacity parameter to the `make` function to specify the size of the buffer:

```
ch := make(chan type, capacity)
```

Buffered channels are especially useful for handling "bursty" loads. In fact, we already used this strategy in Chapter 5 when we initialized our `FileTransactionLogger`. Distilling some of the logic that's spread through that chapter produces something like the following:

```
type FileTransactionLogger struct {
    events       chan<- Event       // Write-only channel for sending events
    lastSequence uint64             // The last used event sequence number
}

func (l *FileTransactionLogger) WritePut(key, value string) {
    l.events <- Event{EventType: EventPut, Key: key, Value: value}
}

func (l *FileTransactionLogger) Run() {
    l.events = make(chan Event, 16)            // Make an events channel

    go func() {
        for e := range events {                // Retrieve the next Event
            l.lastSequence++                   // Increment sequence number
        }
    }()
}
```

In this segment, we have a `WritePut` function that can be called to send a message to an `events` channel, which is received in the `for` loop inside the goroutine created in the `Run` function. If `events` was a standard channel, each send would block until the anonymous goroutine completed a receive operation. That might be fine most of the time, but if several writes came in faster than the goroutine could process them, the upstream client would be blocked.

By using a buffered channel we made it possible for this code to handle small bursts of up to 16 closely clustered write requests. Importantly, however, the 17th write *would* block.

It's also important to consider that using buffered channels like this creates a risk of data loss should the program terminate before any consuming goroutines are able to clear the buffer.

Minimizing locking with sharding

As lovely as channels are, as we mentioned in "Efficient Synchronization" on page 93 they don't solve *every* problem. A common example of this is a large, central data structure, such as a cache, that can't be easily decomposed into discrete units of work.[11]

When shared data structures have to be concurrently accessed, it's standard to use a locking mechanism, such as the mutexes provided by the `sync` package, as we do in "Making Your Data Structure Concurrency-Safe" on page 15. For example, we might create a struct that contains a map and an embedded `sync.RWMutex`:

```
var cache = struct {
    sync.RWMutex
    data map[string]string
}{data: make(map[string]string)}
```

When a routine wants to write to the cache, it would carefully use `cache.Lock` to establish the write lock, and `cache.Unlock` to release the lock when it's done. We might even want to wrap it in a convenience function as follows:

```
func ThreadSafeWrite(key, value string) {
    cache.Lock()                                    // Establish write lock
    cache.data[key] = value
    cache.Unlock()                                  // Release write lock
}
```

By design, this restricts write access to whichever routine happens to have the lock. This pattern generally works just fine. However, as we discussed in Chapter 4, as the number of concurrent processes acting on the data increases, the average amount of time that processes spend waiting for locks to be released also increases. You may remember the name for this unfortunate condition: lock contention.

While this might be resolved in some cases by scaling the number of instances, this also increases complexity and latency, as distributed locks need to be established and writes need to establish consistency. An alternative strategy for reducing lock

11 You could probably shoehorn channels into a solution for interacting with a cache, but you might find it difficult to make it simpler than locking.

contention around shared data structures within an instance of a service is *vertical sharding*, in which a large data structure is partitioned into two or more structures, each representing a part of the whole. Using this strategy, only a portion of the overall structure needs to be locked at a time, decreasing overall lock contention.

You may recall that we discussed vertical sharding in some detail in Chapter 4. If you're unclear on vertical sharding theory or implementation, feel free to take some time to go back and review that section.

Memory Leaks Can…fatal error: runtime: out of memory

Memory leaks are a class of bugs in which memory is not released even after it's no longer needed. These bugs can be quite subtle and often plague languages like C++ in which memory is manually managed. But while garbage collection certainly helps by attempting to reclaim memory occupied by objects that are no longer in use by the program, garbage-collected languages like Go aren't immune to memory leaks. Data structures can still grow unbounded, unresolved goroutines can still accumulate, and even unstopped `time.Ticker` values can get away from you.

In this section we'll review a few common causes of memory leaks particular to Go, and how to resolve them.

Leaking goroutines

I'm not aware of any actual data on the subject,[12] but based purely on my own personal experience, I strongly suspect that goroutines are the single largest source of memory leaks in Go.

