

O'REILLY®

# Cloud Native Go

Building Reliable Services in  
Unreliable Environments



Early  
Release

RAW &  
UNEDITED

Matthew A. Titmus

# **Cloud Native Go**

**Building Reliable Services in Unreliable Environments**

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

**Matthew A. Titmus**

## **Cloud Native Go**

by Matthew A. Titmus

Copyright © 2021 O'Reilly Media. All rights reserved.

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](mailto:corporate@oreilly.com).

Acquisitions Editor: Suzanne McQuade

Development Editor: Amelia Blevins

Production Editor: Daniel Elfanbaum

Interior Designer: David Futato

Cover Designer: Karen Montgomery

Illustrator: Rebecca Demarest

January 2021: First Edition

## **Revision History for the Early Release**

- 2020-01-02: First Release
- 2020-01-27: Second Release
- 2020-03-02: Third Release
- 2020-04-08: Fourth Release

- 2020-05-28: Fifth Release
- 2020-10-28: Sixth Release
- 2020-12-21: Seventh Release
- 2021-01-26: Eighth Release

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

The O'Reilly logo is a registered trademark of O'Reilly Media, Inc. *Cloud Native Go*, the cover image, and related trade dress are trademarks of O'Reilly Media, Inc.

The views expressed in this work are those of the author, and do not represent the publisher's views. While the publisher and the author have used good faith efforts to ensure that the information and instructions contained in this work are accurate, the publisher and the author disclaim all responsibility for errors or omissions, including without limitation responsibility for damages resulting from the use of or reliance on this work. Use of the information and instructions contained in this work is at your own risk. If any code samples or other technology this work contains or describes is subject to open source licenses or the intellectual property rights of others, it is your responsibility to ensure that your use thereof complies with such licenses and/or rights.

978-1-492-07626-1

[LSI]

## **Dedication**

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.

# **Part I. Going Cloud Native**

---

# Chapter 1. What Is a “Cloud Native” Application?

---

## A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 1st chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at [ablevins@oreilly.com](mailto:ablevins@oreilly.com).

*The most dangerous phrase in the language is, “We’ve always done it this way.”<sup>1</sup>*

—Grace Hopper, Computerworld (January 1976)

If you’re reading this book, then you’ve no doubt at least heard the term “cloud native”. More likely you’ve probably seen some of the many, many articles, written by vendors bubbling over with breathless adoration and dollar signs in their eyes. If this is the bulk of your experience with the term so far then you can be forgiven for thinking the term to be ambiguous and buzzwordy, just another of a series of markety expressions that might have started as something useful but have since been taken over by people trying to sell you something. See also: Agile, DevOps.

For similar reasons a web search for “cloud native definition” might lead you to think that all an application needs to be cloud native is to be written in the “right” language<sup>2</sup> or framework, or to use the “right” technology. Certainly, your choice of language can make your life significantly easier or harder, but it’s neither necessary nor sufficient for making an application cloud native.

So is “cloud native” just a matter of *where* an application runs? The term “cloud native” certainly suggests that. All you’d need to do is pour your kludgy<sup>3</sup> old application into a container and run it in Kubernetes, and you’re cloud native now, right? Nope. All you’ve done is make your application harder to deploy and harder to manage<sup>4</sup>. A kludgy application in Kubernetes is still kludgy.

So what *is* a “cloud native” application? In this chapter, we’ll answer exactly that. First, we’ll examine the history of computing service paradigms up to (and especially) the present, and discuss how the relentless pressure to scale drove (and continues to drive) the development and adoption of technologies that provide high levels of dependability at often vast scales. Finally, we’ll identify the specific attributes associated with such an application.

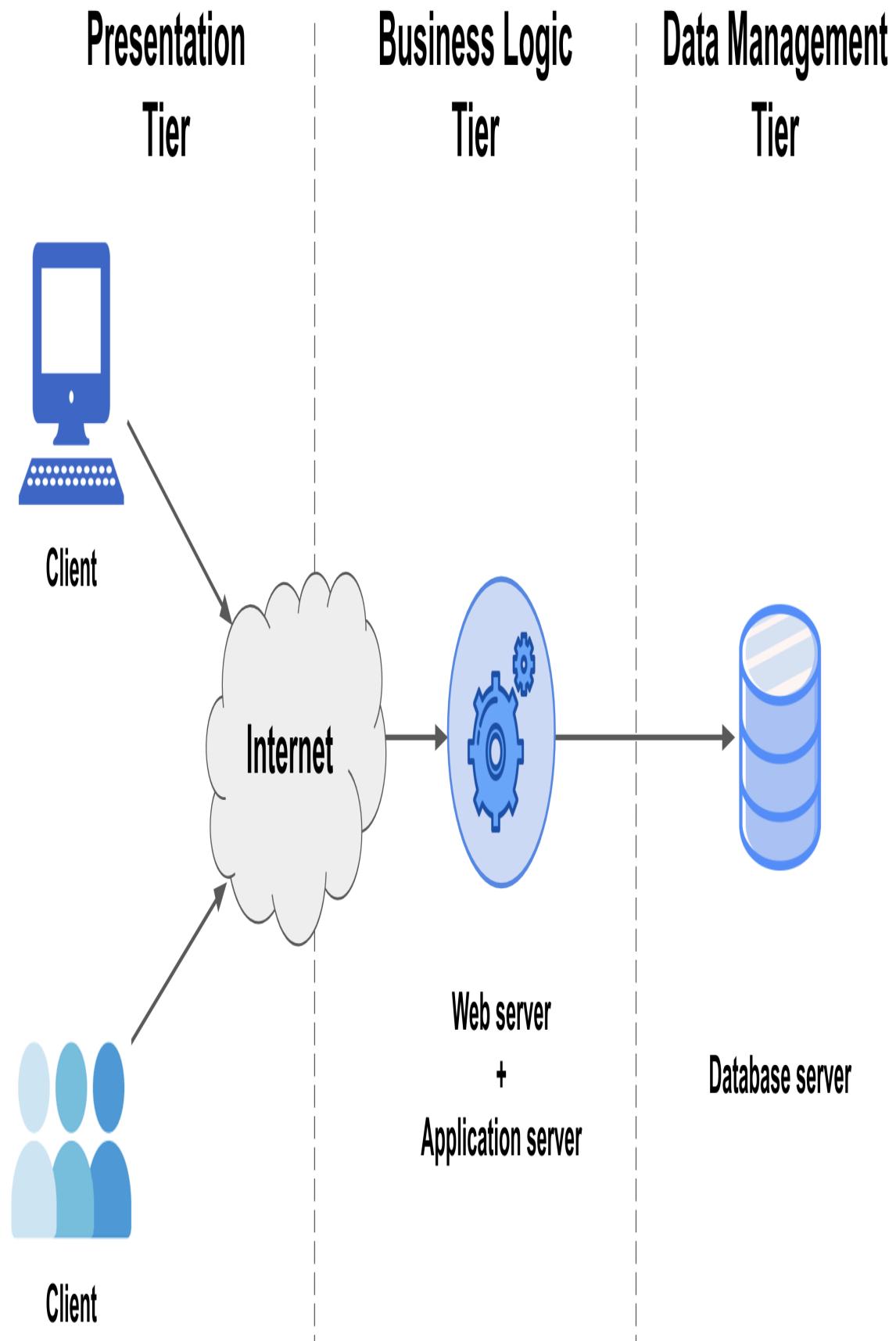
## The Story So Far

The story of networked applications is the story of the pressure to scale.

The late 1950s saw the introduction of the mainframe computer. At the time, every program and piece of data was stored in a single giant machine that users could access by means of dumb terminals with no computational ability of their own. All the logic and all the data all lived together as one big happy monolith. It was a simpler time.

Everything changed in the 1980s with the arrival of inexpensive network-connected PCs. Unlike dumb terminals, PCs were able to

do some computation of their own, making it possible to offload some of an application's logic onto them. This new "multi-tiered" architecture — which separated presentation logic, business logic, and data ([Figure 1-1](#)) — made it possible for the first time for the components of a networked application to be modified or replaced independent of the others.



*Figure 1-1. A traditional three-tiered architecture, with clearly-defined presentation, business logic, and data components.*

In the 1990s the popularization of the World Wide Web and the subsequent “dot-com” gold rush introduced the world to the software-as-a-service (SaaS). Entire industries were built on the SaaS model, driving the development of more complex and resource-hungry applications, which were in turn harder to develop, maintain, and deploy. Suddenly the classic multi-tiered architecture wasn’t enough anymore. In response, business logic started to get decomposed into sub-components that could be developed, maintained, and deployed independently, ushering in the age of microservices.

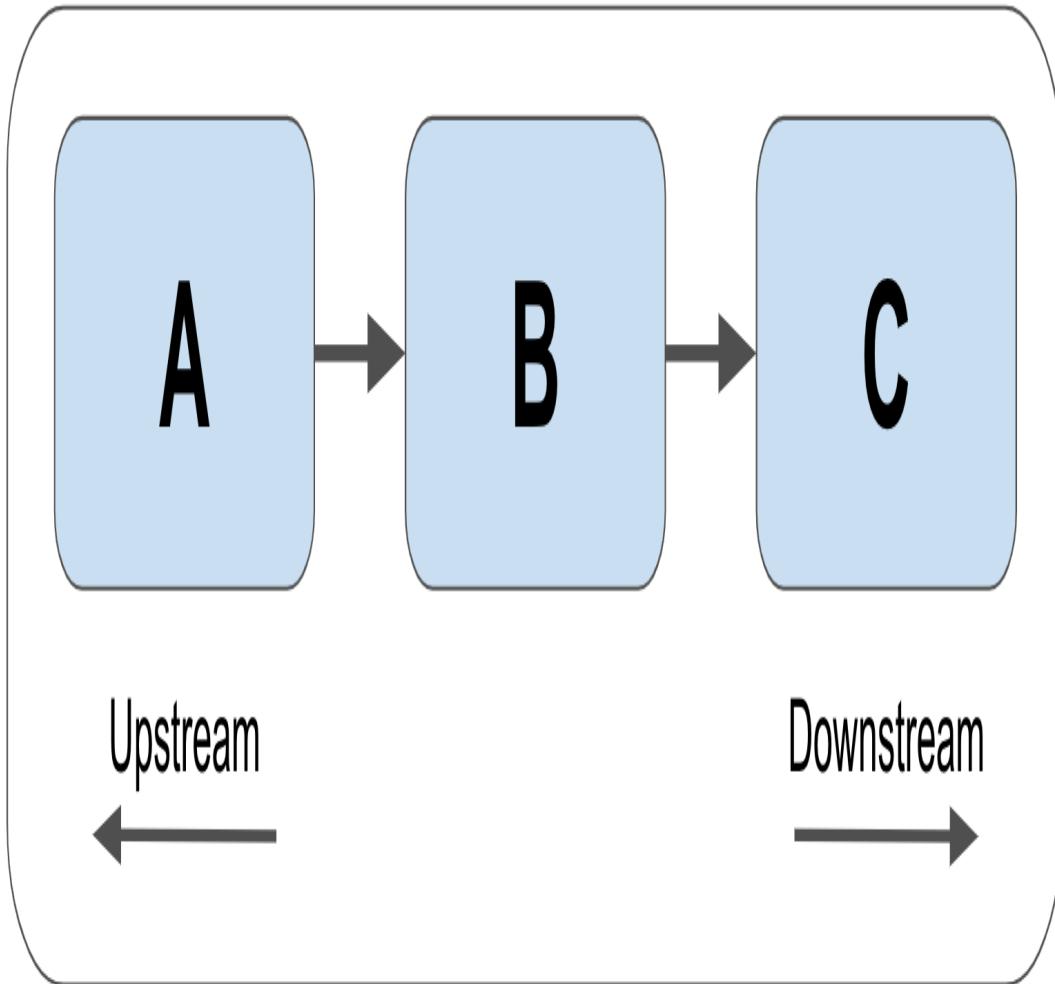
In 2006 Amazon launched Amazon Web Services (AWS), which included the Elastic Compute Cloud (EC2) service. Although AWS wasn’t the *first* infrastructure-as-a-service (IaaS) offering, it revolutionized the on-demand availability of data storage and computing resources, bringing Cloud Computing — and the ability to quickly scale — to the masses, catalyzing a massive migration of resources into “the cloud”.

Unfortunately, organizations soon learned that life at scale isn’t easy. Bad things happen, and when you’re working with hundreds or thousands (or more!) of resources, bad things happen *a lot*. Traffic will wildly spike up or down, essential hardware will fail, upstream dependencies will become suddenly and inexplicably inaccessible. Even if nothing goes wrong for a while, you still have to deploy and manage all of these resources. At this scale, it’s impossible (or at least wildly impractical) for humans to keep up with all of these issues manually.

## **UPSTREAM AND DOWNSTREAM DEPENDENCIES**

In this book we'll sometimes use the terms "upstream dependency" and "downstream dependency" to describe the relative positions of two resources in a dependency relationship. There's no real consensus in the industry around the directionality of these terms, so this book will use them as follows:

Imagine that we have three services: A, B, and C, as shown below.



← = “depends on”

In this scenario, Service A makes requests to (and therefore depends on) Service B, which in turn depends on Service C.

Because Service B depends on Service C, we can say that Service C is an *upstream dependency* of Service B. By extension, because Service A depends on Service B which

depends on Service C, Service C is also a *transitive upstream dependency* of Service A.

Inversely, because Service C is depended upon by Service B, we can say that Service B is a *downstream dependency* of Service C, and that Service A is a *transitive downstream dependency* of Service A.

## What Is Cloud Native?

Fundamentally, a truly cloud native application incorporates everything we've learned about running networked applications at scale over the past 60 years. They are scalable in the face of wildly-changing load, resilient in the face of environmental uncertainty, and manageable in the face of ever-changing requirements. In other words, a cloud native application is built for life in a cruel, uncertain universe.

But how do we *define* the term "cloud native"? Fortunately for all of us<sup>5</sup>, we don't have to. The Cloud Native Computing Foundation, a sub-foundation of the renowned Linux Foundation and something of an acknowledged authority on the subject, has already done it for us:

*Cloud native technologies empower organizations to build and run scalable applications in modern, dynamic environments such as public, private, and hybrid clouds....*

*These techniques enable loosely coupled systems that are resilient, manageable, and observable. Combined with robust automation, they allow engineers to make high-impact changes frequently and predictably with minimal toil<sup>6</sup>.*

—Cloud Native Computing Foundation, CNCF Cloud Native Definition v1.0

By this definition, cloud native applications are more than just applications that happen to live in a cloud. They're also scalable,

loosely coupled, resilient, manageable, and observable. Taken together, these “cloud native attributes” can be said to constitute the foundation of what it means for a system to be “cloud native”.

As it turns out, each of those words have pretty specific meaning of their own, so let’s break them down.

## Scalability

In the context of cloud computing, *scalability* can be defined as the ability of a system to continue to behave as expected in the face of significant upward or downward changes in demand. A system can be considered to be *scalable* if it doesn’t need to be refactored to perform its intended function during or after a steep increase in demand.

Because unscalable services can seem to function perfectly well under initial conditions, scalability isn’t always a primary consideration during service design. While this might be fine 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.

There are two different ways that a service can be scaled, each with its own associated pros and cons:

### *Vertical scaling*

A system can be *vertically scaled* (or *scaled up*) by up-sizing (or down-sizing) the hardware resources that are already allocated to it. For example, by adding memory or CPU to a database that’s running on a dedicated computing instance. Vertical scaling has the benefit of being technically relatively straight-forward, but any given instance can only be up-sized so much.

## *Horizontal scaling*

A system can be *horizontally scaled* (or *scaled out*) by adding (or removing) service instances. For example, by increasing the number of service nodes behind a load balancer or containers in Kubernetes or other container orchestration system. This strategy has a number of advantages, including redundancy and freedom from the limits of available instance sizes. However, more replicas means greater design and management complexity, and not all services can be horizontally scaled.

Given that there are two ways scaling a service — up or out — does that mean that any service whose hardware can be up-scaled (and is capable of taking advantage of increased hardware resources) is “scalable”? If you want to split hairs, then sure, to a point. But how scalable is it? Vertical scaling is inherently limited by the size of available computing resources, so in actuality a service that can only be scaled up isn’t very scalable at all. If you want to be able to scale by ten times, or a hundred, or a thousand, your service really has to be horizontally scalable.

So what’s the difference between a service that’s horizontally scalable and one that’s not? It all boils down to one thing: state. A service that doesn’t maintain any application state — or which has been very carefully designed to distribute its state between service replicas — will be relatively straight-forward to scale out. For any other application, it will be hard. It’s that simple.

The concepts of scalability, state, and redundancy will be discussed in much more depth in [Chapter 7](#).

## **Loose coupling**

*Loose coupling* is a system property and design strategy in which a system’s components have minimal knowledge of any other

components. Two systems can be said to be *loosely coupled* when changes to one component generally don't require changes to the other.

For example, web servers and web browsers can be considered to be loosely coupled: servers can be updated or even completely replaced without affecting our browsers at all. In their case, this is possible because standard web servers have agreed that they would communicate using a set of standard protocols<sup>7</sup>. In other words, they provide a *service contract*. Imagine the chaos if all the world's web browsers had to be updated each time Nginx or Httpd had a new version<sup>8</sup>!

It could be said that “loose coupling” is just a restatement of the whole point of microservice architectures: to partition components so that changes in one don't necessarily affect another. This might even be true. However, this principle is often neglected, and bears repeating. The benefits of loose coupling — and the consequences if it's neglected — cannot be understated. It's very easy to create a “worst of all worlds” system that pairs the management and complexity overhead of having multiple services with the dependencies and entanglements of a monolithic system: the dreaded *distributed monolith*.

Unfortunately, there's no magic technology or protocol that can keep your services from being tightly coupled. Any data exchange format can be misused. There are, however, several that help, and — when applied with practices like declarative APIs and good versioning practices — can be used to create services that are both loosely-coupled *and* modifiable.

These technologies and practices will be discussed and demonstrated in detail in [Chapter 8](#).

## Resilience

*Resilience* (roughly synonymous with *fault tolerance*) is a measure of how well a system withstands and recovers from errors and faults. A system can be considered *resilient* if it can continue operating correctly — possibly at a reduced level — rather than failing completely when some part of the system fails.

When we discuss resilience (and the other “cloud native attributes” as well, but especially when we discuss resilience) we use the word “system” quite a lot. A “system”, depending on how it’s used, can refer to anything from a complex web of interconnected services (such as an entire distributed application), to a collection of closely-related components (such as the replicas of a single function or service instance), or a single process running on a single machine. Every system is composed of several subsystems, which in turn are composed of sub-subsystems, which are themselves composed of sub-sub-subsystems. It’s turtles all the way down.

In the language of systems engineering, any system can contain defects, or *faults*, which we lovingly refer to as “bugs” in the software world. As we all know too well, under certain conditions, any fault can give rise to an *error*, which is the name we give to any discrepancy between a system’s intended behavior and its actual behavior. Errors have the potential to cause a system to fail to perform its required function: a *failure*. It doesn’t stop there though: a failure in a subsystem or component becomes a fault in the larger system; any fault that isn’t properly contained has the potential to cascade upwards until it causes a total system failure.

In an ideal world, every system would be carefully designed to prevent faults from ever occurring, but this is an unrealistic goal. You can’t prevent every possible fault, and it’s wasteful and unproductive to try. However, by assuming that all of system’s components are certain to fail — which they are — and designing them to respond to potential faults and limit the effects of failures, you can produce a system that’s functionally healthy even when some of its components are not.

There are many ways of designing a system for resiliency. Deploying redundant components is perhaps the most common approach, but that also assumes that a fault won't affect all components of the same type. Circuit breakers and retry logic can be included to prevent failures from propagating between components. Faulty components can even be reaped — or can intentionally fail — to benefit the larger system.

We'll discuss all of these approaches (and more) in much more depth in [Chapter 9](#).

## RESILIENCE IS NOT RELIABILITY

The terms *resilience* and *reliability* describe closely-related concepts, and are often confused. But, as we'll discuss in [Chapter 9](#), they aren't quite the same thing<sup>9</sup>.

- The *resilience* of a system is the degree to which it can continue to operate correctly in the face of errors and faults. Resilience, along with the other four cloud-native properties, is just one factor that contributes to reliability.
- The *reliability* of a system is its ability to behave as expected for a given time interval. Reliability, in conjunction with attributes like availability and maintainability, contributes to a system's overall dependability.

## Manageability

A system's *manageability* is the ease (or lack thereof) with which its behavior can be modified to keep it secure, running smoothly, and compliant with changing requirements. A system can be considered *manageable* if it's possible to sufficiently alter its behavior without having to alter its code.

As a system property, manageability gets a lot less attention than some of the more attention-grabbing attributes like scalability or observability. It's every bit as critical, though, particularly in complex, distributed systems.

For example, imagine a hypothetical system that includes a service and a database, and that the service refers to the database by a URL. What if you needed to update that service to refer to another database? If the URL was hard-coded you might have to update the code and redeploy, which depending on the system might be awkward for its own reasons. Of course, you could update the DNS record to point to the new location, but what if you needed to redeploy a development version of the service, with its own development database?

A manageable system might, for example, represent this value as an easily-modified environment variable; if the service that uses it is deployed in Kubernetes, adjustments to its behavior might be a matter of updating a value in a ConfigMap. A more complex system might even provide a declarative API that a developer can use to tell the system what behavior she expects. There's no single right answer<sup>10</sup>.

Manageability isn't limited to configuration changes. It encompasses all possible dimension of a system's behavior, be it the ability to activate feature flags, or rotate credentials or TLS certificates, or even (and perhaps especially) deploy or upgrade (or downgrade) system components.