Whenever a goroutine is executed, it's initially allocated a small memory stack—2048 bytes—that can be dynamically adjusted up or down as it runs to suit the needs of the process. The precise maximum stack size depends on a lot of things,[13] but it's essentially reflective of the amount of available physical memory.

Normally, when a goroutine returns, its stack is either deallocated or set aside for recycling.[14] Whether by design or by accident, however, not every goroutine actually returns. For example:

```
func leaky() {
    ch := make(chan string)
```

12 If you are, let me know!

13 Dave Cheney wrote an excellent article on this topic called *Why is a Goroutine's stack infinite?* that I recommend you take a look at if you're interested in the dynamics of goroutine memory allocation.

14 There's a very good article by Vincent Blanchon on the subject of goroutine recycling entitled *How Does Go Recycle Goroutines?*

```
        go func() {
            s := <-ch
            fmt.Println("Message:", s)
        }()
    }
```

In the previous example, the `leaky` function creates a channel and executes a goroutine that reads from that channel. The `leaky` function returns without error, but if you look closely you'll see that no values are ever sent to `ch`, so the goroutine will never return and its stack will never be deallocated. There's even collateral damage: because the goroutine references `ch`, that value can't be cleaned up by the garbage collector.

So we now have a bona fide memory leak. If such a function is called regularly the total amount of memory consumed will slowly increase over time until it's completely exhausted.

This is a contrived example, but there are good reasons why a programmer might want to create long-running goroutines, so it's usually quite hard to know whether such a process was created intentionally.

So what do we do about this? Dave Cheney offers some excellent advice here: "You should never start a goroutine without knowing how it will stop….Every time you use the `go` keyword in your program to launch a goroutine, you must know how, and when, that goroutine will exit. If you don't know the answer, that's a potential memory leak."[15]

This may seem like obvious, even trivial advice, but it's incredibly important. It's all too easy to write functions that leak goroutines, and those leaks can be a pain to identify and find.

Forever ticking tickers

Very often you'll want to add some kind of time dimension to your Go code, to execute it at some point in the future or repeatedly at some interval, for example.

The `time` package provides two useful tools to add such a time dimension to Go code execution: `time.Timer`, which fires at some point in the future, and `time.Ticker`, which fires repeatedly at some specified interval.

However, where `time.Timer` has a finite useful life with a defined start and end, `time.Ticker` has no such limitation. A `time.Ticker` can live forever. Maybe you can see where this is going.

15 Cheney, Dave. "Never Start a Goroutine without Knowing How It Will Stop." dave.cheney.net, 22 Dec. 2016. *https://oreil.ly/VUlrY*.

Both Timers and Tickers use a similar mechanism: each provides a channel that's sent a value whenever it fires. The following example uses both:

```
func timely() {
    timer := time.NewTimer(5 * time.Second)
    ticker := time.NewTicker(1 * time.Second)

    done := make(chan bool)

    go func() {
        for {
            select {
            case <-ticker.C:
                fmt.Println("Tick!")
            case <-done:
                return
            }
        }
    }()

    <-timer.C
    fmt.Println("It's time!")
    close(done)
}
```

The `timely` function executes a goroutine that loops at regular intervals by listening for signals from `ticker`—which occur every second—or from a `done` channel that returns the goroutine. The line `<-timer.C` blocks until the 5-second timer fires, allowing `done` to be closed, triggering the `case <-done` condition and ending the loop.

The `timely` function completes as expected, and the goroutine has a defined return, so you could be forgiven for thinking that everything's fine. There's a particularly sneaky bug here though: running `time.Ticker` values contain an active goroutine that can't be cleaned up. Because we never stopped the timer, `timely` contains a memory leak.

The solution: always be sure to stop your timers. A `defer` works quite nicely for this purpose:

```
func timelyFixed() {
    timer := time.NewTimer(5 * time.Second)
    ticker := time.NewTicker(1 * time.Second)
    defer ticker.Stop()                          // Be sure to stop the ticker!

    done := make(chan bool)

    go func() {
        for {
            select {
            case <-ticker.C:
```

```
                fmt.Println("Tick!")
            case <-done:
                return
            }
        }
    }()

    <-timer.C
    fmt.Println("It's time!")
    close(done)
}
```

By calling `ticker.Stop()`, we shut down the underlying `Ticker`, allowing it to be recovered by the garbage collector and preventing a leak.