Manageable systems are designed for adaptability, and can be readily adjusted to accommodate changing functional, environmental, or security requirements. Unmanageable systems, on the other hand, tend to be far more brittle, frequently requiring ad hoc — often manual — changes. The overhead involved in managing such systems places fundamental limits on their scalability, availability, and reliability.

The concept of manageability — and some preferred practices for implementing them in Go — will be discussed in much more depth in [Link to Come].

## MANAGEABILITY IS NOT MAINTAINABILITY

It can be said that manageability and maintainability have some “mission overlap” in that they’re both concerned with the ease with which a system can be modified<sup>11</sup>, but they’re actually quite different.

- *Manageability* describes the ease with which changes can be made to the behavior of a running system, up to and including deploying (and re-deploying) components of that system. It’s how easy it is to make changes *from the outside*.
- *Maintainability* describes the ease with which changes can be made to a system’s underlying functionality, most often its code. It’s how easy it is to make changes *from the inside*.

## Observability

The *observability* of a system is a measure of how well its internal states can be inferred from knowledge of its external outputs. A system can be considered *observable* when it’s possible to quickly and consistently ask novel questions about it with minimal prior knowledge and without having to re-instrument or build new code.

On its face, this might sound simple enough; just sprinkle in some logging and slap up a couple of dashboards, and your system is observable, right? Almost certainly not. Not with modern, complex systems at any rate, in which almost any problem is the manifestation of a web of multiple things going wrong

simultaneously. The Age of the LAMP Stack is over; things are harder now.

This isn't to say that metrics, logging, and tracing aren't important. On the contrary: they represent the building blocks of observability. But their mere existence is not enough: data is not information. They need to be used the right way. They need to be rich. Together, they need to be able to answer questions that you've never even thought to ask before.

The ability to detect and debug problems is a fundamental requirement for the maintenance and evolution of a robust system. But in a distributed system it's often hard enough just figuring out *where* a problem is. Complex systems are just too... *complex*. The number of possible failure states for any given system is proportional to the product of the number of possible partial and complete failure states of each of its components, and it's impossible to predict all of them. The traditional approach of focusing attention on the things we expect to fail simply isn't enough.

Emerging practices in observability can be seen as the evolution of monitoring. Years of experience with designing, building, and maintaining complex systems have taught us that traditional methods of instrumentation — including but not limited to dashboards, unstructured logs, or alerting on various "known unknowns" — just aren't up to the challenges presented by modern distributed systems.

Observability is a complex and subtle subject, but fundamentally it comes down to this: instrument your systems richly enough, and under real enough scenarios, that a future you can answer questions that you haven't thought to ask yet.

The concept of observability — and some suggestions for implementing it — will be discussed in much more depth in [Link to Come].

## Why Is Cloud Native a Thing?

The move towards “cloud native” is an example of architectural and technical adaptation, driven by environmental pressure and selection. It’s evolution; survival of the fittest. Bear with me here; I’m a biologist by training.

Eons ago, in the Dawn of Time<sup>12</sup>, applications would be built and deployed (generally by hand) to one or a small number of servers, where they were carefully maintained and nurtured. If they got sick, they were lovingly nursed back to health. If a service went down, you could often fix it with a restart. Observability was shelling into a server to run top and review logs. It was a simpler time.

In 1997 only 11% of people in industrialized countries and 2% worldwide were regular Internet users. The subsequent years saw exponential growth in Internet access and adoption, however, and by 2017 that number had exploded to 81% in industrialized countries and 48% worldwide<sup>13</sup> — and continues to grow.

All of those users — and their money — applied stress to services, generating significant incentive to scale. What’s more, as user sophistication and dependency on web services grew, so did expectations that their favorite web applications would be both feature-rich and always available.

The result was — and is — a significant evolutionary pressure towards scale, complexity, and dependability. These three attributes don’t play well together, though, and the traditional approaches simply couldn’t — and can’t — keep up. New techniques and practices had to be invented.

Fortunately, the introduction of public clouds and infrastructure-as-a-service (IaaS) made it relatively straight-forward to scale infrastructure out. Shortcomings with dependability could often be compensated for with sheer numbers. But that introduced new problems. How do you maintain a hundred servers? A thousand?

Ten thousand? How do you install your application onto them, or upgrade it? How do you debug it when it misbehaves? How do you even know it's healthy? Problems that are merely annoying at small scale tend to become very hard at large scale.

Cloud native is a thing because scale is the cause of and solution to all of our problems. It's not magic. It's not special. All fancy language aside, cloud native techniques and technologies exist for no other reasons than to make it possible to leverage the benefits of a "cloud" (quantity) while compensating for its downsides (lack of dependability).

## Summary

In this chapter, we talked a fair amount about the history of computing, and how what we now call "cloud native" isn't a new phenomenon so much as the inevitable outcome of a virtuous cycle of technological demand driving innovation driving more demand.

Ultimately, though, all of those fancy words distill down to a single point: today's applications have to dependably serve a lot of people. The techniques and technologies that we call "cloud native" represent the best current practices for building a service that's scalable, adaptable, and resilient enough to do that.

But what does all of this to do with Go? As it turns out, cloud native infrastructure requires cloud native tools. In [Chapter 2](#) we'll start to talk about what that means, exactly.

---

<sup>1</sup> Surden, Esther. "Privacy Laws May Usher In Defensive DP: Hopper." *Computerworld*, 26 Jan. 1976, p. 9.

<sup>2</sup> Which is Go. Don't get me wrong, this is still a Go book after all.

<sup>3</sup> A "kludge" is "an awkward or inelegant solution". It's a fascinating word with a fascinating history.

<sup>4</sup> Have you ever wondered why so many Kubernetes migrations fail?

- 5 Especially for me. I get to write this cool book.
- 6 Cloud Native Computing Foundation. “[CNCF Cloud Native Definition v1.0](#)”, GitHub, 7 Dec. 2020.
- 7 Those of us who remember the Browser Wars of the 1990’s will recall that this wasn’t always strictly true.
- 8 Or if every web site required a different browser. That would stink, *wouldn’t it?*
- 9 If you’re interested in a complete academic treatment, I highly recommend [\*Reliability and Availability Engineering\*](#) by Kishor S. Trivedi and Andrea Bobbio.
- 10 There are some wrong ones though.
- 11 Plus, they both start with *M*. Super confusing.
- 12 That time was the 1990s.
- 13 International Telecommunication Union (ITU). “Internet users per 100 inhabitants 1997 to 2007” and “Internet users per 100 inhabitants 2005 to 2017.” [\*ICT Data and Statistics \(IDS\)\*](#)

# Chapter 2. Why Go Rules the Cloud Native World

---

## A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 2nd chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at [ablevins@oreilly.com](mailto:ablevins@oreilly.com).

*Any intelligent fool can make things bigger, more complex, and more violent. It takes a touch of genius – and a lot of courage – to move in the opposite direction<sup>1</sup>.*

—E.F. Schumacher, *Small Is Beautiful* (August 1973)

## The Motivation Behind Go

The idea of Go emerged in September of 2007 at Google, the inevitable outcome of putting a bunch of smart people in a room and frustrating the heck out of them.

The people in question were Robert Griesemer, Rob Pike, and Ken Thompson; all already highly-regarded for their individual work in designing other languages. The source of their collective ire was

nothing less than the entire set of programming languages that were available at the time, which they were finding just weren't well-suited to the task of describing the kinds of distributed, scalable, resilient services that Google was building<sup>2</sup>.

Essentially, the common languages of the day had been developed in a different era, one before multiple processors were commonplace and networks were quite so ubiquitous. Their support for multicore processing and networking — essential building blocks of modern “cloud native” services<sup>3</sup> — was often limited or required extraordinary efforts to utilize. Simply put, programming languages weren't keeping up with the needs of modern software development.

## Features For a Cloud Native World

Their frustrations were many, but all of them amounted to one thing: the undue complexity of the languages they were working with was making it harder to build server software. These included, but weren't limited to<sup>4</sup>:

### *Low program comprehensibility*

Code had become too hard to read. Unnecessary bookkeeping and repetition was compounded by functionally overlapping features that often encouraged cleverness over clarity.

### *Slow builds*

Language construction and years of feature creep resulted in build times that ran for minutes or hours, even on large build clusters.

### *Inefficiency*

Many programmers responded to the above problems by adopting more fluid, dynamic languages, effectively trading efficiency and type safety for expressiveness.

### *High cost of updates*

Incompatibilities between even minor versions of a language, as well as any dependencies it may have (and its transitive dependencies!) often made updating an exercise in frustration.

Over the years, multiple — often quite clever — solutions have been presented to address some of these issues in various ways, usually introducing additional complexity in the process. Clearly, they couldn't be fixed with a new API or language feature. So, Go's designers envisioned a modern language, the first language built for the cloud native era, supporting modern networked and multicore computing, expressive yet comprehensible, and allowing its users to focus on solving their problems instead of struggling with their language.

The result, the Go language, is notable as much for the features it explicitly *doesn't have* as it is for the ones it does. Some of those features (and non-features) and the motivation behind them are discussed below.

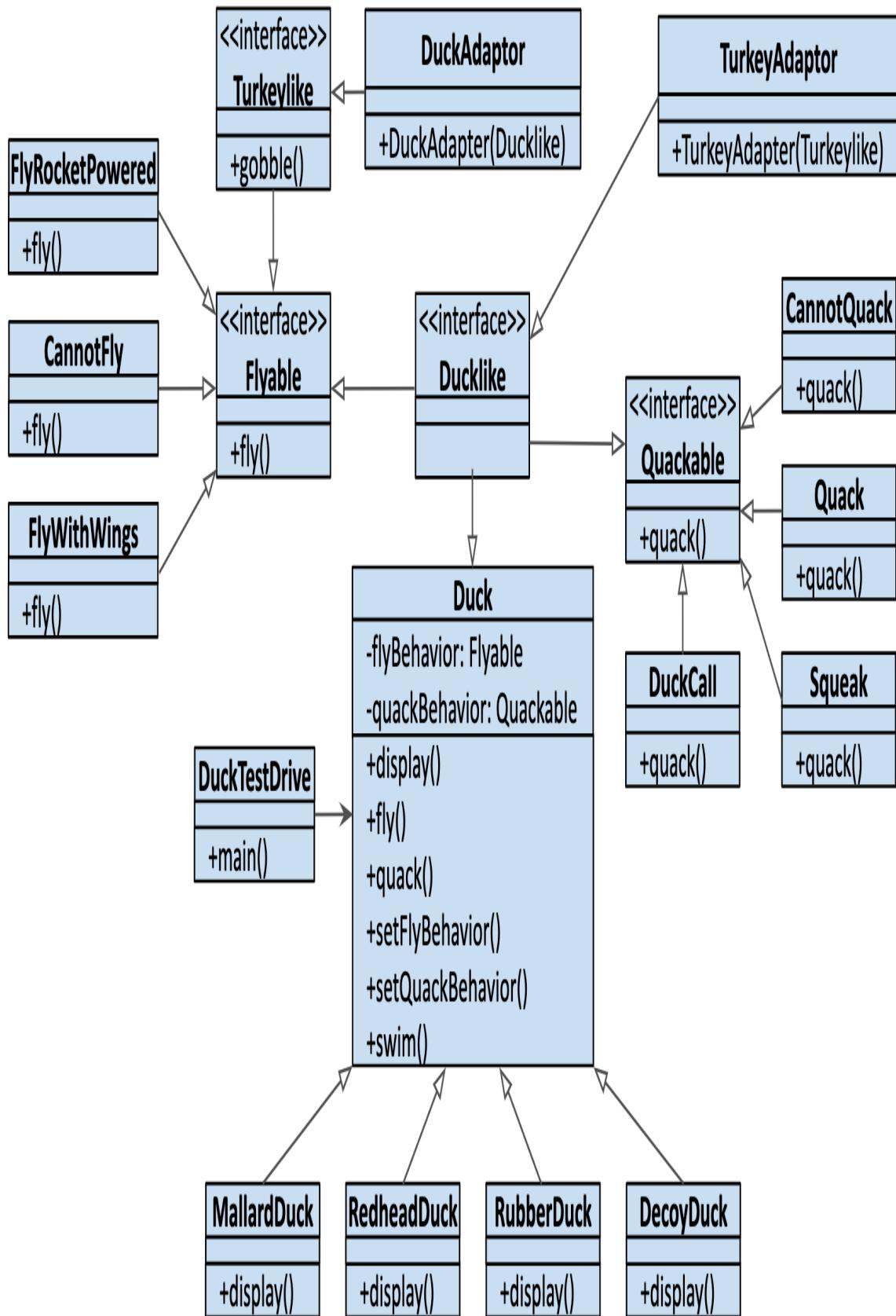
## **Composition and Structural Typing**

Object-oriented programming, which is based on the concept of “objects” of various “types” possessing various attributes, has existed since the 1960’s, but it truly came into vogue in the early to mid-1990’s with the release of Java and the addition of object-oriented features to C++. Since then it has emerged as the dominant programming paradigm, and remains so even today.

The promise of object-oriented programming is seductive, and the theory behind it even makes a certain kind of intuitive sense. Data and behaviors can be associated with *types* of things, which can be inherited by *sub-types* of those things. *Instances* of those types can be conceptualized as tangible objects with properties and behaviors,

components of a larger system modeling concrete, real-world concepts.

In practice however, object-oriented programming using inheritance often requires that relationships between types be carefully considered and painstakingly designed, and that particular design patterns and practices be faithfully observed. As such, as illustrated in [Figure 2-1](#), the tendency in object-oriented programming is for the focus to shift away from developing algorithms and towards developing and maintaining taxonomies and ontologies.



*Figure 2-1. Over time, object-oriented programming trends towards taxonomy.*

That's not to say that Go doesn't have object-oriented features that allow polymorphic behavior and code reuse. It, too, has a type-like concept in the form of structs, which can have properties and behaviors. What it rejects is inheritance and the elaborate relationships that come with it, opting instead to assemble more complex types by *embedding* simpler ones within them: an approach known as *composition*.

Specifically, where inheritance revolves around extending "is a" relationships between classes (i.e., a car "is a" motored vehicle), composition allows types to be constructed using "has a" relationships to define what they can do (i.e., a car "has a" motor). In practice, this permits greater design flexibility while allowing the creation of business domains that are less susceptible to disruption by the quirks of "family members".

By extension, while Go uses interfaces to describe behavioral contracts, it has no "is a" concept, so equivalency is determined by inspecting a type's definition, not its lineage. For example, given a Shape interface that defines an Area method, any type with an Area method will implicitly satisfy the Shape interface, without having to explicitly declare itself as a Shape:

```
type Shape interface {                                // Any Shape must have an Area
    Area() float64
}

type Rectangle struct {                           // Rectangle doesn't
    explicitly                                     // declare itself to be a
    width, height float64
    Shape
}

func (Rectangle r) Area() float64 {           // Rectangle has an Area
    method; it
    return r.width * r.height                  // satisfies the Shape
```

```
interface  
}
```

This *structural typing* mechanism, which has been described as duck typing<sup>5</sup> at compile time, largely sheds the burdensome maintenance of tedious taxonomies that saddle more traditional object-oriented languages like Java and C++, freeing programmers to focus on data structures and algorithms.

## Comprehensibility

Service languages like C++ and Java are often criticized for being clumsy, awkward to use, and unnecessarily verbose. They require lots of repetition and careful bookkeeping, saddling projects with superfluous boilerplate that gets in the way of programmers who have to divert their attention to things other than the problem they're trying to solve and limiting projects' scalability under the weight of all the resulting complexity.

Go was designed with large projects with lots of contributors in mind. Its minimalist design — just 25 keywords and 1 loop type — and the strong opinions of its compiler strongly favor clarity over cleverness<sup>6</sup>. This in turn encourages simplicity and productivity over clutter and complexity. The resulting code is relatively easy to ingest, review, and maintain, and harbors far fewer “gotchas”.

## CSP-Style Concurrency

Most mainstream languages provide some means of running multiple processes concurrently, allowing a program to be composed of independently-executed processes. Used correctly, concurrency can be incredibly useful, but it also introduces a number of challenges, particularly around ordering events, communication between processes, and coordination of access to shared resources.

Traditionally, a programmer will confront these challenges by allowing processes to share some piece of memory, which is then wrapped in locks or mutexes to restrict access to one process at a time. But even when well-implemented, this strategy can generate a fair amount of bookkeeping overhead. It's also easy to forget to lock or unlock shared memory, potentially introducing race conditions, deadlocks, or concurrent modifications. This class of errors can be fiendishly difficult to debug.

Go, on the other hand, favors another strategy, based on a formal language called CSP (“Communicating Sequential Processes”), first described in Tony Hoare’s influential paper of the same name<sup>7</sup> that describes patterns of interaction in concurrent systems in terms of message passing via channels.

The resulting concurrency model, implemented in Go with language primitives like goroutines and channels, makes Go uniquely<sup>8</sup> capable of elegantly structuring concurrent software without depending entirely on locking. It encourages developers to limit sharing memory, and to instead allow processes to interact with one another *entirely* by passing messages. This idea is often summarized by the Go proverb:

*Do not communicate by sharing memory. Instead, share memory by communicating.*

—Go Proverb

## CONCURRENCY IS NOT PARALLELISM

Computational concurrency and parallelism are often confused, which is understandable given that both concepts describe the state of having multiple processes executing during the same period of time. However, they are most definitely not the same thing<sup>9</sup>:

- *Parallelism* describes the simultaneous execution of multiple independent processes.
- *Concurrency* describes the composition of independently executing processes; it says nothing about when processes will execute.

## Fast Builds

One of the primary motivations for the Go language was the maddeningly long build times for certain languages of the time<sup>10</sup>, which even on Google's large compilation clusters often require minutes, or even hours, to complete. This eats away at development time and grinds down developer productivity. Given Go's primary purpose of enhancing rather than hindering developer productivity, long build times had to go.

The specifics of the Go compiler are beyond the scope of this book (and beyond my own expertise). Briefly, however, the Go language was designed to provide a model of software construction free of complex relationships, greatly simplifying dependency analysis and eliminating the need for C-style include files and libraries and the overhead that comes with them. As a result, most Go builds complete in seconds, or occasionally minutes, even on relatively humble hardware. For example, building all 1.8 million lines<sup>11</sup> of Go in Kubernetes v1.20.2 on a MacBook Pro with a 2.4 GHz 8-Core

Intel i9 processor and 32 GB of RAM required about 45 seconds of real time:

```
mtitus:~/workspace/kubernetes[MASTER]$ time make

real    0m45.309s
user    1m39.609s
sys     0m43.559s
```

Not that this doesn't come without compromises. Any proposed change to the Go language is weighed in part against its likely effect on build times; some otherwise promising proposals have been rejected on the grounds that they would increase it.

## Linguistic Stability

Go 1 was released in March of 2012, defining both the specification of the language and the specification of a set of core APIs. The natural consequence of this is an explicit promise, from the Go design team to the Go users, that programs written in Go 1 will continue to compile and run correctly, unchanged, for the lifetime of the Go 1 specification. That is, Go programs that work today can be expected to continue to work even under future “point” releases of Go 1 (Go 1.1, Go 1.2, etc.)<sup>12</sup>.

This stands in stark contrast to many other languages, which sometimes add new features enthusiastically, gradually increasing the complexity of the language — and anything written in it — in time leading to a once elegant language becoming a sprawling featurescape that's often exceedingly difficult to master<sup>13</sup>.

The Go Team considers this exceptional level of linguistic stability to be a vital feature of Go; it allows users to trust Go and to build on it. It allows libraries to be consumed and built upon with minimal hassle, and dramatically lowers the cost of updates, particularly for large projects and organizations. Importantly, it also allows the Go

community to use Go and to learn from it; to spend time writing with the language rather than writing the language.

This is not to say that Go won't grow: both the APIs and the core language certainly *can* acquire new packages and features<sup>14</sup>, and there are many proposals for exactly that<sup>15</sup>, but not in a way that breaks existing Go 1 code.

That being said, it's quite possible<sup>16</sup> that there will actually *never* be a Go 2. More likely, Go 1 will continue to be compatible indefinitely; and in the unlikely event that a breaking change is introduced, Go will provide a conversion utility, like the `go fix` command that was used during the move to Go 1.

## Memory Safety

The designers of Go have taken great pains to ensure that the language is free of the various bugs and security vulnerabilities — not to mention tedious bookkeeping — associated with direct memory access. Pointers are strictly typed and are always initialized to some value (even if that value is `nil`), and pointer arithmetic is explicitly disallowed. Built-in reference types like maps and channels, which are represented internally as pointers to mutable structures, are initialized by the `make` function. Simply put, Go neither needs nor allows the kind of manual memory management and manipulation that lower-level languages like C and C++ allow and require, and the subsequent gains with respect to complexity and memory safety can't be overstated.

For the programmer, the fact that Go is a garbage collected language obviates the need to carefully track and free up memory every allocated byte, eliminating a considerable bookkeeping burden from the programmer's shoulders. Life without `malloc` is liberating.

What's more, by eliminating manual memory management and manipulation — even pointer arithmetic — Go's designers have made it effectively immune to an entire class of memory errors and the

security holes they can introduce. No memory leaks, no buffer overruns, no address space layout randomization. Nothing.

Of course, this simplicity and ease-of-development comes with some tradeoffs, and while Go's garbage collector is incredibly sophisticated, it does introduce some overhead. As such, Go can't compete with languages like C++ and Rust in pure raw execution speed. That said, as we see in the next section, Go still does pretty well for itself in that arena.

## Performance

Confronted with the slow builds and tedious bookkeeping of the statically-typed, compiled languages like C++ and Java, many programmers moved towards more dynamic, fluid languages like Python. While these languages are excellent for many things, they're also very inefficient relative to compiled languages like Go, C++, and Java.

Some of this is made quite clear the benchmarks in [Table 2-1](#). Of course, benchmarks in general should be taken with a grain of salt, but some results are particularly striking.

*Table 2-1. Relative benchmarks for common service languages (seconds)<sup>a</sup>*

	C++	Go	Java	NodeJS	Python3	Ruby	Rust
Fannkuch-Redux	8.08	8.28	11.00	11.89	367.49	1255.50	7.28
FASTA	0.78	1.20	1.20	2.02	39.10	31.29	0.74
K-Nucleotide	1.95	8.29	5.00	15.48	46.37	72.19	2.76
Mandlebrot	0.84	3.75	4.11	4.03	172.58	259.25	0.93
N-Body	4.09	6.38	6.75	8.36	586.17	253.50	3.31
Spectral norm	0.72	1.43	4.09	1.84	118.40	113.92	0.71

<sup>a</sup> Gouy, Isaac. [The Computer Language Benchmarks Game](#). 18 Jan. 2021.

On inspection, it seems that the results can be clustered into three categories corresponding with the types of languages used to generate them:

- Compiled, strictly-typed languages with manual memory management (C++, Rust).
- Compiled, strictly-typed languages with garbage collection (Go, Java, NodeJS).
- Interpreted, dynamically-typed languages (Python, Ruby).

Interestingly, while these results suggest that while the garbage-collected languages are generally slightly less performant than the ones with manual memory management, the differences don't appear to be great enough to matter except under the most demanding requirements.

The differences between the interpreted and compiled languages, however, is striking. At least in these examples, Python, the archetypical dynamic language, benchmarks about *10 to 100 times*

*slower* than most compiled languages. Of course, it can be argued that this is still perfectly adequate for many — if not most — purposes, but this is less true for cloud native applications, which often have to endure significant spikes in demand, ideally without having to rely on potentially costly up-scaling.

## Static Linking

By default, Go programs are compiled directly into native, statically-linked executable binaries into which all necessary Go libraries and the Go runtime are copied. This produces slightly larger files (on the order of about 2MB for a “hello world”), but the resulting binary has no external language runtime to install<sup>17</sup> or external library dependencies to upgrade or conflict<sup>18</sup>, and can be easily distributed to users or deployed to a host without fear of suffering dependency or environmental conflicts.

This ability is particularly useful when you’re working with containers. Because Go binaries don’t require an external language runtime or even a distribution, they can be built into “scratch” images that don’t have parent images. The result is a very small (single digit MB) image with minimal deployment latency and data transfer overhead. These are very useful traits in an orchestration system like Kubernetes that may need to pull the image with some regularity.

## Static Typing

Back in the early days of Go’s design, its authors had to make a choice: would it be *statically typed*, like C++ or Java, requiring variables to be explicitly defined before use, or *dynamically typed*, like Python, allowing programmers to assign values to variables without defining them and therefore generally faster to code? It wasn’t a particularly hard decision; it didn’t take very long. Static typing was the obvious choice. But it wasn’t arbitrary, or based on personal preference<sup>19</sup>.

First, type correctness for statically-typed languages can be evaluated at compile time, making them far more performant (see [Table 2-1](#)).

The designers of Go understood that the time spent in development is only a fraction of a project's total lifecycle, and that any gains in coding velocity with dynamically typed languages is more than made up for by the increased difficulty in debugging and maintaining such code. After all, what Python programmer hasn't had their code crash because they tried to use a string as an integer?

Take the following Python code snippet, for example:

```
my_variable = 0

while my_variable < 10:
    my_varaible = my_variable + 1    # Typo! Infinite loop!
```

See it yet? Keep trying if you don't. It can take a second.

Any programmer can make this kind of subtle misspelling error, which just so happens to also produce perfectly valid, executable Python. These are just two trivial examples of an entire class of errors that Go will catch at compile time rather than (heavens forbid) in production, and generally closer in the code to the location where they are introduced. After all, it's well-understood that the earlier in the development cycle you catch a bug, the easier (read: cheaper) it is to fix it.

Finally, I'll even assert something somewhat controversial: typed languages are more readable. Python is often lauded as especially readable with its forgiving nature and somewhat English-like syntax<sup>20</sup>, but what would you do if presented with the following Python function signature?

```
def send(message, recipient):
```

Is message a string? Is recipient an instance of some class described elsewhere? Yes, this could be improved with some documentation and a couple of reasonable defaults, but many of us have had to maintain enough code to know that that's a pretty distant star to wish on. Explicitly defined types can guide development and ease the mental burden of writing code by automatically tracking information the programmer would otherwise have to track mentally by serving as documentation for both the programmer and everybody who has to maintain her code.

## Summary

If [Chapter 1](#) focused on what makes a *system* “cloud native”, then this chapter can be said to have focused on what makes a *language* — Go, specifically — a good fit for building “cloud native” services.

However, while a cloud native system needs to be scalable, loosely coupled, resilient, manageable, and observable, a language for the cloud native era has to be able to do more than just build systems with those attributes. After all, with a bit of effort, pretty much any language can, technically, be used to build such systems. So what makes Go so special?

It can be argued that all of the features presented in this chapter directly or indirectly contribute to the cloud native attributes from the previous chapter. Concurrency and memory safety can be said to contribute to service scalability, and structural typing to allow loose coupling, for example. But while Go is the only mainstream language I know that puts all of these features in one place, are they *really* so novel?

Perhaps most conspicuous of Go’s features are its baked-in — not bolted-on — concurrency features, which allow a programmer to fully and more safely utilize modern networking and multicore hardware. Goroutines and channels are wondrous, of course, and make it far easier to build resilient, highly-concurrent networked services, but

they're technically not unique if you consider some less common languages like Clojure or Crystal.

I would assert that where Go really shines is its faithful adherence to the principle of clarity over cleverness, which extends from an understanding that source code is written by humans for other humans<sup>21</sup>. That it compiles into machine code is almost immaterial.

Go is designed to support the way people actually work together: in teams, which sometimes change membership, whose members also work on other things. In this environment, code clarity, the minimization of “tribal knowledge”, and the ability to rapidly iterate are critical. Go’s simplicity is often misunderstood and unappreciated, but it lets programmers focus on solving problems instead of struggling with the language.

In [Chapter 3](#), we’ll review many of the specific features of the Go language, where we’ll get to see that simplicity up close.

---

<sup>1</sup> Schumacher, E.F. “Small Is Beautiful.” *The Radical Humanist*, August 1973, p. 22.

<sup>2</sup> These were “cloud native” services before the term “cloud native” was coined.

<sup>3</sup> Of course, they weren’t called “cloud native” at the time; to Google they were just “services”.

<sup>4</sup> Pike, Rob. [“Go at Google: Language Design in the Service of Software Engineering.”](#) *Go at Google: Language Design in the Service of Software Engineering*, Google, Inc., 2012.

<sup>5</sup> In languages that use “duck typing”, the type of an object is less important than the methods it defines. In other words, “if it walks like a duck and it quacks like a duck, then it must be a duck”

<sup>6</sup> Cheney, Dave. [“Clear Is Better than Clever.”](#) *The Acme of Foolishness*, 19 July 2019.

<sup>7</sup> Hoare, C.A.R. [“Communicating Sequential Processes.”](#) *Communications of the ACM*, vol. 21, no. 8, Aug. 1978, pp. 666–677.

<sup>8</sup> At least among the “mainstream” languages, whatever that means.

- 9 Gerrand, Andrew. “Concurrency is not parallelism.” *The Go Blog*, 16 Jan. 2016.
- 10 C++. We’re talking about C++.
- 11 Not counting comments; Openhub.net. “Kubernetes.” *Open Hub*, Black Duck Software, Inc., 18 Jan. 2021.
- 12 The Go Team. “Go 1 and the Future of Go Programs.” *The Go Documentation*.
- 13 Anybody remember Java 1.1? I remember Java 1.1. Sure, we didn’t have generics or autoboxing or enhanced for loops back then, but we were happy. Happy, I tell you.
- 14 I’m on team generics. Go, fightin’ Parametric Polymorphics!
- 15 The Go Team. “Proposing Changes to Go.” *GitHub*, 7 Aug. 2019.
- 16 Pike, Rob. “Sydney Golang Meetup - Rob Pike - Go 2 Draft Specifications” (video). *YouTube*, 13 Nov. 2018.
- 17 Take that, Java.
- 18 Take that, Python.
- 19 Few arguments in programming generate as many snarky comments as Static vs. Dynamic typing, except perhaps the Great Tabs vs Spaces Debate, on which Go’s unofficial position is “shut up, who cares?”
- 20 I, too, have been lauded for my forgiving nature and somewhat English-like syntax.
- 21 Or for the same human after a few months of thinking about other things.

# **Part II. Cloud Native Go Constructs**

---

# Chapter 3. Go Language Foundations

---

## A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 3rd chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at [ablevins@oreilly.com](mailto:ablevins@oreilly.com).

*A language that doesn't affect the way you think about programming is not worth knowing<sup>1</sup>.*

—Alan Perlis, ACM SIGPLAN Notices (September 1982)

No programming book would be complete without at least a brief refresher of its language of choice, so here we are!

This chapter will differ slightly from the ones in more introductory level books, however, in that we’re assuming that you’re at least familiar with common coding paradigms but may or may not be a little rusty with the finer points of Go syntax. As such, this chapter will focus less on Go’s fundamentals and more on its nuances and subtleties. For a deeper dive into the fundamentals, I recommend either *Introducing Go* by Caleb Doxsey (O’Reilly Media) or *The Go Programming Language* by Alan A. A. Donovan and Brian W. Kernighan (O’Reilly Media).

*Programming Language* by Alan A. A. Donovan and Brian W. Kernighan (Addison-Wesley Professional).

If you’re relatively new to the language, you’ll definitely want to read on. Even if you’re somewhat comfortable with Go, you might want to skim this chapter: there might be a gem or two in here for you. If you’re a seasoned veteran of the language, you can go ahead and move on to the next chapter (or read it ironically and judge me, you Go hipster you<sup>2</sup>).

## Types

### Simple Numbers

Go has a small menagerie of systematically-named, floating-point and signed and unsigned integer numbers:

#### *Signed Integer*

`int8, int16, int32, int64`

#### *Unsigned Integer*

`uint8, uint16, uint32, uint64`

#### *Floating-Point*

`float32, float64`

Systematic naming is nice, but code is written by humans with squishy human brains, so the Go designers provided two lovely conveniences.

First, there are two “machine dependent” types, simply called `int` and `uint`, whose size is determined based on available hardware. These are useful if the specific size of your numbers aren’t critical. Sadly, there is no machine-dependent floating-point number type.

Second, two integer types have mnemonic aliases: `byte`, which is an alias for `uint8`; and `rune`, which is an alias for `uint32`.

### TIP

For most uses, it generally makes sense to just use `int` and `float64`

## Complex Numbers

Go offers two sizes of complex numbers, if you're feeling a little imaginative<sup>3</sup>: `complex64` and `complex128`. These can be expressed as an *imaginary literal* by a floating point immediately followed by an `i`:

```
var x complex64 = 3.1415i
fmt.Println(x)                                // "(0+3.1415i)"
```

Complex numbers are very neat but don't come into play all that often, so we won't drill down into them here. If you're as fascinated by them as I hope you are, *The Go Programming Language* by Donovan and Kernighan gives them the full treatment they deserve.

## Booleans

The Boolean data type, representing the two logical truth values, exists in some form<sup>4</sup> in every programming language ever devised. It is represented by the `bool` type, a special 1-bit integer type that has two possible values:

- `true`
- `false`

Go supports all of the typical logical operations:

```
and := true && false
fmt.Println(and)          // false

or := true || false
fmt.Println(or)           // true

not := !true
fmt.Println(not)          // false
```

### NOTE

Curiously, Go doesn't include a logical XOR operator. There *is* a ^ operator, but it's reserved for bitwise XOR operations.

## Variables

Variables can be declared by using the var keyword to pair an identifier with some typed value, and may be updated at any time, with the general form:

```
var name type = expression
```

However, there is considerable flexibility in variable declaration:

- Declaration with initialization: var foo int = 42
- Declaration of multiple variables: var foo, bar int = 42, 1302
- Declaration with type inference: var foo = 42
- Declaration of mixed multiple types: var b, f, s = true, 2.3, "four"
- Declaration without initialization (the “zero” value; see “[Zero Values](#)”): var s string

## NOTE

Go is very opinionated about clutter: it *hates* it. If you declare a variable in a function but don't use it, your program will refuse to compile.

## Short Variable Declarations

Go provides a bit of syntactic sugar that allows variables within functions to be simultaneously declared and assigned by using the `:=` operator in place of a `var` declaration with an implicit type.

Short variable declarations have the general form:

```
name := expression
```

These can be used to declare both single and multiple assignments:

- With initialization: `percent := rand.Float64() * 100.0`
- Multiple variables at once: `x, y := 0, 2`

In practice, short variable declarations are the most common way that variables are declared and initialized in Go; `var` is usually only used either for local variables that need an explicit type or to declare a variable that will be assigned a value later.

## WARNING

Remember that `:=` is a declaration, and `=` is an assignment. A `:=` operator that only attempts to redeclare existing variables will fail at compile time.

Interestingly (and sometimes confusingly), if a short variable declaration has a mix of new and existing variables on its left-hand side, the short variable declaration acts like an assignment to the existing variables.

## Zero Values

When a variable is declared without an explicit value, it's assigned to the *zero value* for its type:

- Integers: 0
- Floats: 0.0
- Booleans: false
- Strings: "" (the empty string)

## Constants

Constants are very similar to variables, using the `const` keyword to associate an identifier with some typed value. However, constants differ from variables in some important ways. First, and most obviously, attempting to modify a constant will generate an error at compile time. Second, constants *must* be assigned a value at declaration: they have no zero value.

Both `var` and `const` may be used at both the package and function level, as follows:

```
const language string = "Go"

var favorite bool = true

func main() {
    const text string = "Does %s rule? %t!"
    var output string = fmt.Sprintf(text, language, favorite)

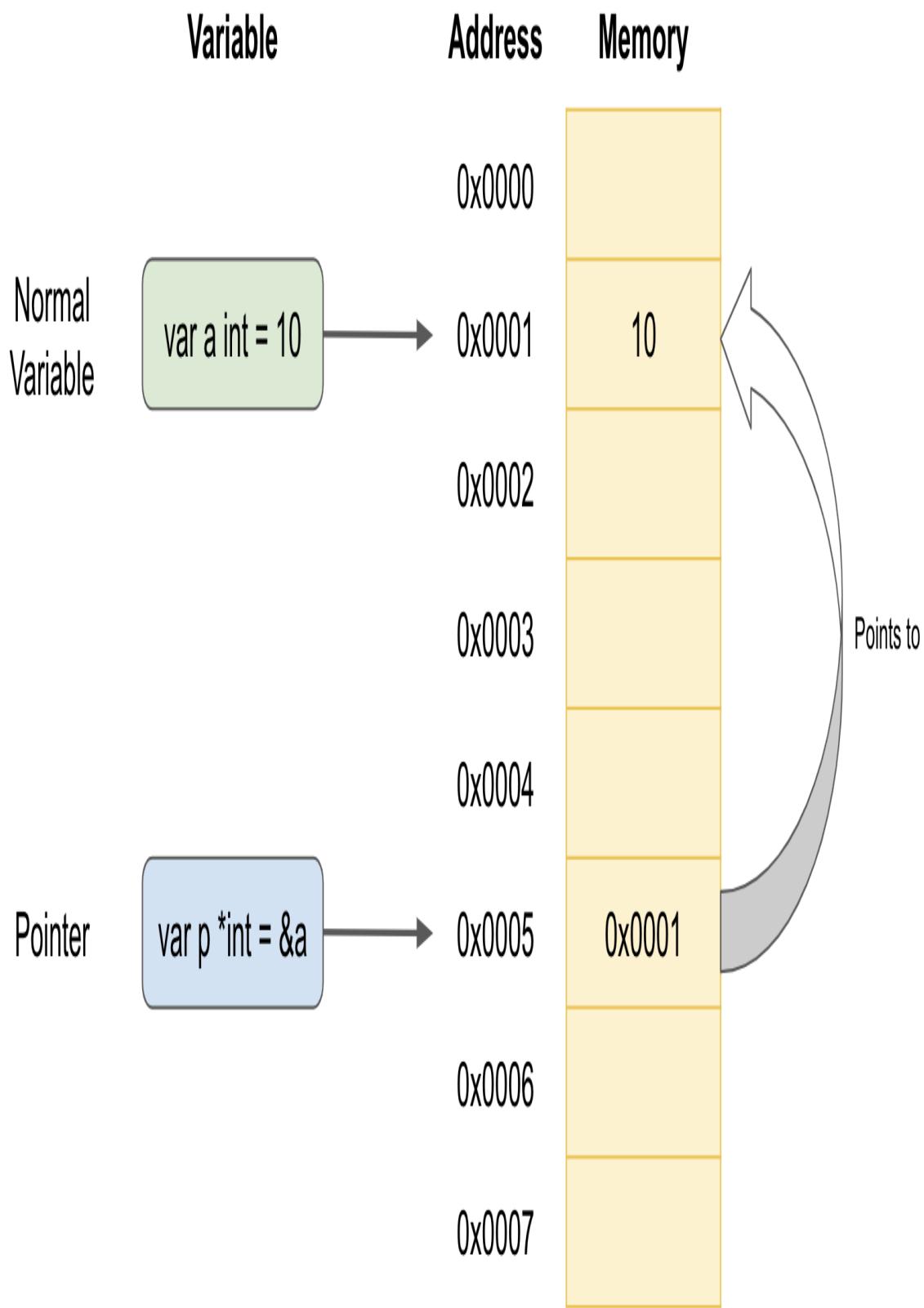
    fmt.Println(output) // "Does Go rule? true!"
}
```

## Pointers

Okay. Pointers. The bane and undoing of undergraduates the world over. If you're coming from a dynamically-typed language, the idea of the pointer may seem alien to you. While we're not going to drill down *too* deeply into the subject, we'll do our best to cover it well enough to provide some clarity on the subject.

Going back to first principles, a “variable” is a piece of storage in memory that contains some value. Typically, when you refer to a variable by its name (`foo = 10`) or by an expression (`s[i] = "foo"`), you’re directly reading or updating the value of the variable.

A *pointer* stores the *address* of a variable: the location in memory where the value is stored. Every variable has an address, and using pointers allows us to indirectly read or update the value of their variables.



*Figure 3-1. The expression p := &a gets the address of a and assigns it to p.*

### *Retrieving the address of a variable*

The address of a named variable can be retrieved by using the & operator. For example, the expression p := &a will obtain the address of a and assign it to p.

### *Pointer types*

The variable p, which we can say “points to” a, has a type of \*int, where the \* indicates that it’s a pointer type that points to an int.

### *Dereferencing a pointer*

To retrieve the value of the value a from p, we can *dereference* it using a \* before the pointer variable name, allowing us to indirectly read or update a.

Putting everything in one place:

```
var a int = 10

var p *int = &a           // p of type *int points to a
fmt.Println(p)             // "0x0001"
fmt.Println(*p)            // "10"

*p = 20                  // indirectly update a
fmt.Println(a)             // "20"
```

Pointers can be declared like any other variable, with a zero value of nil if not explicitly initialized. They are also comparable, being equal only if they contain the same address (that is, they point to the same variable) or if they are both nil:

```
var n *int
var x, y int

fmt.Println(n)           // "<nil>"
```

```

fmt.Println(n == nil)           // "true"
fmt.Println(x == y)             // "true"
fmt.Println(&x == &x)           // "true"
fmt.Println(&x == &y)           // "false"
fmt.Println(&x == nil)          // "false"

```

## Collections: Arrays, Slices, and Maps

### Arrays

In Go, as in most other mainstream languages, an *array* is a fixed-length sequence of zero or more elements of a particular type.

Arrays can be declared by including a length declaration. Individual elements are indexed from 0 to N-1, and can be accessed using the familiar bracket notation. The zero value of an array is an array of the specified length containing zero valued elements. The `len` builtin can be used to discover the length of an array.

```

var a [3]int = [3]int{2, 4, 6}    // Array of 3 integers
fmt.Println(a)                   // "[2 4 6]"
fmt.Println(len(a))              // "3"
fmt.Println(a[0])                // "2"
fmt.Println(a[len(a)-1])         // "6"

var b [5]int                     // Zero value array with 5 integers
fmt.Println(b)                   // "[0 0 0 0 0]"
fmt.Println(len(b))              // "5"

b[2] = 42
fmt.Println(b)                   // "[0 0 42 0 0]"

```

### Slices

Slices are a data type in Go that provide a powerful abstraction around a traditional array, such that working with slices looks and feels very much like working with arrays to the programmer. Like arrays, slices provide access to a sequence of elements of a

particular type via the familiar bracket notation, indexed from 0 to N-1. However, where arrays are fixed-length, slices can be resized at runtime.

In actuality, each slice is represented by a lightweight data structure with three components: a pointer to some element of a backing array that represents the first element of the slice, not necessarily the first element of the array; a length, representing the number of elements in the slice, and a capacity, which represents the upper value of the length. If not otherwise specified, the capacity value equals the number of elements between the start of the slice and the end of the backing array. The built-in `len` and `cap` functions will provide the length and capacity of a slice, respectively.