On Efficiency

In this section, we covered a number of common methods for improving the efficiency of your programs, ranging from using an LRU cache rather than a map to constrain your cache's memory footprint, to approaches for effectively synchronizing your processes, to preventing memory leaks. While these sections might not seem particularly closely connected, they're all important for building programs that scale.

Of course, there are countless other methods that I would have liked to include as well, but wasn't able to given the fundamental limits imposed by time and space.

In the next section, we'll change themes once again to cover some common service architectures and their effects on scalability. These might be a little less focused on Go specifically, but they're critical for a study of scalability, especially in a cloud native context.

Service Architectures

The concept of the *microservice* first appeared in the early 2010s as a refinement and simplification of the earlier service-oriented architecture (SOA) and a response to the *monoliths*—server-side applications contained within a single large executable—that were then the most common architectural model of choice.[16]

At the time, the idea of the microservice architecture—a single application composed of multiple small services, each running in its own process and communicating with lightweight mechanisms—was revolutionary. Unlike monoliths, which require the entire application to be rebuilt and deployed for any change to the system, microservices were independently deployable by fully automated deployment mechanisms. This sounds small, even trivial, but its implications were (and are) vast.

16 Not that they've gone away.

If you ask most programmers to compare monoliths to microservices, most of the answers you get will probably be something about how monoliths are slow, sluggish, and bloated, while microservices are small, agile, and the new hotness. Sweeping generalizations are always wrong, though, so let's take a moment to ask ourselves whether this is true, and whether monoliths might sometimes be the right choice.

We will begin by defining what we mean when we talk about monoliths and microservices.

The Monolith System Architecture

In a *monolith architecture*, all of the functionally distinguishable aspects of a service are coupled together in one place. A common example is a web application whose user interface, data layer, and business logic are all intermingled, often on a single server.

Traditionally, enterprise applications have been built in three main parts, as illustrated in Figure 7-3: a client-side interface running on the user's machine, a relational database where all of the application's data lives, and a server-side application that handles all user input, executes all business logic, and reads and writes data to the database.

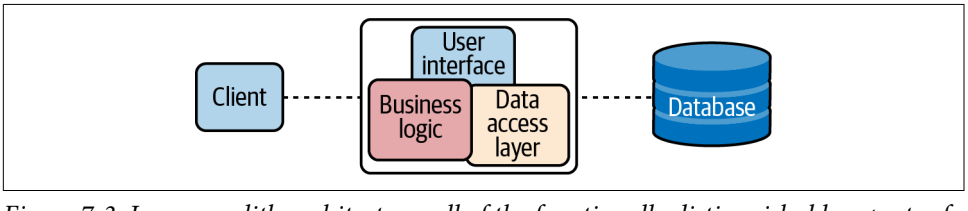

Figure 7-3. In a monolith architecture, all of the functionally distinguishable aspects of a service are coupled together in one place

At the time, this pattern made sense. All the business logic ran in a single process, making development easier, and you could even scale by running more monoliths behind a load balancer, usually using sticky sessions to maintain server affinity. Things were *perfectly fine*, and for many years this was by far the most common way of building web applications.

Even today, for relatively small or simple applications (for some definition of "small" and "simple") this works perfectly well (though I still strongly recommend statelessness over server affinity).

However, as the number of features and general complexity of a monolith increases, difficulties start to arise:

- Monoliths are usually deployed as a single artifact, so making even a small change generally requires a new version of the entire monolith to be built, tested, and deployed.
- Despite even the best of intentions and efforts, monolith code tends to decrease in modularity over time, making it harder to make changes in one part of the service without affecting another part in unexpected ways.
- Scaling the application means creating replicas of the entire application, not just the parts that need it.

The larger and more complex the monolith gets, the more pronounced these effects tend to become. By the early- to mid-2000s, these issues were well known, leading frustrated programmers to experiment with breaking their big, complex services into smaller, independently deployable and scalable components. By 2012, this pattern even had a name: microservices architecture.