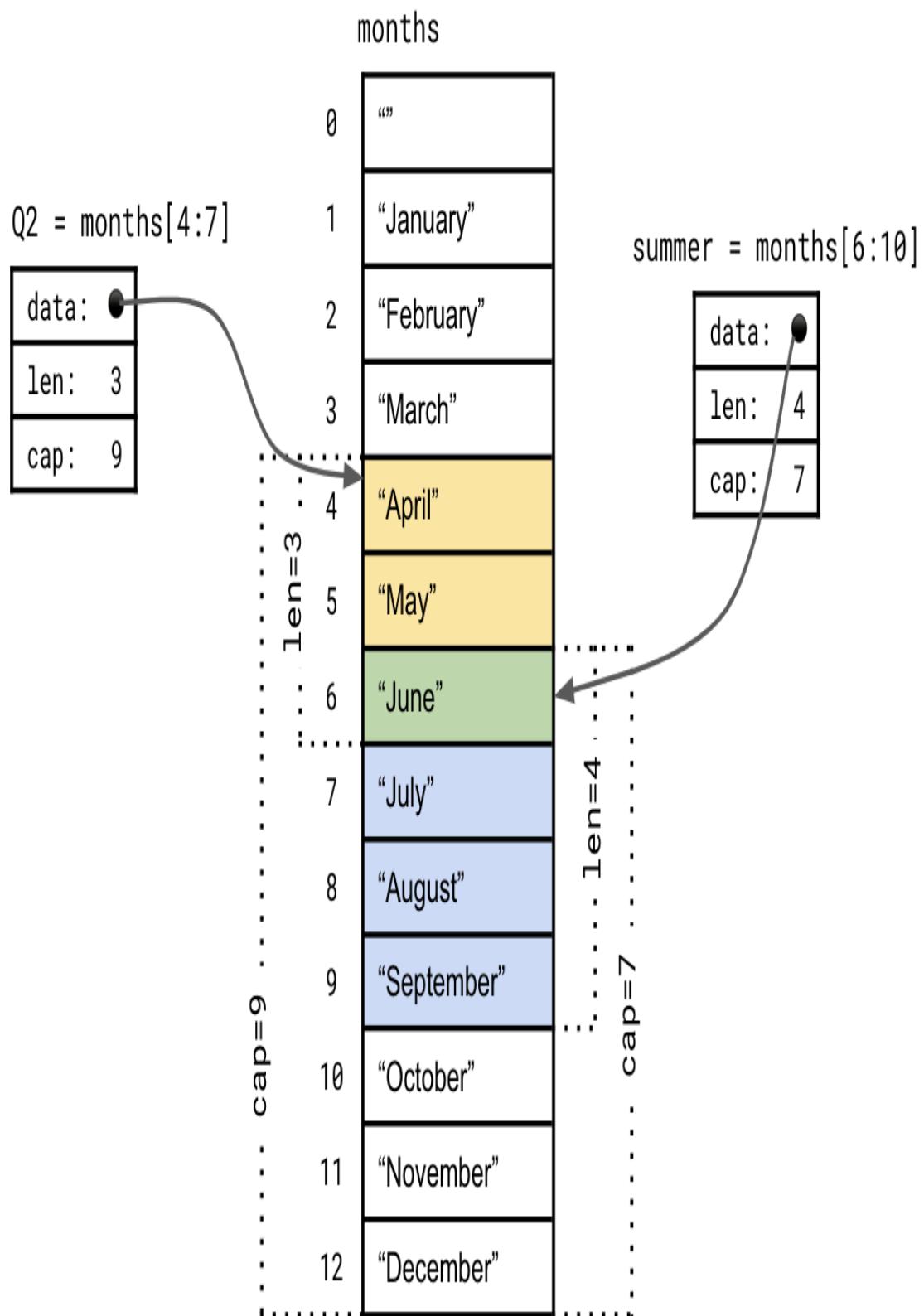


Figure 3-2. Two slices backed by the same array.

## The Slice Operator

Slices support a *slice operator* with the syntax `s[i:j]`, where `i` and `j` are in the range  $0 \leq i \leq j \leq \text{cap}(s)$ . This will produce a new slice backed by the same array with a length of  $j - i$ . If omitted, the values of `i` and `j` will default to `0` and `len(s)`, respectively.

For example:

```
s0 := []int{0, 1, 2, 3, 4, 5, 6}      // A slice literal
fmt.Println(s0)                         // "[0 1 2 3 4 5 6]"

s1 := s0[:4]
fmt.Println(s1)                         // "[0 1 2 3]"

s2 := s0[3:]
fmt.Println(s2)                         // "[3 4 5 6]"

s0[3] = 42                            // Change reflected in all 3
slices
fmt.Println(s0)                         // "[0 1 2 42 4 5 6]"
fmt.Println(s1)                         // "[0 1 2 42]"
fmt.Println(s2)                         // "[42 4 5 6]"
```

## Working With Slices

Creating a slice is somewhat different from an array: slices are typed only according to the type of their elements, not their number. The `make` builtin function can be used to create a zero value slice with a non-zero length as follows:

```
var n []int = make([]int, 3)    // Create an int slice with 3 elements

fmt.Println(len(n))              // "3"; len works for slices and
arrays

n[0] = 8
n[1] = 16
n[2] = 32
```

```
fmt.Println(n)           // "[8 16 32]"
```

Slices can be extended using the `append` builtin, which returns an extended slice containing one or more new values appended to the original ones.

The `append` is variadic, so it can accept any number of final arguments and will append them all to the input slice parameter.

```
m := []int{1}           // A literal []int declaration  
fmt.Println(m)          // "[1]"  
  
m = append(m, 2)  
m = append(m, 3, 4)  
m = append(m, m...)    // Append m to itself  
  
fmt.Println(m)          // "[1 2 3 4 1 2 3 4]"
```

### WARNING

Note that `append` *returns* the appended slice. Failing to accept it is a common error.

## Maps

Go's *map* data type references a *hash table*: an incredibly useful associative data structure that allows distinct keys to be arbitrarily "mapped" to values as key-value pairs. This data structure is common among today's mainstream languages: if you're coming to Go from one of these then you probably already use them, perhaps in the form of Python's `dict`, Ruby's Hash, or Java's `HashMap`.

Map types in Go are written `map[K]V`, where `K` and `V` are the types of its keys and values, respectively. Any type that is comparable using the `==` operator may be used as a key, and `K` and `V` need not be of

the same type. For example, string keys may be mapped to float32 values.

A map can be initialized using the built-in `make` function, and its values can be referenced using the usual `name[key]` syntax. Our old friend `len` will return the number of key/value pairs in a map; the `delete` builtin can remove key/value pairs.

```
freezing := make(map[string]float32)      // Empty map of string to
                                              float32

freezing["celsius"] = 0.0
freezing["fahrenheit"] = 32.0
freezing["kelvin"] = 273.2

fmt.Println(freezing["kelvin"])           // "273.2"
fmt.Println(len(freezing))                // "3"

delete(freezing, "kelvin")               // Delete "kelvin"
fmt.Println(len(freezing))                // "2"
```

Maps may also be initialized and populated as *map literals*:

```
freezing := map[string]float32{
    "celsius": 0.0,
    "fahrenheit": 32.0,
    "kelvin": 273.2,                      // The trailing comma is
                                              required
}
```

Requesting the value of a key that's not in a map won't throw an exception (which don't exist in Go) or return some kind of null value: rather, the zero value for its type is returned. This can be a very useful feature because it reduces a lot of boilerplate membership testing when working with maps, but it can be a little tricky when our map happens to contain zero valued values. Fortunately, accessing a map can also returns a second optional `bool` that indicates whether the key is present in the map:

```

freezing := map[string]float32{
    "celsius":    0.0,
    "fahrenheits": 32.0,
    "kelvin":     273.2,
}

newton, present := freezing["newton"] // What about the Newton
scale?
fmt.Println(newton)                // "0" (Correct for Newton
scale!)
fmt.Println(present)              // "false"

```

## Control Structures

Any programmer coming to Go from another language will find its suite of control structures to be generally familiar, even comfortable (at first) for those coming from a language heavily influenced by C. However, there are some pretty important deviations in their implementation and usages that might seem odd at first.

For example, control structure statements don't require lots of parentheses. Okay. Less clutter. That's fine.

There's also only one loop type. There is no `while`; only `for`. Seriously! It's actually pretty cool, though. Read on, and you'll see what we mean.

## Fun With `for`

The `for` statement is Go's one and only loop construct, and while there's no explicit `while` loop, Go's `for` can provide all of its functionality, effectively unifying all of the entry control loop types to which you've become accustomed.

Go has no `do-while` equivalent.

### The General `for` Statement

The general form of `for` loops in Go is nearly identical to that from other C-family languages, in which three statements — the init statement, the continuation condition, and the post statement — are separated by semicolons in the traditional style. Any variables declared in the init statement will be scoped only to the `for` statement:

```
sum := 0

for i := 0; i < 10; i++ {
    sum += 1
}

fmt.Println(sum)           // "10"
```

### NOTE

Unlike most C-family languages, `for` loops don't require parentheses around their clauses, and braces are not optional.

Interestingly, Go has made the init and post statements optional, making the `for` considerably more flexible and allowing it to fill the role of the traditional `while` loop:

```
sum, i := 0, 0

for i < 10 {
    sum += i
    i++
}

fmt.Println(i, sum)      // "10 45"
```

Omitting all three clauses from the `for` creates a block that loops infinitely, just like a traditional `while (true)`:

```
fmt.Println("For ever...")  
  
for {  
    fmt.Println("...and ever")  
}
```

## Looping Over Arrays and Slices

Go provides a useful keyword, `range`, that simplifies looping over a variety of data types.

In the case of arrays and slices, `range` can be used with a `for` to retrieve the index and the value of each element as it iterates:

```
s := []int{2, 4, 8, 16, 32}           // A slice of ints  
  
for i, v := range s {  
    fmt.Printf("%d -> %d\n", i, v)    // range gets each index/value  
    // Output the index and its  
    value  
}
```

As elsewhere in Go, unneeded return values can be discarded by using the *blank identifier* underscore operator:

```
a := []int{0, 1, 2, 3, 4}  
sum := 0  
  
for _, v := range a {  
    sum += v  
}  
  
fmt.Println(sum)    // "10"
```

## Looping Over Maps

The `range` keyword may be also be used with a `for` to loop over maps, with each iteration returning the current key and value.

```
m := map[int]string{  
    1: "January",  
    2: "February",
```

```
    3: "March",
    4: "April",
}

for k, v := range m {
    fmt.Printf("%d -> %s\n", k, v)
}
```

Note that maps aren't ordered, so the output won't be either:

```
3 -> March
4 -> April
1 -> January
2 -> February
```

## The if Statement

The typical application of the `if` statement in Go is consistent with other C-style languages, except for the lack of parentheses around the clause and the fact that braces are required:

```
if 7 % 2 == 0 {
    fmt.Println("7 is even")
} else {
    fmt.Println("7 is odd")
}
```

### NOTE

Remember: `if` statements don't require parentheses around their clauses, and braces are not optional.

Interestingly, Go allows an initialization statement to proceed the condition clause in an `if` statement, allowing for a particularly useful idiom. For example:

```
if _, err := os.Open("foo.ext"); err != nil {
    fmt.Println(err)
```

```
    } else {
        fmt.Println("All is fine.")
    }
```

In this example, the `err` variable is being initialized prior to a check for its definition, making it somewhat similar to:

```
_ , err := os.Open("foo.go")
if err != nil {
    fmt.Println(err)
} else {
    fmt.Println("All is fine.")
}
```

The two constructs are not equivalent however: in the former example `err` is scoped only to the `if` statement; in the latter example `err` is visible to the entire containing function.

## The switch Statement

As in other languages, Go provides a `switch` statement that provides a way to more concisely express a series of `if-then-else` conditionals. However, it differs from the traditional implementation in a number of ways that make it considerably more flexible.

Perhaps the most obvious difference to folks coming from C-family languages is that there's no fallthrough between the cases by default; this behavior can be explicitly added by using the `fallthrough` keyword.

```
i := 0

switch i % 3 {
case 0:
    fmt.Println("Zero")
    fallthrough
case 1:
    fmt.Println("One")
case 2:
```

```

        fmt.Println("Two")
default:
    fmt.Println("Huh?")
}

```

Which, due to the explicit fallthrough, outputs:

```

Zero
One

```

In Go, the case expressions don't need to be integers, or even constants: the cases will be evaluated from top to bottom, running the first case whose value is equal to the condition expression. If a switch has no expression it switches on true. These properties are demonstrated below:

```

hour := time.Now().Hour()

switch {
case hour >= 5 && hour < 9:
    fmt.Println("I'm writing")
case hour >= 9 && hour < 18:
    fmt.Println("I'm working")
default:
    fmt.Println("I'm sleeping")
}

```

Finally, as with if, a statement can precede the condition expression of a switch, in which case any defined values are scoped to the switch. For example, the above example can be rewritten as follows:

```

switch hour := time.Now().Hour(); { // Empty expression means "true"
case hour >= 5 && hour < 9:
    fmt.Println("I'm writing")
case hour >= 9 && hour < 18:
    fmt.Println("I'm working")
default:
    fmt.Println("I'm sleeping")
}

```

## Error Handling

Errors in Go are treated as just another value, represented by the built-in `error` type. This makes error handling straight-forward: idiomatic Go functions may include an `error`-typed value in its list of returns, which if not `nil` indicates an error state that may be handled via the primary execution path. For example, the `os.Open` function returns a non-`nil` error value when it fails to open a file:

```
file, err := os.Open("somefile.ext")
if err != nil {
    log.Fatal(err)
    return err
}
```

The actual implementation of the `error` type is actually incredibly simple: it's just a universally-visible interface that declares a single method:

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

This is very different from many language's system of throwing exceptions, which necessitate a dedicated system for exception catching and handling that can lead execution in unexpected logical directions.

## Creating an Error

There are two simple ways to create error values, and a more complicated way. The simple ways are to use either the `errors.New` or `fmt.Errorf` functions; the latter is handy because it provides string formatting too.

```
e1 := errors.New("error 42")
e2 := fmt.Errorf("error %d", 42)
```

However, the fact that `error` is an interface allows you to implement your own error types, if you need to. For example, a common pattern is to allow errors to be nested within other errors:

```
type NestedError struct {
    Message string
    Err      error
}

func (e *NestedError) Error() string {
    return fmt.Sprintf("%s\n contains: %s", e.Message, e.Err.Error())
}
```

For more information about errors and some good advice on error handling in Go take a look at Andrew Gerrand's [\*Error handling and Go\*](#) on The Go Blog.

## Putting the Fun in Functions: Variadics and Closures

### Functions

Declaring a function in Go is similar to most other languages: they have a name, a list of typed parameters, an optional list of return types, and a body. Go function declarations differ somewhat from other C-family languages: it make use of a dedicated `func` keyword; the type for each parameter follows its name; and return types are placed at the end of the function definition header and may be omitted entirely (there's no `void` type).

A function with a return type list must end with a `return` statement, except when execution can't reach the end of the function due to the presence of an infinite loop or a terminal panic before the function exits.

```
func add(x int, y int) int {
    return x + y
}

func main() {
    sum := add(10, 5)
    fmt.Println(sum)           // "15"
}
```

Additionally, a bit of syntactic sugar allows the type for a sequence of parameters or returns of the same type to be written only once. For example, the following definitions of `func foo` are equivalent:

```
func foo(i int, j int, a string, b string) { /* ... */ }
func foo(i, j int, a, b string)             { /* ... */ }
```

## Multiple Return Values

Functions can return any number of values. The list of return types for multiple returns is enclosed in parentheses:

```
func swap(x, y string) (string, string) {
    return y, x
}

func main() {
    a, b := swap("foo", "bar")
    fmt.Println(a, b)           // "bar foo"
}
```

## Recursion

Go allows *recursive* function calls, in which functions call themselves. Used properly, recursion can be a very powerful tool that can be applied to many types of problems. The canonical example is the calculation of the factorial of a positive integer, the product of all positive integers less than or equal to n:

```
func factorial(n int) int {
    if n == 0 {
```

```

        return 1
    }
    return n * factorial(n-1)
}

func main() {
    fmt.Println(factorial(11))      // "39916800"
}

```

## Defer

Go's `defer` keyword can be used to schedule the execution of a function until immediately before the surrounding function returns, and is commonly used to guarantee that resources are released or otherwise cleaned up.

```

func main() {
    defer fmt.Printf("cruel world")

    fmt.Printf("goodbye ")           // "goodbye cruel world"
}

```

In the following example, we create an empty file and attempt to write to it. A `closeFile` function is provided to close the file when we're done with it. However, if we simply call it at the end of `main`, an error could result in it never being called and the file being left in an open state. Therefore, we use a `defer` to ensure that the `closeFile` function is called before the function returns, however it returns.

```

func main() {
    file, err := os.Create("/tmp/foo.txt") // Create an empty file
    defer closeFile(file)                // Ensure closeFile(file)
is called
    if err != nil {
        return
    }

    _, err = fmt.Fprintln(file, "Your mother was a hamster")
    if err != nil {
        return
    }
}

```

```

        fmt.Println("File written to successfully")
    }

func closeFile(f *os.File) {
    err := f.Close()
    if err != nil {
        fmt.Println("Error closing file:", err.Error())
    } else {
        fmt.Println("File closed successfully")
    }
}

```

When we run the above, we (should) get the following output:

```

File written to successfully
File closed successfully

```

If multiple defer calls are used in a function, each is pushed onto a stack. When the surrounding function returns, the deferred calls are executed in last-in-first-out order. For example:

```

func main() {
    defer fmt.Println("world")
    defer fmt.Println("cruel")
    defer fmt.Println("goodbye")
}

```

The above function, when run, will output the following:

```

goodbye
cruel
world

```

## Pointers as Parameters

Much of the power of pointers becomes visible when they're combined with functions. Typically, function parameters are passed *by value*: when a function is called it receives a copy of each parameter, and changes made to the copy by the function don't

affect the caller. However, pointers contain a *reference* to a value, rather than the value itself, and can be used by a receiving function to indirectly modify the value passed to the function in a way that can affect the function caller.

The below function demonstrates both scenarios.

```
func main() {
    x := 5

    zeroByValue(x)
    fmt.Println(x)           // "5"

    zeroByReference(&x)
    fmt.Println(x)           // "0"
}

func zeroByValue(x int) {
    x = 0
}

func zeroByReference(x *int) {
    *x = 0                  // Dereference x and set it to 0
}
```

This behavior isn't unique to pointers. In fact, "under the hood" several data types are actually references to memory locations, including slices, maps, functions, and channels. Changes made to such *reference types* in a function can affect the caller, without needing to explicitly dereference them.

```
func main() {
    m := map[string]int{ "a" : 0, "b" : 1}

    fmt.Println(m)           // "map[a:0 b:1]"

    update(m)

    fmt.Println(m)           // "map[a:0 b:1 c:2]"
}
```

```
func update(m map[string]int) {
    m["c"] = 2
}
```

## Variadic Functions

A *variadic function* is one that may be called with zero or more trailing arguments. The most familiar example is the members of the `fmt.Printf` family of functions, which accept a single format specifier string and an arbitrary number of arguments.

This is the signature for the standard `fmt.Printf` function:

```
func Printf(format string, a ...interface{}) (n int, err error)
```

Note that it accepts a string, and an `interface{}`. If you're rusty on the `interface{}` syntax, we'll review it later in this chapter, but you can interpret `interface{}` to mean "some arbitrarily typed thing". What's most interesting here, however, is that the final argument contains an ellipsis: "...". This is the *variadic operator*, which indicates that the function may be called with any number of arguments of this type. For example, we can call `fmt.Printf` with a format and two differently-typed parameters.

```
const name, age = "Kim", 22
fmt.Printf("%s is %d years old.\n", name, age)
```

Within the variadic function, the variadic argument is a slice of the argument type. In the below example, the variadic `factors` parameter of the `product` method is of type `[]int` and may be ranged over accordingly.

```
func product(factors ...int) int {
    p := 1

    for _, n := range factors {
        p *= n
    }
}
```

```
    return p
}

func main() {
    fmt.Println(product(2, 2, 2)) // "8"
}
```

## Passing Slices as Variadic Values

What if your value is already in slice form, and you still want to pass it to a variadic function? Do you need to split it into multiple individual parameters? Goodness no.

In this case, you can apply the variadic operator after the variable name when calling the variadic function.

```
m := []int{3, 3, 3}
fmt.Println(product(m...)) // "27"
```

## Anonymous Functions and Closures

In Go, functions are *first-class values* that can be operated upon in the same way as any other entity in the language: they have types, may be assigned to variables, and may even be passed to and returned by other functions. The zero value of a function type is `nil`; calling a `nil` function value will cause a panic.

```
func sum(x, y int) int { return x + y }
func product(x, y int) int { return x * y }

func main() {
    var f func(int, int) int // Function variables have types

    f = sum
    fmt.Println(f(3, 5)) // "8"

    f = product // Legal: product has same type as sum
    fmt.Println(f(3, 5)) // "15"
}
```

Functions may be created within other functions as *anonymous functions*, which may be called, passed, or otherwise treated like any other functions. A particularly powerful feature of Go is that anonymous functions have access to the state of their parent, and retain that access *even after* the parent function has executed. This is, in fact, the definition of a *closure*.

### TIP

A *closure* is a nested function that has access to the variables of its parent function, even after the parent has executed.

Take, for example, the following `incrementor` function. This function has state, in the form of the variable `i`, and returns an anonymous function that increments that value before returning it. The returned function can be said to *close over* the variable `i`, making it a true (if trivial) closure.

```
func incrementor() func() int {
    i := 0

    return func() int {      // Return an anonymous function
        i++                  // "Closes over" parent function's i
        return i
    }
}
```

When we call `incrementor`, it creates its own new, local value of `i`, and returns a new anonymous function that will increment that value. Subsequent calls to `incrementor` will each receive their own copy of `i`. We can demonstrate that below.

```
func main() {
    increment := incrementer()

    fmt.Println(increment())      // "1"
```

```
    fmt.Println(increment())      // "2"
    fmt.Println(increment())      // "3"

    newIncrement := incrementer()
    fmt.Println(newIncrement())   // "1"
}
```

## Structs, Methods, and Interfaces

One of the biggest mental switches that people sometimes have to make when first coming to the Go language is that Go isn't a traditional object-oriented language. Not really. Sure, Go has types with methods, which kind of look like objects, but they don't have a prescribed inheritance hierarchy. Instead Go allows components to be assembled into a whole using *composition*.