The Microservices System Architecture

The defining characteristic of a *microservices architecture* is a service whose functional components have been divided into a set of discrete sub-services that can be independently built, tested, deployed, and scaled.

This is illustrated in Figure 7-4, in which a user interface service—perhaps an HTML-serving web application or a public API—interacts with clients, but rather than handling the business logic locally, it makes secondary requests of one or more component services to handle some specific functionality. Those services might in turn even make further requests of yet more services.

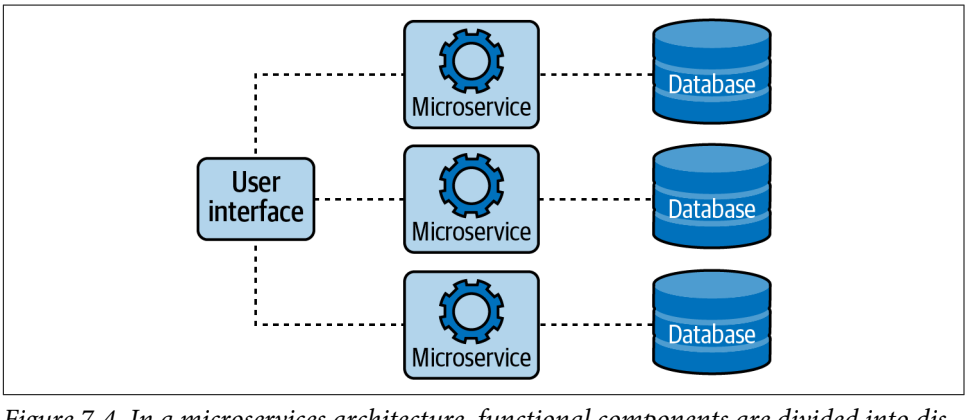

Figure 7-4. In a microservices architecture, functional components are divided into discrete subservices

While the microservices architecture has a number of advantages over the monolith, there are significant costs to consider. On one hand, microservices provide some significant benefits:

- A clearly-defined separation of concerns supports and reinforces modularity, which can be very useful for larger or multiple teams.
- Microservices should be independently deployable, making them easier to manage and making it possible to isolate errors and failures.
- In a microservices system, it's possible for different services to use the technology —language, development framework, data storage, etc—that's most appropriate to its function.

These benefits shouldn't be underestimated: the increased modularity and functional isolation of microservices tends to produce components that are themselves generally far more maintainable than a monolith with the same functionality. The resulting system isn't just easier to deploy and manage, but easier to understand, reason about, and extend for a larger number of programmers and teams.

Mixing different technologies may sound appealing in theory, but use restraint. Each adds new requirements for tooling and expertise. The pros and cons of adopting a new technology—any new technology[17]—should always be carefully considered.

The discrete nature of microservices makes them far easier to maintain, deploy, and scale than monoliths. However, while these are real benefits that can pay real dividends, there are some downsides as well:

- The distributed nature of microservices makes them subject to the Fallacies of Distributed Computing (see Chapter 4), which makes them significantly harder to program and debug.
- Sharing any kind of state between your services can often be extremely difficult.
- Deploying and managing multiple services can be quite complex and tends to demand a high level of operational maturity.

So given these, which do you choose? The relative simplicity of the monolith, or the flexibility and scalability of microservices? You might have noticed that most of the benefits of microservices pay off as the application gets larger or the number of teams

17 Yes, even Go.

working on it increases. For this reason many authors advocate starting with a monolith and decomposing it later.

On a personal note, I'll mention that I've never seen any organization successfully break apart a large monolith, but I've seen many try. That's not to say it's impossible, just that it's hard. I can't tell you whether you should start your system as microservices, or with a monolith and break it up later. I'd certainly get a lot of angry emails if I tried. But please, whatever you do, stay stateless.

Serverless Architectures

Serverless computing is a pretty popular topic in web application architecture, and a lot of (digital) ink has been spilled about it. Much of this hype has been driven by the major cloud providers, which have invested heavily in serverlessness, but not all of it.

But what is serverless computing, really?