For example, where a more strictly object oriented language might have a `Car` class that extends an abstract `Vehicle` class; perhaps it would implement `Wheels` and `Engine`. This sounds fine in theory, but these relationships can grow to become convoluted and hard to manage.

Go's composition approach, on the other hand, allows components to be "put together" without having to define their ontological relationships. Extending the above example, Go could have a `Car` struct, which could have its various parts, such as `Wheels` and `Engine`, embedded within it. Furthermore, methods in Go can be defined for any sort of data; they're not just for structs anymore.

## Structs

In Go, a *struct* is nothing more than an aggregation of zero or more fields as a single entity, where each field is a named value of an arbitrary type. A struct can be defined using the following type `Name struct` syntax. A struct is never `nil`: rather, the zero value of a struct is the zero value of all of its fields.

```

type Vertex struct {
    X, Y float64
}

func main() {
    var v Vertex           // Structs are never nil
    fmt.Println(v)          // "{0 0}"

    v = Vertex{}            // Explicitly define an empty struct
    fmt.Println(v)          // "{0 0}"

    v = Vertex{1.0, 2.0}    // Defining fields, in order
    fmt.Println(v)          // "{1 2}"

    v = Vertex{Y:2.5}      // Defining specific fields, by label
    fmt.Println(v)          // "{0 2.5}"
}

```

Struct fields can be accessed using the standard dot notation.

```

func main() {
    v := Vertex{X: 1.0, Y: 3.0}
    fmt.Println(v)           // "{1 3}"

    v.X *= 1.5
    v.Y *= 2.5

    fmt.Println(v)           // "{1.5 7.5}"
}

```

Structs are commonly created and manipulated by reference, so Go provides a little bit of syntactic sugar: pointers to structs can be accessed using dot notation; the pointers are automatically dereferenced.

```

func main() {
    var v *Vertex = &Vertex{1, 3}
    fmt.Println(v)           // &{1 3}

    v.X, v.Y = v.Y, v.X
    fmt.Println(v)           // &{3 1}
}

```

## Methods

In Go, *methods* are functions that are attached to types, including but not limited to structs. The declaration syntax for a method is very similar to that of a function, except that it includes an extra *receiver argument* before the function name that specifies the type that the method is attached to. When the method is called, the instance is accessible by the name specified in the receiver.

For example, our earlier `Vertex` type can be extended by attaching a `Square` method with a receiver named `v` of type `*Vertex`.

```
func (v *Vertex) Square() {    // Attach method to the *Vertex type
    v.X *= v.X
    v.Y *= v.Y
}

func main() {
    vert := &Vertex{3, 4}
    fmt.Println(vert)          // "&{3 4}"

    vert.Square()
    fmt.Println(vert)          // "&{9 16}"
}
```

### WARNING

Receivers are type specific: methods attached to a pointer type can only be called on a pointer to that type.

In addition to structs, you can also claim standard composite types — structs, slices, or maps — as your own, and attach methods to them. For example, we declare a new type, `MyMap`, which is just a standard `map[string]int`, and attach a `Length` method to it.

```
type MyMap map[string]int

func (m MyMap) Length() int {
```

```

        return len(m)
    }

func main() {
    mm := MyMap{"A":1, "B": 2}

    fmt.Println(mm)          // "map[A:1 B:2]"
    fmt.Println(mm.Length()) // "2"
}

```

## Interfaces

In Go, an *interface* is just a set of method signatures. As in other languages with a concept of an “interface”, they are used to describe the general behaviors of other types without being coupled to implementation details. An interface can thus be viewed as a *contract* that a type may satisfy, opening the door to powerful abstraction techniques.

For example, a Shape interface can be defined that includes an Area method signature. Any type that wants to be a Shape must have a method with type an Area method that returns a float64.

```

type Shape interface {
    Area() float64
}

```

Now we define two shapes, Circle and Rectangle, that satisfy the Shape interface by attaching an Area method to each one. Note that we don’t have to explicitly declare that they satisfy the interface: if a type possesses all of its methods, it can *implicitly satisfy* an interface. This is particularly useful when you want to design interfaces that are satisfied by types that you don’t own or control.

```

type Circle struct {
    Radius float64
}

func (c Circle) Area() float64 {

```

```

        return math.Pi * c.Radius * c.Radius
    }

type Rectangle struct {
    Width, Height float64
}

func (r Rectangle) Area() float64 {
    return r.Width * r.Height
}

```

Because both `Circle` and `Rectangle` implicitly satisfy the `Shape` interface, we can pass them to any function that expects a `Shape`.

```

func PrintArea(s Shape) {
    fmt.Printf("%T's area is %0.2f\n", s, s.Area())
}

func main() {
    r := Rectangle{Width:5, Height:10}
    PrintArea(r)                                // "main.Rectangle's area is
                                                // 50.00"

    c := Circle{Radius:5}
    PrintArea(c)                                // "main.Circle's area is
                                                // 78.54"
}

```

## Type Assertions

A *type assertion* can be applied to an interface value to “assert” its identity as a concrete type. The syntax takes the general form of `x.(T)`, where `x` is an expression of an interface and `T` is the asserted type.

Referring to the `Shape` interface and `Circle` struct we used above:

```

var s Shape
s = Circle{}                      // s is an expression of Shape
c := s.(Circle)                   // Assert that s is a Circle
fmt.Printf("%T\n", c)              // "main.Circle"

```

## The Empty Interface

One curious construct is the *empty interface*: `interface{}`. The empty interface specifies no methods. It carries no information; it says nothing<sup>5</sup>.

A variable of type `interface{}` can hold values of any type, which can be very useful when your code needs to handle values with of any type. The `fmt.Println` method is a good example of a function using this strategy.

There are downsides, however. Working with the empty interface requires certain assumptions to be made, which have to be checked at runtime and depend on reflection and type assertions and resulting in code that's code that's more fragile and less efficient.

## The Good Stuff: Concurrency

*Do not communicate by sharing memory. Instead, share memory by communicating.*

—Go Proverb

The intricacies of concurrent programming are many, and are well beyond the scope of this humble work. However, we can say that reasoning about concurrency is hard, and that the way concurrency is generally done makes it harder. In most languages, the usual approach to processes orchestration is to create some shared bit of memory, which is then wrapped in locks to restrict access to one process at a time, often introducing maddeningly difficult to debug errors such as race conditions or deadlocks.

Go, on the other hand, favors another strategy: it provides two concurrency primitives — goroutines and channels — that can be used together to elegantly structure concurrent software that don't depend quite so much on locking. It encourages developers to limit sharing memory, and to instead allow processes to interact with one other entirely by passing messages.

## Goroutines

One of Go's most powerful features is the go keyword. Any function call prepended with go keyword will run as usual, but the caller can proceed uninterrupted rather wait for the function to return. Under the hood, the function is executed as lightweight, concurrently executing processes called a *goroutine*.

The syntax is strikingly simple: a function foo, which may be executed sequentially as foo(), may be executed as a concurrent goroutine simply by adding the go keyword: go foo():

```
foo()      // Call foo() and wait for it to return
go foo()   // Spawn a new goroutine that calls foo() concurrently
```

Goroutines can also be used to invoke a function literal:

```
func Log(w io.Writer, message string) {
    go func() {
        fmt.Fprintln(w, message)
    }()
}
```

## Channels

In Go, *channels* are typed primitives that allow communication between two goroutines. They act as pipes into which a value can be sent and then received by a goroutine on the other end.

Channels may be created using the make function. Each channel can transmit values of a specific type, called its *element type*; channel types are written using the chan keyword followed by their element type. The example below declares and allocates an int channel:

```
var ch chan int = make(chan int)
```

The two primary operations support by channels are *send* and *receive*, both of which use the <- operator, where the arrow indicates

the direction of the data flow as demonstrated below:

```
ch <- val      // Sending on a channel
val = <-ch     // Receiving on a channel and assigning it to val
<-ch          // Receiving on a channel and discarding the result
```

## Channel Blocking

By default, a channel is *unbuffered*. Unbuffered channels have a very useful property in that sends on them block until another goroutine receives on the channel, and receives block until another goroutine sends on the channel. This behavior can be exploited to synchronize two goroutines, as demonstrated below:

```
func main() {
    ch := make(chan string)      // Allocate a string channel

    go func() {
        message := <-ch          // Blocking receive; assigns to message
        fmt.Println(message)      // "ping"
        ch <- "pong"             // Blocking send
    }()

    ch <- "ping"                // Send "ping"
    fmt.Println(<-ch)           // "pong"
}
```

Although `main` and the anonymous goroutine run concurrently and could in theory run in any order, the blocking behavior of unbuffered channels guarantees that the output will always be “ping” followed by “pong”.

## Channel Buffering

Go channels may be *buffered*, in which case they contain an internal value queue with a fixed *capacity* that’s specified when the buffer is initialized. Sends to a buffered channel only block when the buffer is full; receives from a channel only block when the buffer is empty. Any

other time, send and receive operations write to or read from the buffer, respectively, and exit immediately.

A buffered channel can be created by providing a second argument to the `make` function to indicate its capacity.

```
ch := make(chan string, 2)      // Buffered channel with capacity 2  
  
ch <- "foo"                  // Two non-blocking sends  
ch <- "bar"  
  
fmt.Println(<-ch)             // Two non-blocking receives  
fmt.Println(<-ch)  
  
fmt.Println(<-ch)             // The third receive will block
```

## Closing Channels

The third available channel operation is `close`, which sets a flag to indicate that no more values will be sent on it. The builtin `close` function can be used to close a channel: `close(ch)`.

### TIP

The channel `close` operation is just a flag to tell the receiver not to expect any more values. You don't *have to* explicitly close channels.

Trying to send on a closed channel will cause a panic. Receiving from a closed channel will retrieve any values sent on the channel prior to its closure; any subsequent receive operations will immediately yield the zero value of the channel's element type. Receivers may also test whether a channel has been closed (and its buffer is empty) by assigning a second `bool` parameter to the receive expression:

```
ch := make(chan string, 10)
```

```

ch <- "foo"

close(ch)                                // One value left in the buffer

msg, ok := <-ch
fmt.Printf("%q, %v\n", msg, ok)      // "foo", true

msg, ok = <-ch
fmt.Printf("%q, %v\n", msg, ok)      // "", false

```

## WARNING

While either party may close a channel, in practice only the sender should do so. Inadvertently sending on a closed channel will cause a panic.

## Looping Over Channels

The `range` keyword may be used to loop over channels that are open or contain buffered values. The loop will block until a value is available to be read; if the channel is closed the loop will exit immediately. In the example below, we create a buffered channel, to which we add three values to before closing. Because the channel contains three values to be read before the channel was closed, looping over this channel will output all three lines:

```

ch := make(chan string, 3)

ch <- "foo"                                // Send three (buffered) values to the
channel
ch <- "bar"
ch <- "baz"

close(ch)                                    // Close the channel

for s := range ch {                         // Range will continue to the "closed"
flag
    fmt.Println(s)
}

```

## Select

Go's `select` statements are a little like `switch` statements that provide a convenient mechanisms for multiplexing communications with multiple channels. The syntax for `select` is very similar to `switch`, with some number of `case` statements that specifies code to be executed upon a successful send or receive operation:

```
select {
    case <-ch1:                                // Discard received value
        fmt.Println("Got something")

    case x := <-ch2:                            // Assign received value to x
        fmt.Println("Got", x)

    case ch3 <- y:                            // Send y to channel
        fmt.Println("Sent", y)

    default:
        fmt.Println("None of the above")
}
```

If no cases are ready, the `default` statements will be executed. If there's no `default`, then the `select` will block until one of its cases is ready, at which point it performs the associated communication and executes the associated statements. If multiple cases are ready, `select` will execute one at random.

### GOTCHA!

When using `select`, keep in mind that a closed channel never blocks and is always readable.

## Implementing Channel Timeouts

The ability to use `select` to multiplex on channels can be very powerful, and can make otherwise very difficult or tedious tasks

trivial. Take, for example, the implementation of a timeout on an arbitrary channel. In some languages this might require some awkward thread work, but a `select` with a call to `time.After`, which returns a channel that sends a message after a specified duration, makes short work of it:

```
var ch chan int

select {
    case m := <-ch:                                // Read from ch; blocks forever
        fmt.Println(m)

    case <-time.After(10 * time.Second):           // time.After returns a channel
        fmt.Println("timed out")
}
```

---

1 Perlis, Alan. *ACM SIGPLAN Notices* 17(9), September 1982, pp. 7–13.

2 Note to self: Gopher fedoras and beard combs.

3 See what I did there?

4 Earlier versions of C, C++, and Python lacked a native Boolean type, instead representing them using the integers 0 (for false) or 1 (for true). Some languages like Perl, Lua, and Tcl still use a similar strategy.

5 Pike, Rob. “Go Proverbs”. YouTube. 1 Dec. 2015,  
<https://youtu.be/PAAkCSZUG1c?t=458>

# Chapter 4. Cloud Native Patterns

---

## A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 4th chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at [ablevins@oreilly.com](mailto:ablevins@oreilly.com).

*Progress is possible only if we train ourselves to think about programs without thinking of them as pieces of executable code<sup>1</sup>.*

—Edsger W. Dijkstra, August 1979

In 1991, while still at Sun Microsystems, L Peter Deutsch<sup>2</sup> formulated the *Fallacies of Distributed Computing*, which list some of the false assumptions that programmers new (and not so new) to distributed applications often make:

- **The network is reliable:** Switches fail, routers get misconfigured.
- **Latency is zero:** It takes time to move data across a network.

- **Bandwidth is infinite:** A network can only handle so much data at a time.
- **The network is secure:** Don't share secrets in plain text. Encrypt everything.
- **Topology doesn't change:** Servers and services come and go.
- **There is one administrator:** Multiple people leads to heterogeneous solutions.
- **Transport cost is zero:** Moving data around costs time and money.
- **The network is homogeneous:** Every network is (sometimes very) different.

If I might be so audacious, I'd like to add a ninth one as well:

- **Services are reliable:** Services that you depend on can fail at any time.

In this chapter, we present a selection of idiomatic patterns — tested, proven development paradigms — designed to address one or more of the conditions described in Deutsch's *Fallacies*, and demonstrate how to implement them in Go. None of the patterns discussed in this book are original — some have been around for as long as distributed applications have existed — but most haven't been previously published in a single work. Many of them are unique to Go to or have novel implementations in Go relative to other languages.

Unfortunately, this book won't cover infrastructure-level patterns, like the Bulkhead or Gatekeeper patterns. Largely, this is because our focus is on application-layer development in Go and those patterns — while indispensable — function at an entirely different abstraction level. But if you're interested in learning more, please take a look at

*Cloud Native Infrastructure* by Justin Garrison and Kris Nova and *Designing Distributed Systems* by Brendan Burns (both by O'Reilly Media).

## Layout of this Chapter

The general presentation of each pattern in this chapter is loosely based on the one used in the famous “Gang of Four” *Design Patterns* book, but simpler and less formal. Each pattern opens with a very brief description of its purpose and the reasons for using it, and is followed by the following sections:

- **Applicability:** Context and descriptions of where this pattern may be applied.
- **Participants:** A listing of the components of the pattern and their roles.
- **Implementation:** A discussion of the solution and its implementation.
- **Sample Code:** A demonstration of how the code may be implemented in Go.

## The Context Package

A number of the code examples in this chapter make use of the context package, which was introduced in Go 1.7 to provide an idiomatic means of carrying deadlines, cancelation signals, and request-scoped values between processes. It contains a single interface, `context.Context`, whose methods are listed below.

```
type Context interface {
    // Done returns a channel that's closed when this Context is
    // canceled.
    Done() <-chan struct{}
```

```

    // Err indicates why this context was canceled after the Done
    channel is
    // closed. If Done is not yet closed, Err returns nil.
    Err() error

    // Deadline returns the time when this Context should be canceled;
    it
    // returns ok==false if no deadline is set.
    Deadline() (deadline time.Time, ok bool)

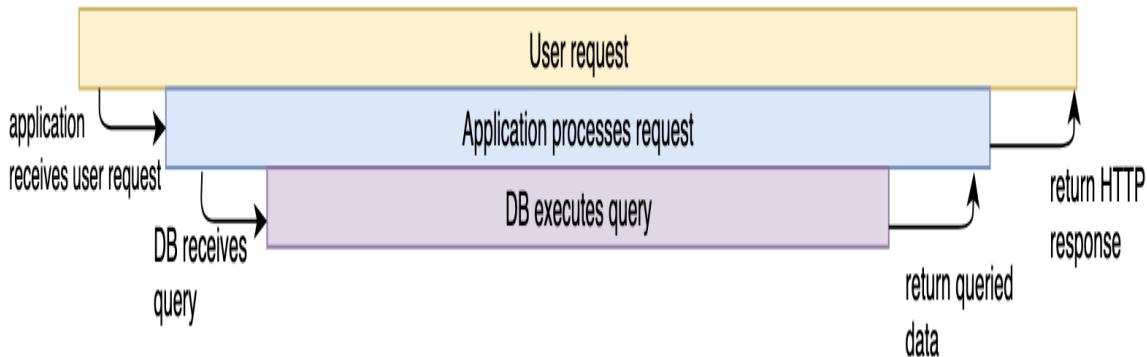
    // Value returns the value associated with this context for key,
    or nil
    // if no value is associated with key. Use with care.
    Value(key interface{}) interface{}
}

```

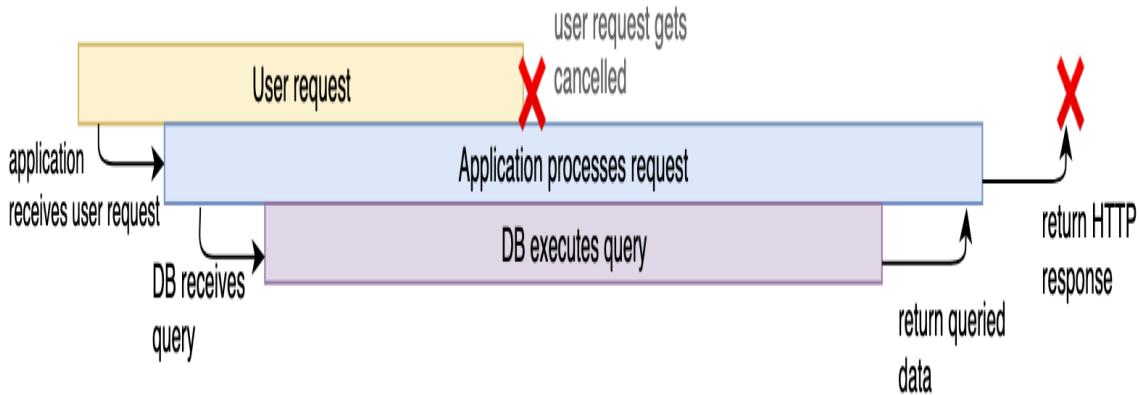
## What Context Can Do for You

A Context is used by passing it directly to a new service request upon its creation, and may then be passed to any sub-requests. What makes this useful is that when the Context canceled, all functions holding it (or a derived Context; more on this below) will receive the signal, allowing them to coordinate their cancelation and reducing the amount of wasted effort.

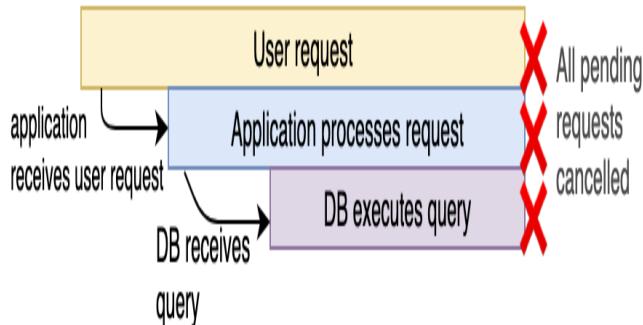
Take, for example, a request from a user to a service, which in turn makes a request to a database. In an ideal scenario, that could be diagrammed as follows:



In most cases, however, if the user terminated her request before it was fully completed, the subprocesses might continue anyway, consuming resources, oblivious to the overall context of the request.



However, by using a Context, and sharing that Context to each subsequent service request, all long-running processes can be sent a simultaneous “done” signal, allowing a single cancelation signal to be coordinated among each of the processes.



Importantly, the same Context value can be passed to functions running in different goroutines: Context values can be safely used simultaneously by multiple goroutines.

## Creating Context

A brand-new Context can be obtained using one of two functions:

`func Background() Context`

Returns an empty Context that's never canceled, has no values, and has no deadline. It is typically used by the main function, initialization, and tests, and as the top-level Context for incoming requests.

`func context.TODO() Context`

Also provides an empty Context, but it's intended to be used as a placeholder when it's unclear which Context to use or it is not yet available.

## Defining Context Deadlines and Timeouts

The context package also includes a number of methods for creating *derived* Context values that allow you to direct its cancelation behavior, either by applying a timeout or by providing a function hook that can be used to explicitly trigger a cancelation.

`func WithDeadline(Context, time.Time) (Context, CancelFunc)`

Accepts a specific time at which the Context will be cancelled and Done will be closed.

`func WithTimeout(Context, time.Duration) (Context, CancelFunc)`

Accepts a duration after which the Context will be cancelled and Done will be closed.