Well, as is often the case, it depends on who you ask. For the purposes of this book, however, we're defining it as a form of utility computing in which some server-side logic, written by a programmer, is transparently executed in a managed ephemeral environment in response to some predefined trigger. This is also sometimes referred to as "functions as a service," or "FaaS." All of the major cloud providers offer FaaS implementations, such as AWS's Lambda or GCP's Cloud Functions.

Such functions are quite flexible and can be usefully incorporated into many architectures. In fact, as we'll discuss shortly, entire *serverless architectures* can even be built that don't use traditional services at all, but are instead built entirely from FaaS resources and third-party managed services.

Be Suspicious of Hype

I may sound like a grizzled old dinosaur here, but I've learned to be wary of new technologies that nobody really understands claiming to solve all of our problems.

According to the research and advisory firm Gartner, which specializes in studying IT and technology trends, serverless infrastructure is hovering at or near the "Peak of Inflated Expectations"[18] of its "hype cycle". This is eventually, but inevitably, followed by the "Trough of Disillusionment."

In time, people start to figure out what the technology is really useful for (not *everything*) and when to use it (not *always*), and it enters the "Slope of Enlightenment" and

18 Bowers, Daniel, et al. "Hype Cycle for Compute Infrastructure, 2019." *Gartner*, Gartner Research, 26 July 2019, *https://oreil.ly/3gkJh*.

"Plateau of Productivity." I've learned the hard way that it's usually best to wait until a technology has entered these two later phases before investing heavily in its use.

That being said: serverless computing *is* intriguing, and it *does* seem appropriate for some use-cases.

The pros and cons of serverlessness

As with any other architectural decision, the choice to go with a partially or entirely serverless architecture should be carefully weighed against all available options. While serverlessness provides some clear benefits—some obvious (no servers to manage!), others less so (cost and energy savings)—it's very different from traditional architectures, and carries its own set of downsides.

That being said, let's start weighing. Let's start with the advantages:

Operational management
: Perhaps the most obvious benefit of serverless architectures is that there's considerably less operational overhead.[19] There are no servers to provision and maintain, no licenses to buy, and no software to install.

Scalability
: When using serverless functions, it's the provider—not the user—who's responsible for scaling capacity to meet demand. As such, the implementor can spend less time and effort considering and implementing scaling rules.

Reduced costs
: FaaS providers typically use a "pay-as-you-go" model, charging only for the time and memory allocated when the function is run. This can be considerably more cost-effective than deploying traditional services to (likely underutilized) servers.

Productivity
: In a FaaS model, the unit of work is an event-driven function. This model tends to encourage a "function first" mindset, resulting in code that's often simpler, more readable, and easier to test.

It's not all roses, though. There are very some real downsides to serverless architectures that need to be taken into consideration as well:

Startup latency
: When a function is first called, it has to be "spun up" by the cloud provider. This typically takes less than a second, but in some cases can add 10 or more seconds to the initial requests. This is known as the *cold start* delay. What's more, if the

19 It's *right in the name!*

function isn't called for several minutes—the exact time varies between providers—it's "spun down" by the provider so that it has to endure another cold start when it's called again. This usually isn't a problem if your function doesn't have enough idle time to get spun down, but can be a significant issue if your load is particularly "bursty."

Observability
: While most of the cloud vendors provide some basic monitoring for their FaaS offerings, it's usually quite rudimentary. While third-party providers have been working to fill the void, the quality and quantity of data available from your ephemeral functions is often less than desired.

Testing
: While unit testing tends to be pretty straightforward for serverless functions, integration testing is quite hard. It's often difficult or impossible to simulate the serverless environment, and mocks are approximations at best.

Cost
: Although the "pay-as-you-go" model can be considerably cheaper when demand is lower, there is a point at which this is no longer true. In fact, very high levels of load can grow to be quite expensive.

Clearly, there's quite a lot to consider—on both sides—and while there *is* a great deal of hype around serverless at the moment, to some degree I think it's merited. However, while serverlessness promises (and largely delivers) scalability and reduced costs, it does have quite a few gotchas, including, but not limited to, testing and debugging challenges. Not to mention the increased burden on operations around observability![20]

Finally, as we'll see in the next section, serverless architectures also require quite a lot more up-front planning than traditional architectures. While some people might call this a positive feature, it can add significant complexity.

Serverless services

As mentioned previously, functions as a service (FaaS) are flexible enough to serve as the foundation of entire serverless architectures that don't use traditional services at all, but are instead built entirely from FaaS resources and third-party managed services.

20 Sorry, there's no such thing as NoOps.

Let's take, as an example, the familiar three-tier system in which a client issues a request to a service, which in turn interacts with a database. A good example is the key-value store we started in Chapter 5, whose (admittedly primitive) monolithic architecture might look something like what's shown in Figure 7-5.

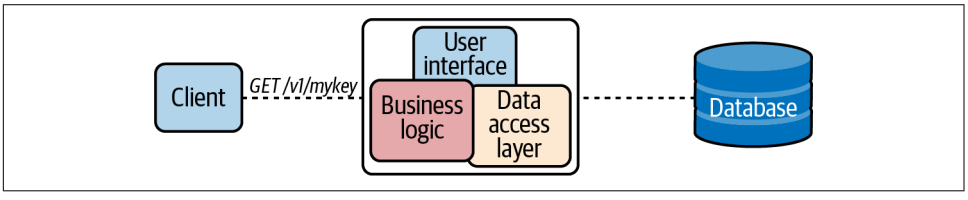

Figure 7-5. The monolithic architecture of our primitive key/value store

To convert this monolith into a serverless architecture, we'll need to use an *API gateway*: a managed service that's configured to expose specific HTTP endpoints and to direct requests to each endpoint to a specific resource—typically a FaaS functions—that handles requests and issue responses. Using this architecture, our key/value store might look something like what's shown in Figure 7-6.

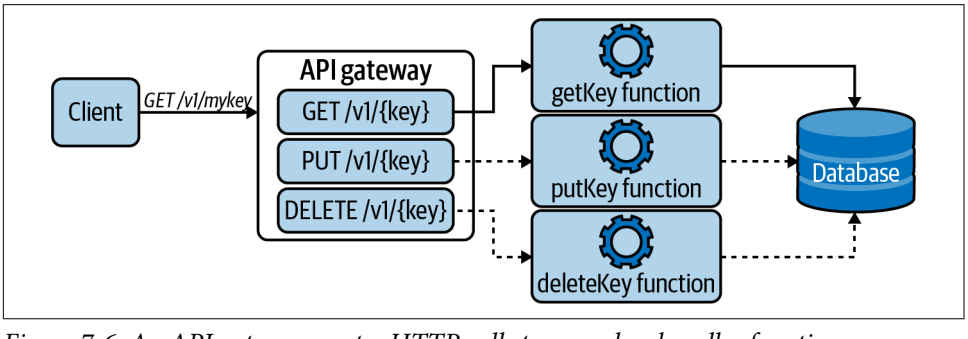

Figure 7-6. An API gateway routes HTTP calls to serverless handler functions

In this example, we've replaced the monolith with an API gateway that supports three endpoints: `GET /v1/{key}`, `PUT /v1/{key}`, and `DELETE /v1/{key}` (the `{key}` component indicates that this path will match any string, and refer to the value as `key`).

The API gateway is configured so that requests to each of its three endpoints are directed to a different handler function—`getKey`, `putKey`, and `deleteKey`, respectively—which performs all of the logic for handling that request and interacting with the backing database.

Granted, this is an incredibly simple application and doesn't account for things like authentication (which can be provided by a number of excellent third-party services like Auth0 or Okta), but some things are immediately evident.

First, there are a greater number of moving parts that you have to get your head around, which necessitates quite a bit more up-front planning and testing. For example, what happens if there's an error in a handler function? What happens to the request? Does it get forwarded to some other destination, or is it perhaps sent to a dead-letter queue for further processing?

Do not underestimate the significance of this increase in complexity! Replacing in-process interactions with distributed, fully managed components tends to introduce a variety of problems and failure cases that simply don't exist in the former. You may well have turned a relatively simple problem into an enormously complex one. Complexity kills; simplicity scales.