`func WithCancel(Context) (Context, CancelFunc)`

Unlike the above functions, WithCancel accepts nothing, but returns a function that can be called to cancel the Context.

All three of the above functions return the derived Context that includes any requested decoration, and a CancelFunc, a zero-

parameter function that can be called to explicitly cancel the Context and all of its derived values.

### TIP

When a Context is canceled, all Contexts that are *derived from it* are also canceled. Contexts that *it was derived from* are not.

## Defining Request-Scope Values

Finally, the context package includes one final function, which can be used to define an arbitrary *request-scoped* key-value pair that can be accessed by the returned Context and all Context values derived from it.

`func WithValue(parent Context, key, val interface{}) Context`

WithValue returns a derivation of parent in which key is associated with the value val.

## ON CONTEXT VALUES

The `context.WithValue` and `context.Value` functions provide convenient mechanisms for setting and getting arbitrary key-value pairs that can be used by consuming processes and APIs. However, it has been argued that this functionality is orthogonal to Context's function of orchestrating the cancelation of long-lived requests, obscures your program's flow, and can easily break compile-time coupling. For a more in-depth discussion, please see Dave Cheney's blog post [Context is for cancelation](#).

This functionality isn't used in any of the examples in this chapter (or this book). If you choose to make use of it, please take care to ensure that all of your values are scoped only to the request, don't alter the functioning of any processes, and don't break your processes if they happen to be absent.

## Using a Context

As discussed above, when a service request is initiated, either by an incoming request or triggered by the `main` function, the top-level process will use the `Background` function to create a new Context value, possibly decorating it with values or timeouts with one or more of the various `context.WithFoo` functions, and passing it along to any sub-requests. Those requests then only need to watch the `Done` channel for cancelation signals.

For example, take a look at the following `Stream` function:

```
func Stream(ctx context.Context, out chan<- Value) error {
    // Create a derived Context with a 10 second timeout and passes it
    to
    // ServiceCall. It will be cancelled upon timeout, but ctx will
    not.
    dctx := context.WithTimeout(ctx, time.Second * 10)
    res, err := ServiceCall(dctx)
    if err != nil {                                // Can be true if dctx
```

```

times out
    return err
}

for {
    select {
        case out <- res:                                // Read from res; send to
out
        case <-ctx.Done():                            // Triggered when Done
closes
        return ctx.Err()
    }
}
}

```

Stream receives a `ctx Context` as an input parameter, which it sends to `WithTimeout` to create `dctx`, a derived Context with a 10-second timeout. Because of this decoration, the `ServiceCall(dctx)` call could possibly time out after ten seconds and return an error. Functions using the original `ctx`, however, will not have this timeout decoration, and will not time out.

Further down, the original `ctx` value is used in a `for` loop around a `select` statement to retrieve values from the `res` channel provided by the `ServiceCall` function. Note the `case <-ctx.Done()` statement, which is executed when the `ctx.Done` channel closes to return an appropriate error value.

## Stability Patterns

The stability patterns are generally intended to be applied by distributed applications to improve their own stability and the stability of the larger system they're a part of.

## Circuit Breaker

Circuit Breaker automatically degrades service functions in response to a likely fault, preventing larger or cascading failures by eliminating

recurring errors and providing reasonable error responses.

## Applicability

An undeniable fact of life for distributed, cloud native systems is that failures happen. Services become misconfigured, databases crash, networks partition. We can't prevent it.

Failing to account for the inevitable failure of the services that our own service depends upon can have some pretty unpleasant consequences. We've all seen them; they aren't pretty. Some services might keep futilely trying to do their job and returning nonsense to their client; others might fail catastrophically and maybe even fall into a crash/restart death spiral. It doesn't matter, because in the end they're all wasting resources, obscuring the source of original failure, and making cascading failures even more likely.

On the other hand, a service that's designed with the assumption that its dependencies can fail at any time can respond sanely when they do. The Circuit Breaker allows a service to detect such failures and to "open the circuit" by stopping servicing requests, and instead providing clients with an error message consistent with the service's communication contract.

For example, imagine a service that receives a request from a client, executes a database query, and returns a response. What if the database fails? The service might keep trying to query the database, flooding the logs with error messages, and eventually timing out or returning useless 500 errors. Such a service can use a Circuit Breaker to "open the circuit" when the database fails, prevents the service from making any more doomed database requests (at least for a while), and responding to the client immediately with a meaningful status.

## Participants

*Circuit*

The function that interacts with the service.

### *Breaker*

A closure with the same function signature as Circuit.

## Implementation

Essentially, the Circuit Breaker is just a specialized Adapter pattern, with Breaker wrapping Circuit to add some additional error handling logic.

Breaker can have two states. In the *closed* state everything is functioning normally. All requests received from the client by Breaker are forwarded unchanged to Circuit, and all responses from Circuit are forwarded back to the client. In the *open* state, Breaker doesn't forward requests to Circuit. Instead it "fails fast" by responding with an informative error message.

Breaker internally tracks the errors returned by Circuit; if the number of consecutive errors returned by Circuit exceeds a defined threshold, Breaker *trips* and its state switches to *open*.

Most implementations of Circuit Breaker include some logic to automatically close the circuit after some period of time. Keep in mind, though, that hammering an already malfunctioning service with lots of retries can cause its own problems, so it's standard to include some kind of *backoff*, in which the rate of retries is slowly decreased over time.

In a multi-node service, this implementation may be extended to include some shared storage mechanism, such Memcached or Redis network cache, to track the circuit state.

## Sample Code

We begin by creating Circuit type that specifies the signature of the function that's interacting with your database or other upstream

service. In practice, this can take whatever form is appropriate for your functionality. It should include an error in its returns, however.

In this example, we use a Context value, which is often used to carry things like deadlines and cancelation signals between processes. Your implementation may vary.

```
type Circuit func(context.Context) (string, error)
```

The Breaker function accepts c, a function of type Circuit, and a failure threshold.

It returns a closure, also of type Circuit. The closure counts the number of consecutive errors returned by c, and returns the error “Service unreachable” if that value exceeds the failure threshold.

In this example, the closure will allow requests to interact with the service again after several seconds, with exponential backoff.

```
type Circuit func(context.Context) (string, error)

func Breaker(circuit Circuit, failureThreshold uint64) Circuit {
    var lastStateSuccessful = true
    var consecutiveFailures uint64 = 0
    var lastAttempt time.Time = time.Now()

    return func(ctx context.Context) (string, error) {
        if consecutiveFailures >= failureThreshold {
            backoffLevel := consecutiveFailures - failureThreshold
            shouldRetryAt := lastAttempt.Add(time.Second * 2 <<
backoffLevel)

            if !time.Now().After(shouldRetryAt) {
                return "", errors.New("circuit open -- service
unreachable")
            }
        }

        lastAttempt = time.Now()
        response, err := circuit(ctx)

        if err != nil {
```

```
        if !lastStateSuccessful {
            consecutiveFailures++
        }
        lastStateSuccessful = false
        return response, err
    }

    lastStateSuccessful = true
    consecutiveFailures = 0

    return response, nil
}
}
```

## Debounce

Debounce limits the frequency of a function call to one among a cluster of invocations.

### Applicability

Debounce is the second of our patterns to be labeled with an electrical circuit theme. Specifically, it's named after a phenomenon in which a switch's contacts "bounce" when they're opened or closed, causing the circuit to fluctuate a bit before settling down. It's usually no big deal, but this "contact bounce" can be a real problem in logic circuits where a series of on/off pulses can be interpreted as a data stream. The practice of eliminating contact bounce so that only one signal is transmitted by an opening or closing contact is called "debouncing".

In the world of services, we sometimes find ourselves needing to perform a cluster of potentially slow or costly operations where only one will do. Using the Debounce pattern, a series of similar calls that are tightly-clustered in time are restricted to only one call, typically the first or last in a batch.

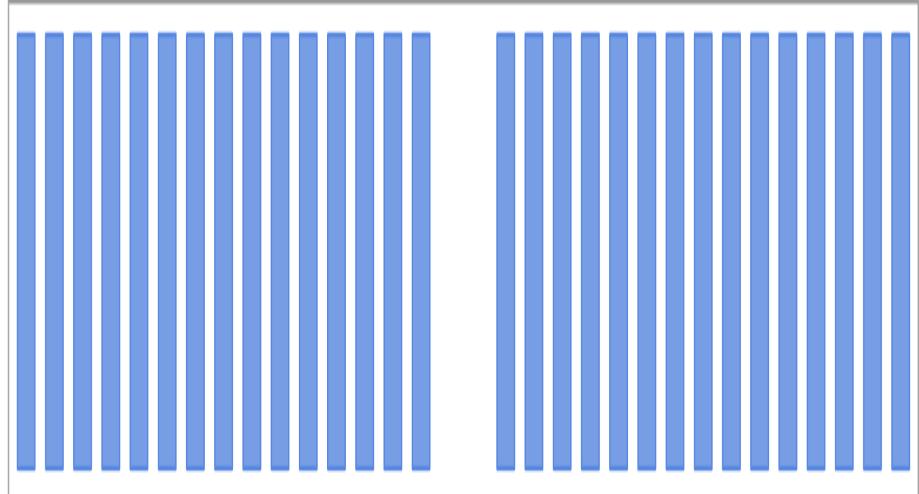
This technique has been used in the JavaScript world for years to limit the number of operations that could slow the browser by taking

only the first in a series of user events or to delay a call until a user is ready. You've probably seen an application of this technique in practice before. We're all familiar with the experience of using a search bar whose autocomplete popup doesn't display until after you pause typing, or spam-clicking a button only to see the clicks after the first ignored.

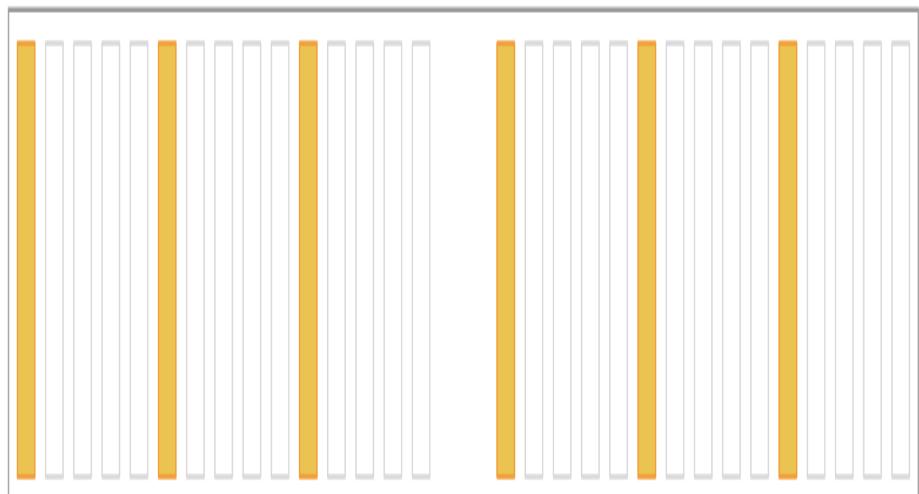
Those of us who specialize in back-end services can learn a lot from our front-end brethren, who have been working for years to account for the reliability, latency, and bandwidth issues inherent to distributed systems. For example, this approach could be used to retrieve some slowly-updating remote resource without bogging down wasting both client and server time with wasteful requests.

## **WHAT'S THE DIFFERENCE BETWEEN DEBOUNCE AND THROTTLE?**

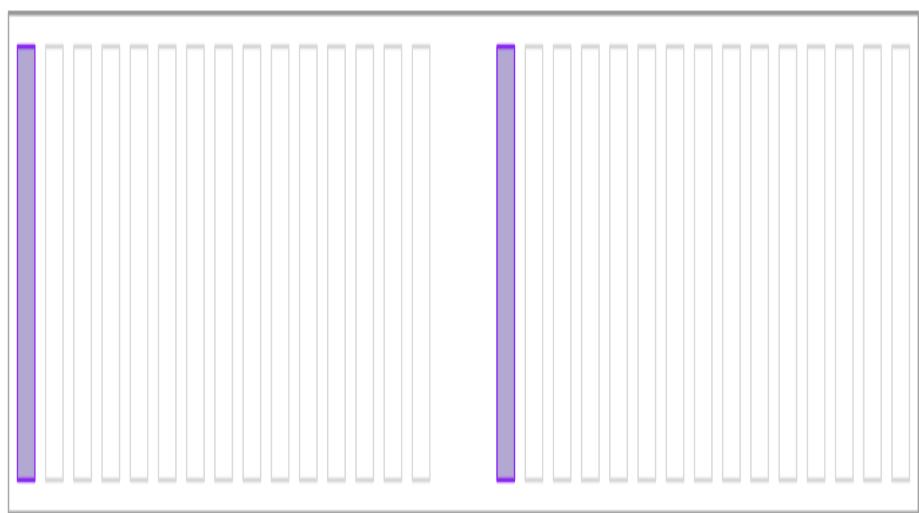
Input



Throttle



Debounce



*Figure 4-1. Throttle limits the event rate; debounce allows only one event in a cluster.*

Conceptually, Debounce and Throttle seem fairly similar. After all, they're both about reducing the number of calls per unit of time. However, as illustrated in [Figure 4-1](#), the precise timing of each differs quite a bit.

- *Throttle* works like the throttle in a car, limiting the amount of fuel going into the engine by capping the flow of fuel to some maximum rate. This is illustrated on the figure above: no matter how many times the input function is called, Throttle only allows a fixed amount of calls to proceed per unit of time.
- *Debounce* focuses on clusters of activity, making sure that a function is called only once during a cluster of requests, either at the start or the end of the cluster. A function-first debounce implementation is illustrated above: for each of the two clusters of calls to the input function, Debounce only allows one call to proceed at the beginning of each cluster.

## Participants

### *Circuit*

The function to regulate.

### *Debounce*

A closure with the same function signature as Circuit.

## Implementation

The Debounce implementation is actually very similar to the one for Circuit Breaker in that it wraps Circuit to provide the rate-limiting

logic. That logic is actually quite straight-forward: on *every* call of the outer function function — regardless of its outcome — a time interval is set. Any subsequent call made before that time interval expires is ignored; any call made afterwards is passed along to the inner function. This implementation, in which the inner function is called once and subsequent calls are ignored, is called *function-first*, and is useful because it allows the initial response from the inner function to be cached and returned.

A *function-last* implementation will wait for a pause after a series of calls before calling the inner function. This variant is common in the JavaScript world when a programmer wants certain amount of input before making a function call, such as when a search bar waits for a pause in typing before autocompleting. Function-last tends to be less common in back end services because it doesn't provide an immediate response, but it can be useful if your function doesn't need results right away.

## Sample Code

Just like in the Circuit Breaker implementation, we start by defining a function type with the signature of the function we want to limit. Also like Circuit Breaker, we call it `circuit`; it's identical to the one declared in that example. Again, `Circuit` this can take whatever form is appropriate for your functionality, but it should include an `error` in its returns.

```
type Circuit func(context.Context) (string, error)
```

The similarity with the Circuit Breaker implementation is quite intentional: their compatibility makes them “chainable”, as demonstrated below.

```
func myFunction func(ctx context.Context) (string, error) { /* ... */  
}  
  
response, err := Breaker(Debounce(myFunction(ctx)))
```

The function-first implementation of Debounce is very straight-forward compared to function-last because it only needs to track the last time it was called and return a cached result if it's called again less than `d` time after.

```
func DebounceFirst(circuit Circuit, d time.Duration) Circuit {
    var threshold time.Time
    var cResult string
    var cError error

    return func(ctx context.Context) (string, error) {
        if threshold.Before(time.Now()) {
            cResult, cError = circuit(ctx)
        }

        threshold = time.Now().Add(d)
        return cResult, cError
    }
}
```

Our function-last implementation is a bit more awkward because it involves the use of a `time.Ticker` to determine whether enough time has passed since the function was last called, and to call `circuit` when it has. Alternatively, we could create a new `time.Ticker` with every call, but that can get quite expensive if it's called frequently.

```
type Circuit func(context.Context) (string, error)

func DebounceLast(circuit Circuit, d time.Duration) Circuit {
    var threshold time.Time = time.Now()
    var ticker *time.Ticker
    var result string
    var err error

    return func(ctx context.Context) (string, error) {
        threshold = time.Now().Add(d)

        if ticker == nil {
            ticker = time.NewTicker(time.Millisecond * 100)
            tickerc := ticker.C
        }
```

```

go func() {
    defer ticker.Stop()

    for {
        select {
        case <-tickerc:
            if threshold.Before(time.Now()) {
                result, err = circuit(ctx)
                ticker.Stop()
                ticker = nil
                break
            }
        case <-ctx.Done():
            result, err = "", ctx.Err()
            break
        }
    }()
}

return result, err
}
}

```

## Retry

Retry accounts for a possible transient fault in a distributed system by transparently retrying a failed operation.

### Applicability

Transient errors are a fact of life when working with complex distributed systems. These can be caused by any number of temporary conditions, especially if the upstream service or network resource has protective strategies in place, such as throttling that temporarily rejects requests under high workload, or adaptive strategies like autoscaling that can add capacity when needed.

These faults typically resolve themselves after a bit of time, so repeating the request after a reasonable delay is likely to be successful. Failing to account for transient faults can lead to a

system that's unnecessarily brittle. On the other hand, implementing an automatic retry strategy can considerably improve the stability of the service that can benefit both it and its downstream consumers.

## Participants

### *Effector*

The function that interacts with the service.

### *Retry*

A function that accepts Effector and returns a closure with the same function signature as Effector.

## Implementation

This pattern works similarly to Circuit Breaker or Debounce in that there is a type, Effector, that defines a function signature. This signature can take whatever form is appropriate for your implementation, but when the function executing the potentially-failing operation is implemented, it must match the signature defined by Effector.

The Retry function accepts the user-defined Effector and returns an Effector that wraps the user-defined function to provide the retry logic. Along with the user-defined function, Retry also accepts `retries`, an integer describing the maximum number of retry attempts that the retry logic will make, and `delay`, a `time.Duration` that describes that amount of time that it will wait between each retry attempt. If the `retries` parameter is 0, then the retry logic will effectively become a no-op.

## Sample Code

```
type Effector func(context.Context) (string, error)

func Retry(effectector Effector, retries int, delay time.Duration)
```

```

Effector {
    return func(ctx context.Context) (string, error) {
        for r := 0; ; r++ {
            response, err := effector(ctx)
            if err == nil || r >= retries {
                return response, err
            }

            log.Printf("Attempt %d failed; retrying in %v",
delay)

            select {
            case <-time.After(delay):
            case <-ctx.Done():
                return "", ctx.Err()
            }
        }
    }
}

```

To use this, we implement the function that executes the potentially-failing operation and whose signature matches the `Effector` type; this role is played by `EmulateTransientError` in the below example. Passing this function to `Retry` yields a function variable whose signature also matches `Effector`; this is `r` below.

When the function variable `r` is called, `EmulateTransientError` is called, and called again after a delay if it returns an error, according to the above retry logic.

```

var count int

func EmulateTransientError(ctx context.Context) (string, error) {
    count++

    if count <= 3 {
        return "intentional fail", errors.New("error")
    } else {
        return "success", nil
    }
}

```

```

func main() {
    r := Retry(EmulateTransientError, 5, 2*time.Second)

    res, err := r(context.Background())

    fmt.Println(res, err)
}

```

## Throttle

Throttle limits the frequency of a function call to some maximum number of invocations per unit of time.

### Applicability

The Throttle pattern is named after a device used to manage the flow of a fluid, such as the amount of fuel going into a car engine. Like its namesake mechanism, Throttle is used to restrict the number of times that a function can be called during over a period of time.

For example:

- A user may only be allowed 10 service requests per second.
- A client may restrict itself to call a particular function once every 500 milliseconds.
- An account may only be allowed 3 failed login attempts in a 24 hour period.

Perhaps the most common reason to apply a Throttle is to account for sharp activity spikes that could saturate the system's with a possibly unreasonable number of requests that may be expensive to satisfy, or lead to service degradation and eventually failure. While it may be possible for a system to scale up to add sufficient capacity to meet user demand, this takes time and may not be able to react quickly enough.

This pattern is similar to Debounce in that it places a limitation on the frequency of a function call, but where Debounce restricts clusters of

invocations, Throttle simply limits according to time period. For more on the difference between the Debounce and Throttle patterns, see “[What’s the difference between Debounce and Throttle?](#)”.

## Participants

### *Effector*

The function to regulate.

### *Throttle*

A function that accepts Effector and returns a closure with the same function signature as Effector.

## Implementation

The Throttle pattern is similar to many of the other patterns described in this chapter: it’s implemented as a function that accepts an effector function, and returns a Throttle closure with the same signature that provides the rate-limiting logic.

The most common algorithm for implementing rate-limiting behavior is the “token bucket<sup>3</sup>”, which uses the analogy of a bucket that can hold some maximum number of tokens. When a function is called, a token is taken from the bucket, which then refills at some fixed rate.

The way that a Throttle treats requests when there are insufficient tokens in the bucket to pay for it can vary depending according to the needs of the developer. Some common strategies are:

### *Return an error*

This is the most basic strategy, and is common when you’re only trying to restricting unreasonable or potentially abusive numbers of client requests. A RESTful service adopting this strategy might respond with a status 429 (Too Many Requests).

### *Replay the response of the last successful function call*

This strategy can be useful when a service or expensive function call is likely to provide an identical result if called too soon. It's commonly used in the JavaScript world.

*Enqueue the request for execution when sufficient tokens are available*

This approach can be useful when you want to eventually handle all requests, but it's also more complex and may require care to be taken to ensure that memory isn't exhausted.

## Sample Code

The below example implements a very basic “token bucket” algorithm that uses the “replay” strategy. It’s similar to our other examples in that it wraps an effector function `e` in a closure that contains the rate-limiting logic. The bucket is initially allocated `max` tokens; each time the closure is triggered it checks whether it has any remaining tokens. If tokens are available, it decrements the token count by one and triggers the effector function. If not, the last recorded result is replayed. Tokens are added at a rate of `refill` tokens every duration `d`.

```
type Effector func(context.Context) (string, error)

func Throttle(e Effector, max uint, refill uint, d time.Duration)
Effector {
    var ticker *time.Ticker = nil
    var tokens uint = max

    var lastReturnString string
    var lastReturnError error

    return func(ctx context.Context) (string, error) {
        if ctx.Err() != nil {
            return "", ctx.Err()
        }

        if ticker == nil {
```

```

ticker = time.NewTicker(d)
defer ticker.Stop()

go func() {
    for {
        select {
        case <-ticker.C:
            t := tokens + refill
            if t > max {
                t = max
            }
            tokens = t
        case <-ctx.Done():
            ticker.Stop()
            break
        }
    }
}()

if tokens > 0 {
    tokens--
    lastReturnString, lastReturnError = e(ctx)
}

return lastReturnString, lastReturnError
}
}

```

## Timeout

Timeout allows a process to stop waiting for an answer once it's clear that an answer may not be coming.

### Applicability

The first of the *Fallacies of Distributed Computing* is that “the network is reliable”, and it’s first for a reason. Switches fail, routers and firewalls get misconfigured; packets get blackholed. Even if your network is working perfectly, not every service is thoughtful enough to guarantee a meaningful and timely response — or any response at all — if and when it malfunctions.

Timeout represents a common solution to this dilemma, and is so beautifully simple that it barely even qualifies as a pattern at all: given a service request or function call that's running for a longer-than-expected time, the caller simply... stops waiting.

However, don't mistake "simple" or "common" for "useless". On the contrary, the ubiquity of the timeout strategy is a testament to its usefulness. The judicious use of timeouts can provide a degree of fault isolation, preventing cascading failures and reducing the chance that a problem in a downstream resource becomes *your* problem.

## Participants

### *Client*

The client who wants to execute SlowFunction.

### *SlowFunction*

The long-running function that implements the functionality desired by Client.

### *Timeout*

A wrapper function around SlowFunction that implements the timeout logic.

## Implementation

There are several ways to implement a timeout in Go, but the idiomatic way is to use the functionality provided by the context package. See "[The Context Package](#)" for more information.

In an ideal world, your possibly long-running function will accept a context.Context parameter directly. If so, your work is fairly straightforward: you need only pass it a Context value decorated with the context.WithTimeout function:

```
ctx := context.Background()
ctxt, cancel := context.WithTimeout(ctx, 10 * time.Second)
defer cancel()

result, err := SomeFunction(ctxt)
```

However, this isn't always the case, and with 3rd party libraries you don't always have the option of refactoring to accept a Context value. In these cases, the best course of action may be to wrap the function call in such a way that it *does* respect your Context.

In the below sample code, we do exactly that. In our implementation, we have a slow function with the following signature:

```
func SlowFunction(string) (string, error)
```

Unfortunately, SlowFunction comes from a 3rd party package and doesn't accept a Context value. If Client were to call SlowFunction directly it would be forced to wait until the function completes, if indeed it ever does. Now what?

Instead of calling SlowFunction directly, we provide the Timeout function, which accepts the same parameters as SlowFunction, but returns a closure that accepts a Context value and whose return list matches SlowFunction's.

Now, Client can set whatever timeout she wants — if any — by using the context.WithTimeout function to decorate a new or existing Context value, and have that Context respected.

## Sample Code

The following example imagines the existence of the fictional `gitplace.io/random-repo/random-package` package, which provides the `SlowFunction` function. Imagine that `SlowFunction` provides some functionality that may or may not complete in some reasonable time.

Rather than calling `SlowFunction` directly, we provide a `Timeout` function, which runs `SlowFunction` in a goroutine. Importantly, it also creates a channel for each element in the return list, so that when and if `SlowFunction` completes, it will send the return values into the channels.

Finally, `Timeout` returns a closure that contains a `select` block on two channels: the first of the `SlowFunction` response channels, and the `Context` value's `Done` channel. If the former completes first, the closure will return the `SlowFunction` return values; otherwise it returns the error provided by the `Context`.

```
import (
    rando "gitplace.io/random-repo/random-package"
)

func Timeout(arg string) func(context.Context) (string, error) {
    chres := make(chan string)
    cherr := make(chan error)

    go func() {
        res, err := rando.SlowFunction(arg)
        chres <- res
        cherr <- err
    }()

    return func(ctx context.Context) (string, error) {
        select {
        case res := <-chres:
            return res, <-cherr
        case <-ctx.Done():
            return "", ctx.Err()
        }
    }
}
```

Using the above `Timeout` function isn't much more complicated than consuming `SlowFunction` directly, except that instead of one function call, we have two: the call to `Timeout` to retrieve the closure, and the call to the closure itself.

```

func main() {
    ctx := context.Background()
    ctxt, cancel := context.WithTimeout(ctx, 2 * time.Second)
    defer cancel()

    timeout := Timeout("some input value")
    res, err := timeout(ctxt)

    fmt.Println(res, err)
}

```

Finally, although it's usually preferred to implement service timeouts using `context.Context`, channel timeouts *can* also be implemented using the channel provided by the `time.After` function. See “[Implementing Channel Timeouts](#)” for an example of how this can be done.

## Concurrency Patterns

A cloud native service will often be called upon to efficiently juggle multiple processes and handle high (and highly variable) levels of load, ideally without having to suffer the trouble and expense of scaling up. As such, it needs to be highly concurrent and able to manage multiple simultaneous requests from multiple clients. While Go is known for its concurrency support, bottlenecks can and do happen. Some of the patterns that have been developed to prevent them are presented here.

### Fan-In

Fan-In multiplexes multiple input channels onto one output channel.

#### Applicability

Services that have some number of workers that all generate output may find it useful to combine all of the workers' outputs to be processed as a single unified stream. For these scenarios we use

the Fan-In pattern, which can read from multiple input channels by multiplexing them onto a single destination channel.

## Participants

### *Sources*

A set of one or more input channels with the same type.  
Accepted by Funnel.

### *Destination*

An output channel of the same type as Sources. Created and provided by Funnel.

### *Funnel*

Accepts Sources and immediately returns Destination. Any input from any Sources will be output by Destination.

## Implementation

The Funnel function is a variadic function that receives sources: zero to N channels of some type (int in the example below). For each channel in sources, Funnel starts a separate goroutine that reads values from its assigned source and forwards them to dest, a single output channel shared by all of the goroutines.

Note the use of a `sync.WaitGroup`, which is used in a separate goroutine to close the dest channel after all of the sources are closed.

## Sample Code

```
func Funnel(sources ...chan int) chan int {
    dest := make(chan int) // The shared output
    channel

    var wg sync.WaitGroup // Used to automatically
    close dest
```



```

        defer close(ch)          // Close ch when the
routine ends

    for i := 1; i <= 5; i++ {
        ch <- i
        time.Sleep(time.Second)
    }
}()

}

dest := Funnel(sources...)
for d := range dest {
    fmt.Println(d)
}

fmt.Println("Done")           // Prints only when dest
is closed
}

```

## Fan-Out

Fan-Out evenly distributes messages from an input channel to multiple output channels.

### Applicability

Fan-Out receives messages from an input channel, distributing them evenly among output channels, and is a useful pattern for parallelizing CPU and I/O utilization.

For example, imagine that you have an input source, such as a Reader on an input stream, or a listener on a message broker, that provides the inputs for some resource-intensive unit of work. Rather than coupling the input and computation processes, which would confine the effort to a single serial process, you might prefer to parallelize the workload by distributing it among some number of concurrent worker processes.

### Participants

#### Source

An input channel. Accepted by Split.

### *Destinations*

An output channel of the same type as Source. Created and provided by Split.

### *Split*

A function that accepts Source and immediately returns Destinations. Any input from Source will be output to a Destination.

## Implementation

Fan-Out may be relatively conceptually straight-forward, but the devil is in the details.

Typically, Fan-Out is implemented as a Split function, which accepts a single Source channel and integer representing the desired number of Destination channels. The Split function creates the Destination channels and executes some background process that retrieves values from Source channel and forwards them to one of the Destinations.

The implementation of the forwarding logic can be done in one of two ways:

- Using a single goroutine that reads values from Source and forwards them to the Destinations in a round-robin fashion. This has the virtue of requiring only one master goroutine, but if the next channel isn't ready to read yet it'll slow the entire process.
- Using separate goroutines for each Destination that compete to read the next value from Source and forward it to their respective Destination. This requires slightly more resources,

but is less likely to get bogged down by a single slow-running worker.

The example below uses the latter approach.

## Sample Code

In this example, the `Split` function accepts a single receive-only channel, `source`, and an integer describing the number of channels to split the input into, `n`. It returns a slice of `n` send-only channels with the same type as `source`.

Internally, `Split` creates the destination channels. For each channel created, it executes a goroutine that retrieves values from `source` in a `for` loop and forwards them to their assigned output channel.

Effectively, each goroutine competes for reads from `Source`; if several are trying to read, the “winner” will be randomly determined. If `source` is closed, all goroutines terminate and all of the destination channels are closed.

```
func Split(source <-chan int, n int) []<-chan int {
    dests := make([]<-chan int, 0)           // Create the dests slice

    for i := 0; i < n; i++ {                  // Create n destination
        channels
        ch := make(chan int)
        dests = append(dests, ch)

        go func() {                         // Each channel gets a
            dedicated
            defer close(ch)                // goroutine that competes
            for reads
                for val := range source {
                    ch <- val
                }
            }()
        }

    return dests
}
```

Given a channel of some specific type, the `Split` function will return a number of destination channels. Typically each will be passed to a separate goroutine as demonstrated in the following toy example:

```
func main() {
    source := make(chan int)                      // The input channel
    dests := Split(source, 5)                      // Retrieve 5 output
    channels

    go func() {                                     // Send the number 0..10
        to source
        for i := 1; i <= 10; i++ {                  // and close it when we're
            done
            source <- i
        }

        close(source)
    }()

    var wg sync.WaitGroup                         // Use WaitGroup to wait
    until
    wg.Add(len(dests))                          // the output channels all
    close

    for i, ch := range dests {
        go func(i int, d <-chan int) {
            defer wg.Done()

            for val := range d {
                fmt.Printf("#%d got %d\n", i, val)
            }
        }(i, ch)
    }

    wg.Wait()
}
```

## Future

Future provides a placeholder for a value that's not yet known.

## Applicability

Futures (also known as Promises or Delays<sup>4</sup>) are a synchronization construct that provide a placeholder for some value that's still being generated by an asynchronous process.

This pattern isn't used as frequently in Go as in some other languages because channels can be often used in a similar way. For example:

```
func InverseProduct(a, b Matrix) Matrix {
    inva := make(chan Matrix)
    invb := make(chan Matrix)

    go func() { inva <- Inverse(a) }()
    go func() { invb <- Inverse(b) }()

    return Product(<-inva, <-invb)
}

func Inverse(m Matrix) <-chan Matrix {
    out := make(chan Matrix)

    go func() {
        out <- BlockingInverse(m)
        close(out)
    }()
}

return out
}
```

When using channels in this way, functions with more than one return value will often be assigned a dedicated channel for each member of the return list, which can become awkward as the return list grows or when the values need to be read by more than one goroutine.

Future contains this complexity by encapsulating it in an API that provides the consumer with a simple interface whose method can be called normally, blocking all calling routines until all of its results are resolved. The interface that the value satisfies doesn't have to be

constructed specially for this purpose; any interface that's convenient for the consumer can be used.

## Participants

### *Future*

The interface that is received by the consumer to retrieve the eventual result.

### *SlowFunction*

A wrapper function around some function to be asynchronously executed; provides *Future*.

### *InnerFuture*

Satisfies the *Future* interface; includes an attached method that contains the result access logic.

## Implementation

The API presented to the consumer is fairly straight-forward: the programmer calls *SlowFunction*, which returns a value that satisfies the *Future* interface. *Future* may be a bespoke interface, as in the example below, or it may be something more like an `io.Reader` that can be passed to its own functions.

In actuality, when *SlowFunction* is called it executes the core function of interest as a goroutine. In doing so, it defines channels to capture the core function's output, which it wraps in *InnerFuture*.

*InnerFuture* has one or more methods that satisfy the *Future* interface, which retrieve the values returned by the core function from the channels, cache them, and return them. If the values aren't available on the channel, the request blocks; if they have already been retrieved, the cached values are returned.

## Sample Code

In this example, we use a Future interface that the InnerFuture will satisfy.

```
type Future interface {
    Result() (string, error)
}
```

The InnerFuture struct is used internally to provide the concurrent functionality. In this example, it satisfies the Future interface, but could just as easily choose to satisfy something like io.Reader by attaching a Read method, for example.

In this implementation, the struct itself contains a channel and a variable for each value returned by the Result method. When Result is first called it attempts to read the results from the channels and set them back to the InnerFuture struct so that subsequent calls to Result() can immediately return the cached values.

Note the use of sync.Once and sync.WaitGroup. The former does what it says on the tin: it ensures that the function that's passed to it is called exactly once. The WaitGroup is used to make this function call thread safe: any calls after the first will be blocked at wg.Wait() until the channel reads are complete.

```
type InnerFuture struct {
    once sync.Once
    wg   sync.WaitGroup

    res  string
    err  error
    resCh <-chan string
    errCh <-chan error
}

func (f *InnerFuture) Result() (string, error) {
    f.once.Do(func() {
        f.wg.Add(1)
        defer f.wg.Done()
        f.res = <-f.resCh
        f.err = <-f.errCh
    })
}
```

```

    })

    f.wg.Wait()

    return f.res, f.err
}

```

`SlowFunction` is a wrapper around the core functionality that you want to run concurrently. It has the job of creating the results channels, running the core function in a goroutine, and creating and returning the `Future` implementation (`InnerFuture`, in this example).

```

func SlowFunction(ctx context.Context) Future {
    resCh := make(chan string)
    errCh := make(chan error)

    go func() {
        select {
        case <-time.After(time.Second * 2):
            resCh <- "I slept for 2 seconds"
            errCh <- nil
        case <-ctx.Done():
            resCh <- ""
            errCh <- ctx.Err()
        }
    }()

    return &InnerFuture{resCh: resCh, errCh: errCh}
}

```

To make use of this pattern, you need only call the `SlowFunction` and use the returned `Future` as you would any other value.

```

func main() {
    ctx := context.Background()
    future := SlowFunction(ctx)

    res, err := future.Result()
    if err != nil {
        fmt.Println("error:", err)
        return
    }
}

```

```
    }  
}
```

## Sharding

Sharding splits a large data structure into multiple partitions to localize the effects of read/write locks.

### Applicability

The term “sharding” is typically used in the context of distributed state to describe data that is partitioned between server instances. This kind of *horizontal sharding* is commonly used by databases and other data stores to distribute load and provide redundancy. We’ll discuss horizontal sharding across instances in more detail in [Link to Come].

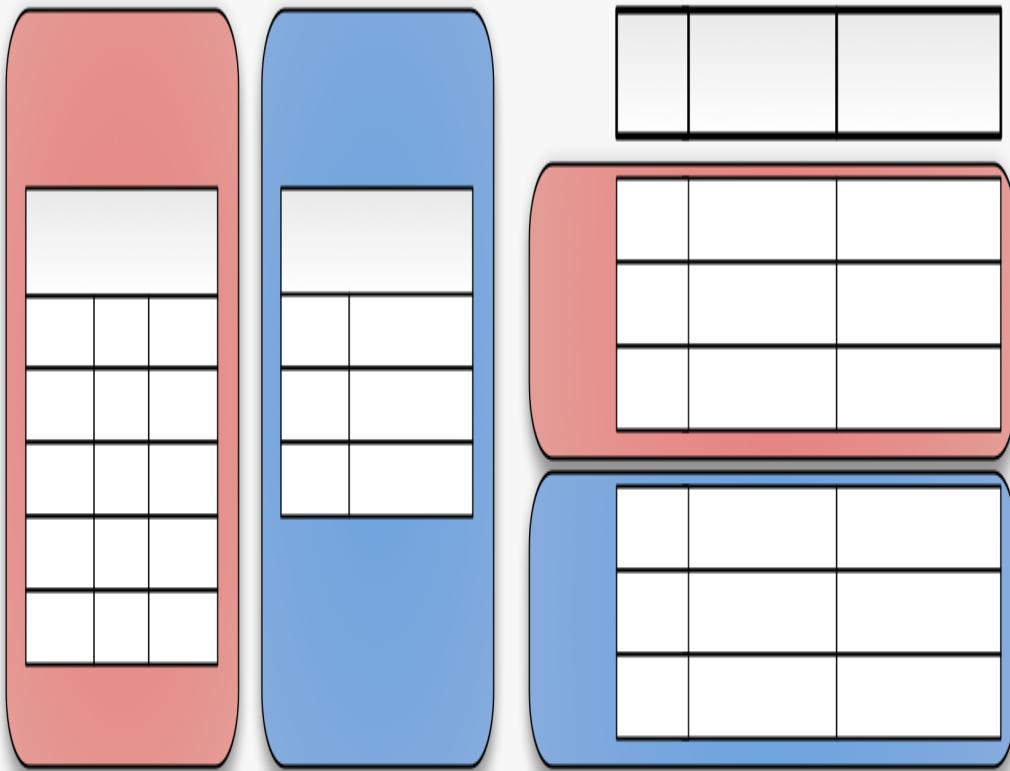
A slightly different issue can sometimes affect highly-concurrent services that have a shared data structure with a locking mechanism to protect it from conflicting writes. In this scenario, the locks that serve to ensure the fidelity of the data can also create a bottleneck when processes start to spend more time waiting for locks than they do doing their jobs. This unfortunate phenomenon is called *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 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.

## HORIZONTAL VS. VERTICAL SHARDING

Large data structures can be *sharded*, or partitioned, in two different ways:

- *Horizontal sharding* is the partitioning of data across service instances. This can provide data redundancy and allow load to be balanced between instances, but also adds the latency and complexity that comes with distributed data.
- *Vertical sharding* is the partitioning of data within a single instance. This can reduce read/write contention between concurrent processes, but also doesn't scale or provide any redundancy.



Vertical

Horizontal

## Participants

### *ShardedMap*

An abstraction around one or more Shards providing read and write access as if the Shards were a single map.

## *Shard*

An individually-lockable collection representing a single data partition.

## Implementation

While idiomatic Go strongly prefers the use of memory sharing via channels over using locks to protect shared resources<sup>5</sup>, this isn't always possible. Maps are particularly unsafe for concurrent use, making the use of locks as a synchronization mechanism a necessary evil. Fortunately, Go provides `sync.RWMutex` for precisely this purpose.

RWMutex provides methods to establish both read and write locks, as demonstrated below. Using this method, any number of processes can establish simultaneous read locks as long as there are no open write locks; a process can establish a write lock only when there are no existing read or write locks. Attempts to establish additional locks will block until any locks ahead of it are released.

```
var items = struct{  
    map and a  
    sync.RWMutex  
    sync.RWMutex  
    m map[string]int  
}{m: make(map[string]int)}  
  
func ThreadSafeRead(key string) int {  
    items.RLock()  
    lock  
    value := items.m[key]  
    items.RUnlock()  
    lock  
    return value  
}  
  
func ThreadSafeWrite(key string, value int) {  
    items.Lock()  
    lock
```

```
    items.m[key] = value
    items.Unlock()                                // Release write
  lock
}
```

This strategy generally works perfectly fine. However, because locks allow access to only one process at a time, the average amount of time spent waiting for locks to clear in a read/write intensive application can increase dramatically with the number of concurrent processes acting on the resource. The resulting lock contention can potentially bottleneck key functionality.

Vertical sharding reduces lock contention by splitting the underlying data structure — usually a map — into several individually lockable maps. An abstraction layer provides access to the underlying shards as if they were a single structure.

Internally, this is accomplished by creating an abstraction layer around what is essentially a map of maps. Whenever a value is read or written to the map abstraction, a hash value is calculated for the key, which is then modded by the number of shards to generate a shard index. This allows the map abstraction to isolate the necessary locking to only the shard at that index.

Key	Key Hash	Value
A	1	"Alpha"
B	2	"Beta"
G	3	"Gamma"
D	4	"Delta"
E	5	"Epsilon"
Z	6	"Zeta"
E	7	"Eta"
Th	8	"Theta"



Key	Key Hash	Value
A	1	"Alpha"
G	3	"Gamma"
E	5	"Epsilon"
E	7	"Eta"

Key	Key Hash	Value
B	2	"Beta"
D	4	"Delta"
Z	6	"Zeta"
Th	8	"Theta"

Figure 4-2. Vertically sharding a map by key hash.

## Sample Code

In the below example, we use the standard sync and crypto/sha1 packages to implement a basic sharded map: ShardedMap.