Second, with all of these different components, there's a need for more sophisticated distributed monitoring than you'd need with a monolith or small microservices system. Due to the fact that FaaS relies heavily on the cloud provider, this may be challenging or, at least, awkward.

Finally, the ephemeral nature of FaaS means that ALL state, even short-lived optimizations like caches, has to be externalized to a database, an external cache (like Redis), or network file/object store (like S3). Again, this can be argued to be a Good Thing, but it does add to up-front complexity.

Summary

This was a very difficult chapter to write, not because there isn't much to say, but because scalability is such a huge topic with so many different things I could have drilled down into. Every one of these battled in my brain for weeks.

I even ended up throwing away some perfectly good architecture content that, in retrospect, simply wasn't appropriate for this book. Fortunately, I was able to salvage a whole other chunk of work about messaging that ended up getting moved into Chapter 8. I think it's happier there anyway.

In those weeks, I spent a lot of time thinking about what scalability really is, and about the role that efficiency plays in it. Ultimately, I think that the decision to spend so much time on programmatic—rather than infrastructural—solutions to scaling problems was the right one.

All told, I think the end result is a good one. We certainly covered a lot of ground:

- We reviewed the different axes of scaling, and how scaling out is often the best long-term strategy.
- We discussed state and statelessness, and why application state is essentially "anti-scalability."

- We learned a few strategies for efficient in-memory caching and for avoiding memory leaks.
- We compared and contrasted monolithic, microservice, and serverless architectures.

That's quite a lot, and although I wish I'd been able to drill down in some more detail, I'm pleased to have been able to touch on the things I did.

About the Author

Matthew A. Titmus is a veteran of the software development industry. Since teaching himself to build virtual worlds in LPC, he's earned a surprisingly relevant degree in molecular biology, written tools to analyze terabyte-sized datasets at a high energy physics laboratory, developed an early web development framework from scratch, wielded distributed computing techniques to analyze cancer genomes, and pioneered machine learning techniques for linked data.

He was an early adopter and advocate of both cloud native technologies in general and the Go language in particular. For the past four years he has specialized in helping companies migrate monolithic applications into a containerized, cloud native world, allowing them to transform the way their services are developed, deployed, and managed. He is passionate about what it takes to make a system production quality, and has spent a lot of time thinking about and implementing strategies for observing and orchestrating distributed systems.

Matthew lives on Long Island with the world's most patient woman, to whom he is lucky to be married, and the world's most adorable boy, by whom he is lucky to be called "Dad."

Colophon

The animal on the cover of *Cloud Native Go* is a member of the tuco-tuco family (*Ctenomyidae*). These neotropical rodents can be found living in excavated burrows across the southern half of South America.

The name "tuco-tuco" refers to a wide range of species. In general, these rodents have heavily built bodies with powerful short legs and well-developed claws. They have large heads but small ears, and though they spend up to 90% of their time underground, their eyes are relatively large compared to other burrowing rodents. The color and texture of the tuco-tucos' fur varies depending on the species, but in general, their fur is fairly thick. Their tails are short and not particularly furry.

Tuco-tucos live in tunnel systems—which are often extensive and complicated—that they dig into sandy and/or loamy soil. These networks often include separate chambers for nesting and food storage. They have undergone a variety of morphological adaptions that help them create and thrive in these underground environments, including an improved sense of smell, which helps them orient themselves in the tunnels. They employ both scratch-digging and skull-tooth excavation when creating their burrows.

The diet of the tuco-tucos consists primarily of roots, stems, and grasses. Today, tuco-tucos are viewed as agricultural pests, but in pre-European South America they were an important foodsource for indigenous peoples, particlarly in Tierra del Fuego. Today, their conservation status is contingent upon species and geographic location. Many species fall into the "Least Concern" category, while others are considered "Endangered." Many of the animals on O'Reilly covers are endangered; all of them are important to the world.

The cover illustration is by Karen Montgomery, based on a black and white engraving from *English Cyclopedia Natural History*. The cover fonts are Gilroy Semibold and Guardian Sans. The text font is Adobe Minion Pro; the heading font is Adobe Myriad Condensed; and the code font is Dalton Maag's Ubuntu Mono.