Internally, ShardedMap is just a slice of pointers to some number of Shard values, but we define it as a type so we can attach methods to it. Each Shard includes a `map[string]interface{}` that contains that shard's data and a composed sync.RWMutex so that it can be individually locked.

```
type Shard struct {
    sync.RWMutex           // Compose from
    sync.RWMutex
    m map[string]interface{} // m contains the shard's
    data
}

type ShardedMap []*Shard           // ShardedMap is a *Shards
slice
```

Go doesn't have any concept of constructors, so we provide a NewShardedMap function to retrieve a new ShardedMap.

```
func NewShardedMap(nshards int) ShardedMap {
    shards := make([]*Shard, nshards)      // Initialize a *Shards
    slice

    for i := 0; i < nshards; i++ {
        shard := make(map[string]interface{})
        shards[i] = &Shard{m: shard}
    }

    return shards                         // A ShardedMap IS a
    *Shards slice!
}
```

ShardedMap has two unexported methods, `getShardIndex` and `getShard`, which are used to calculate a key's shard index and retrieve a key's correct shard, respectively. These could be easily combined into a single method, but slitting them this way makes them easier to test.

```

func (m ShardedMap) getShardIndex(key string) int {
    checksum := sha1.Sum([]byte(key))           // Use Sum from
    "crypto/sha1"
    hash := int(checksum[17])                   // Pick a random byte as
    our hash
    return hash % len(shards)                  // Mod by len(shards) to
    get index
}

func (m ShardedMap) getShard(key string) *Shard {
    index := m.getShardIndex(key)
    return m[index]
}

```

Note that the above example has an obvious weakness: because it's effectively using a byte-sized value as the hash value, it can only handle up to 255 shards. If for some reason you want more than that, you can sprinkle some binary arithmetic on it: `hash := int(sum[13]) << 8 | int(sum[17])`.

Finally, we add methods to `ShardedMap` to allow a user to read and write values. Obviously these don't demonstrate all of the functionality a map might need. The source for this example is in the GitHub repository associated with this book, however, so please feel free to implement them as an exercise. A `Delete` and a `Contains` method would be nice.

```

func (m ShardedMap) Get(key string) interface{} {
    shard := m.getShard(key)
    shard.RLock()
    defer shard.RUnlock()

    return shard.m[key]
}

func (m ShardedMap) Set(key string, value interface{}) {
    shard := m.getShard(key)
    shard.Lock()
    defer shard.Unlock()
}

```

```
    shard.m[key] = value  
}
```

When you do need to establish locks on all of the tables, it's generally best to do so concurrently. Below we implement a Keys function using goroutines and our old friend sync.WaitGroup.

```
func (m ShardedMap) Keys() []string {  
    keys := make([]string, 0) // Create an empty keys slice  
  
    wg := sync.WaitGroup{} // Create a wait group and add a slice  
    wg.Add(len(m)) // wait value for each slice  
  
    for _, shard := range m { // Run a goroutine for each slice  
        go func(s *Shard) {  
            s.RLock() // Establish a read lock on s  
  
            for key, _ := range s.m { // Get the slice's keys  
                keys = append(keys, key)  
            }  
  
            s.RUnlock() // Release the read lock  
            wg.Done() // Tell the WaitGroup it's done  
        }(shard)  
    }  
  
    wg.Wait() // Block until all reads are done  
  
    return keys // Return combined keys
```

Using ShardedMap isn't quite like using a standard map unfortunately, but while it's different, it's no more complicated.

```

func main() {
    shardedMap := NewShardedMap(5)

    shardedMap.Set("alpha", 1)
    shardedMap.Set("beta", 2)
    shardedMap.Set("gamma", 3)

    fmt.Println(shardedMap.Get("alpha"))
    fmt.Println(shardedMap.Get("beta"))
    fmt.Println(shardedMap.Get("gamma"))

    keys := shardedMap.Keys()
    for _, k := range keys {
        fmt.Println(k)
    }
}

```

- 1 Spoken August 1979. Attested to by Vicki Almstrum, Tony Hoare, Niklaus Wirth, Wim Feijen, and Rajeev Joshi. *In Pursuit of Simplicity: A Symposium Honoring Professor Edsger Wybe Dijkstra*, 12-13 May 2000.
- 2 L (yes, his legal name is L) is a brilliant and fascinating human being. Look him up some time.
- 3 Wikipedia contributors. “Token bucket”. *Wikipedia, The Free Encyclopedia*, 5 Jun. 2019.
- 4 While these terms are often used interchangeably, they can also have shades of meaning depending on their context. I know. Please don't write me any angry letters about this.
- 5 See the article “Share Memory By Communicating”, on The Go Blog

# Chapter 5. Building a Cloud Native Service

---

## A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 5th chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at [ablevins@oreilly.com](mailto:ablevins@oreilly.com).

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

In this chapter, our real work finally begins.

In it, we’ll weave together many of the materials discussed throughout **Part II** 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.

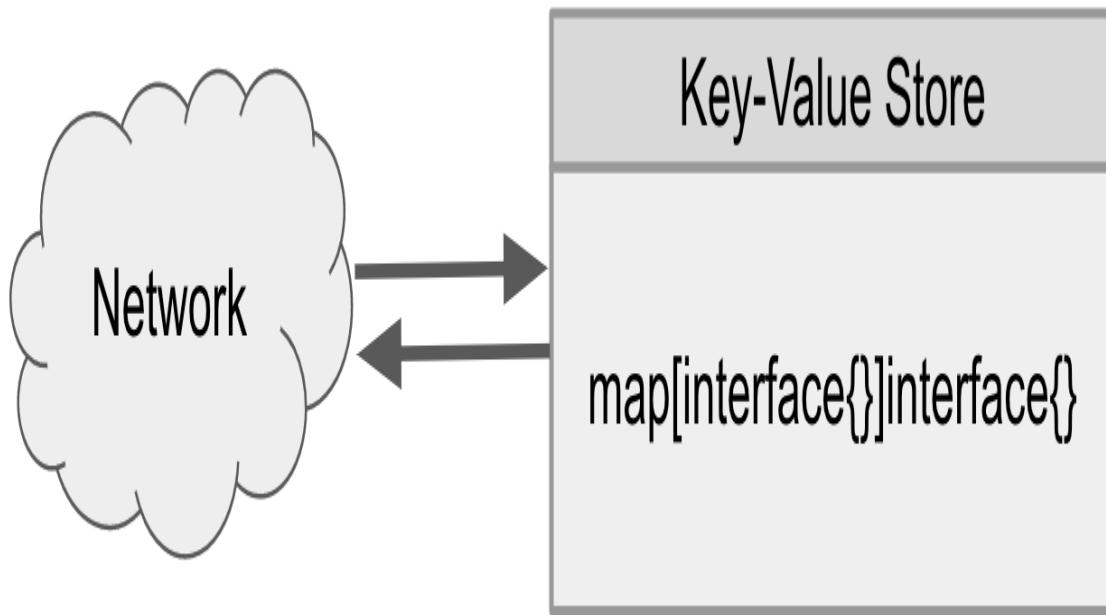
It should be conceptually simple, straight-forward enough to implement in its most basic form, but non-trivial and amenable to scaling and distributing. Something that we can iteratively refine over the remainder of the book. I put a lot of thought into this, considering different ideas for what our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

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

## What's a Key-Value Store?

A key-value store is a kind of non-relational 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<sup>2</sup>. 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. 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 build a simple, non-distributed 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 our 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.

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 more PUT the second time<sup>3</sup>. The operation  $x+=1$ , however, is not idempotent, because every time it's called produces a new state.

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<sup>4</sup>? 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 non-idempotent 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 *synonym* for "cloud native"<sup>5</sup>. 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 Eventual Goal

The above requirements are quite a lot to chew on, but they represent the absolute minimum for our key-value store to be usable. In later chapters 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

to 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.

## Our 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 it says on the tin: 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 Put returns an error, it doesn't use errors.New? Instead it returns the pre-built ErrorKeyExists error value. But why? This allows the consuming service to determine

exactly what type of error it's receiving and to respond accordingly. For example, it may 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 RPC, which is supported by the standard `net/rpc` package, but it requires additional overhead for the client, and again our data just isn't complex enough to warrant it.

That leaves us with 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 the 80% use case. Even better: they're designed to be extensible, so there are a number of Go web frameworks in existence.

For now though, 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 above 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. We 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 we've registered our handlers, we 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 above example is a complete program that can be compiled and run using `go run`.

```
$ go run corehttp.go
```

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 below:

```
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 fine. However, sometimes you’ll need the additional functionality provided by a 3rd party web toolkit. A popular choice is Gorilla<sup>6</sup>, which while being relatively new and less fully-developed and resource-rich as 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'll come in handy going forward. If you're curious and/or impatient, however, you can take a look at <https://www.gorillatoolkit.org/pkg/mux> for more information.

Like all 3rd party packages, to use it you'll need to use go get to install it:

```
go get github.com/gorilla/mux
```

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))
}
```

Like before, this can be compiled and run with `go run`:

```
$ go run gorillamux.go
```

Calling the endpoint with `curl` or browsing to it with a browser will give us the expected response.

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

## Variables in URI Paths

The Gorilla web toolkit provides a wealth of additional functionality over the standard `net/http` package, but one feature in particular interests us 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 may 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 refer 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, including but not 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 our RESTful Service

Now that we know how to use Go's standard HTTP library, we're going to use it to create a RESTful service that a client can interact with to execute call to the API we built in "[Our Super Simple API](#)". Once we've done this we'll have implemented the absolute minimal viable key-value store.

### Our 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](#) below.

The URI for our 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. Our 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 the section “[Variables in URI Paths](#)” we discussed how to use the `gorilla/mux` package to register paths that contain variable segments, which will allow us to define a single variable path that handles *all* keys, mercifully freeing us from having to register every key independently. Next, in “[So Many Matchers](#)” we discussed how to use route matchers to direct requests to specific handler functions based on various non-path criteria, which we’ll use to create a separate handler function for each of the five HTTP methods that we’ll be supporting.

## Implementing our Create Function

Okay, we now have everything we 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 “Our Super Simple API”.
- 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 above requirements are implemented in the below 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 := ioutil.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 we have our “key-value create” handler function, we can register it with our 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 we have our service put together, we can run it using go run . in the usual way. Let’s do that now, and send some requests to it to see how it responds.

First, we’ll 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 output below. The actual output was quite wordy, so we’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,

gives details about the server response. The actual output was quite wordy, so we've selected the relevant bits for readability.

In this output we see that we 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 we hit the /v1/key-a endpoint with an unsupported GET method? Assuming that our matcher function is working correctly, we 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 our Read Function

Now that our service has a a fully-functioning Put method, it sure would be nice if we could read our 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 “Our Super Simple API”.
- 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 above requirements are implemented in the below 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 err != nil {                                    // Unexpected error!
        http.Error(w,
            err.Error(),
            http.StatusInternalServerError)
        return
    }

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

And now that we have the “get” handler function, we 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 our 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 we can get our values back, we're able to test for idempotence as well. Let's repeat the requests and make sure that we 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!
```

We do! But what if we want to overwrite the key with a new value? Will the subsequent GET have the new value? We can test that by changing the value sent by our `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 our new value.

Finally, to complete our method set we'll just need to create a handler for the DELETE method. We'll leave that as an exercise for the reader, though. Enjoy!

# Making Our Data Structure Concurrency-Safe

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

So what do we do? Well, typically when a programmer has a data structure that she needs to read from and write to from concurrently executing goroutines she'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, we don't need to implement this ourselves<sup>7</sup>: Go's sync package provides exactly what we need in the form of sync.RWMutex. The statement below uses the magic of composition to create an anonymous struct that contains our 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 us to use the Lock method to take the write lock when we 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, we 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 Our Application

Now that we know how to use a sync.RWMutex to implement a basic read-write mutex, we go back and working it back into the code we created for “Our Super Simple API”.

First, we’ll want to refactor the store map<sup>8</sup>. We can construct it like myMap above, as an anonymous struct that contains our map and an embedded sync.RWMutex.

```

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

```

Now that we have our store struct, let’s 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.

### WARNING

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.

As we'll discuss in [Link to Come], there are 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](#)” we'll use a file-based *transaction log* to maintain a record of every time a resource is modified. If our service crashes, is restarted, or otherwise finds itself in an inconsistent state, a transaction

log allows us to reconstruct the service's complete resource set simply by replaying the transactions.

- In “[Storing State in an External Database](#)” we'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 we're building, but externalizing data into another service designed specifically for that purpose is a common means sharing state between service replicas and providing resilience.

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

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

## APPLICATION STATE VS 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](#)”.

## 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 our service crashes, is restarted, or otherwise finds itself in an inconsistent state, a transaction log allows us to reconstruct the service's complete resource set simply by replaying the transactions. 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 straight-forward.

## Our Transaction Log Format

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

We will assume that our transaction log will be read only when our service is restarted or otherwise needs to recover its state, and that it will be read from top to bottom, sequentially replaying each event. It follows that our 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 our key-value store, for example, a "delete" is simply recorded on the log.

Given everything above, 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; 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.

## Our 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)
}
```

We'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 our first implementation and add additional methods to our 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. If our service crashes, is restarted, or otherwise finds itself in an inconsistent state, the transaction log allows us to reconstruct the service's complete resource set simply by replaying the transactions.

Transaction log files 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 straight-forward.

This file-based implementation has some tempting pros, but some pretty significant cons as well:

Pros:

- **No upstream dependency** — There's no dependency on an external service that could fail or that we can lose access to.
- **Technically straight-forward** — The logic isn't especially sophisticated. We can be up and running quickly.

Cons:

- **Harder to scale** — We'll need some additional way to distribute our state between nodes when we want to scale.
- **Infinite growth** — These logs have to be stored on disk; we can't let them grow forever. We'll need some way of compacting them.

## Prototyping Our Transaction Logger

Before we get to the code, let's make some design decisions. First, for simplicity, our 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 “[Our Transaction Log Format](#)”, 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; 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, we'll go ahead and define a `FileTransactionLogger` type, which will implicitly implement the `TransactionLogger` interface described in “[Our Transaction](#)

Logger Interface” 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 in a bit.

## Defining The Event Type

Thinking ahead, we probably want the above WritePut and WriteDelete methods to operate asynchronously. Perhaps we can implement that using some kind of events channel that they could write to so that some concurrent routine could retrieve from and perform the writes. That sounds like a nice idea, but if we’re going to do it we’ll need some kind of internal representation of an “event”.

That shouldn’t give us too much trouble. Incorporating all of the fields that we listed in “Our Transaction Log Format” gives us something like the Event struct, below.

```
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 straight-forward, 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 below 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. We can easily 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 below with the numbers of bytes in various digital units.

```
type ByteSize uint64

const (
    _          = iota           // iota == 0; ignore the
                                // implicit repetition
```

```

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

```

Using the `iota` technique, we don't have to manually assign values to constants. Instead, we 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.

We now have an idea of what our `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 Our `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?

We'll probably want to keep track of the physical location of the transaction log, so it makes sense to have an `os.File` attribute representing that. We'll also need to remember the last sequence number that we assigned so we can correctly set it for each event; we can keep that as an unsigned 64-bit integer attribute. That's great, but how will our `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 IO 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
}
```

Now we have our `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 we might have 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're already discussed that writes to our 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 their errors instead.

## Creating a New `FileTransactionLogger`

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

## WARNING

See how `NewFileTransactionLogger` returns a pointer type, but its return list specifies the decidedly non-pointy `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`: Open the file in read-write mode,
- `os.O_APPEND`: Writes to this file will append, not overwrite, and
- `os.O_CREATE`: If the file doesn't exist, create it.

There are quite a few of these flags besides the three we use here.

Take a look at the `os` package documentation at

<https://golang.org/pkg/os> 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* have `NewFileTransactionLogger` create the channels and spawn a goroutine, but that feels like we'd be adding too much mysterious functionality. Instead, we'll create a `Run` method.

## 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.

We're going to change this by creating a `Run` method, shown below:

```

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
            }
        }()
    }
}

```

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 allows the consuming service to handle shorts burst of events without being slowed by disk IO. If the buffer does fill up then the write methods will block until the log writing goroutine catches up.

Second, it creates a 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 non-blocking 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<sup>10</sup>. But how do we do that?

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

Channels, our dependable friend, will let us stream the results to a consumer as we retrieve 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, however, make it almost trivially easy to stream the data to the consumer in a much more space and memory efficient way.

The `ReadEvents` method<sup>11</sup> below 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 events  
channel  
    outError := make(chan error, 1)               // A buffered errors  
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()

        fmt.Sscanf(
            line, "%d\t%d\t%s\t%s",
            &e.Sequence, &e.EventType, &e.Key, &e.Value)

        // Sanity check! Are the sequence numbers in increasing
        order?
        if l.lastSequence >= e.Sequence {
            outError <- fmt.Errorf("transaction numbers out of
sequence")
            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 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 we wanted to we could actually use it to retrieve our data. The bufio package, however, give

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` — `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 our 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`.

The `fmt.Sscanf` method gives us 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 string (`%s`), separated by tab characters. Conveniently, `fmt.Sscanf` lets us pass in pointers to the target values for each verb, which it can update directly<sup>12</sup>.

#### TIP

Go's format verbs 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 but not limited to C++, Java, Perl, PHP, Ruby, and Scala — so you may already be familiar with them. If you're not, I suggest that you take a break now to look over `fmt` package documentation: <https://golang.org/pkg/fmt/>.

At the end of each loop we update our last used sequence number to the value we just read and send the event on its merry way. A minor point: note how we reuse the same `Event` value 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 creates copies of whatever value we send into it.

Finally, we check 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.

## Our Transaction Logger Interface (Redux)

Now that we've implemented a fully-functional FileTransactionLogger, it's time to look back and see which of our new methods we'd like to incorporate into the 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 we've settled that, we're able to finally start integrating the transaction log into our key-value service.

## Initializing the FileTransactionLogger In Our Web Service

Our FileTransactionLogger is now complete! All that's left to do now is to integrate it with our 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`. We'll call it `transact` because naming is hard:

```
var transact TransactionLogger
```

Now that we have that detail out of the way, we can define our initialization method, which we do below:

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

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

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

    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)
            }
        }
    }

    transact.Run()

    return err
}
```

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

The next part is more interesting: it calls `transact.ReadEvents`, and replays the results based on the `Event` values received from it. We

do this 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 Our Web Service

Now that we have the initialization logic put together, all that's left to do to complete the integration of the `TransactionLogger` is add exactly three function calls into our web service. This is fairly straight-forward, so we won't walk through it here. But briefly, we'll need to add the following:

- `initializeTransactionLog` to the `main` method,
- `transact.WriteDelete` to `keyValueDeleteHandler`, and
- `transact.WritePut` to `keyValuePutHandler`.

We'll leave the actual integration as an exercise for the reader<sup>13</sup>.

## 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. Including:

- 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 whitespace 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 will include its core libraries a generic interface around SQL (or SQL-like) databases.

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? Well, maybe it is and maybe it isn't. In any event, 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 between service replicas and provides data resilience. Besides, the point is to show how we might interact with a database, not to design the perfect application.

In this section, we'll be implementing a transaction log backed by a external database and satisfying the `TransactionLogger` interface, just as we did in “[Storing State in a Transaction Log File](#)”. This would certainly work, and even have some benefits as mentioned above, but it comes with some tradeoffs:

Pros:

- **Externalizes application state** — Less need to worry about distributed state; 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 we 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 it, but if you’re building applications with some kind of data component you’ll 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 the `sql.DB`: Go’s primary database abstraction and entry point for creating statements and transactions, executing queries, and fetching results. While 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 on the Go website (<https://golang.org/s/sqldrivers>).

In the following section we'll be working with a Postgres database, so we'll use the Postgres driver implementation from <https://github.com/lib/pq>.

To load a database driver, we simply anonymously import the package, aliasing its package qualifier to `_` so none of its exported names are visible to our code:

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

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

## Implementing Our 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 below:

```

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 Our FileTransactionLogger](#)” for guidance, it looks like we can create a `PostgresTransactionLogger` using very similar logic.

Starting with the `WritePut`, `WriteDelete`, and `Err` methods, we 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               // Our 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) to PostgresTransactionLogger
- Swapping the `*os.File` for a `*sql.DB`
- Removing `lastSequence`; we can let the database handle the sequencing

## Creating a New PostgresTransactionLogger

That's all good and well, 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 our database, returning an `error` if it fails.

We have 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 we wanted an additional parameter? Do we 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, we create a small helper struct.

```

type PostgresDbParams struct {
    dbName    string
    host     string
    user     string
    password string
}

```

Unlike the big bag of strings approach, the above struct is small, readable, and easily extended. To use it we'd create a `PostgresDbParams` variable and pass it to our construction function. Here's what that looks like:

```

transact, err = NewPostgresTransactionLogger(PostgresDbParams{
    host:      "localhost",
    dbName:   "kvs",
    user:     "test",
    password: "hunter2"
})

```

Our 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 create db value: %w", err)
    }

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

    tl := &PostgresTransactionLogger{db: db}

    exists, err := tl.verifyTableExists()
    if err != nil {
        return nil, fmt.Errorf("failed to verify table exists: %w",

```

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

return tl, nil
}
```

This does quite a few things, but fundamentally isn't 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 we pass to `sql.Open` contains several parameters; the `lib/pq` package supports many more than the ones listed here. See <https://godoc.org/github.com/lib/pq> for a complete listing.

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

Finally, we create our `PostgresTransactionLogger` and use 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 for the reader, you are encouraged to dive into the `database/sql` docs (<https://golang.org/pkg/database/sql/>) and think about how you might go about doing that. If you'd prefer not to though, you can find an implementation in the GitHub repository that comes with this book.

We now have a construction function that establishes a connection to our database and returns our newly-created `TransactionLogger`.

But, once again, we need to get things started. For that, we implement our Run method, which will create the events and errors channels and spawn the event ingestion goroutine.

## Using db.Exec to Execute a SQL INSERT

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

Our PostgresTransactionLogger is very similar. However, instead of appending a line to a file, our new logger uses db.Exec to execute an 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,
                    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 transactions table. The numbered args (\$1, \$2, \$3) in the query are placeholders 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](#)” we used a bufio.Scanner to read previously written transaction logs entries.

The Postgres implementation won’t be *quite* as straight-forward, but the principle is the same: we point at the top of our data source and read until we 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"

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

        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 the contains our 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 our `e` to the output channel.
- `Err` returns the error, if any, that may have caused `rows.Next` to return `false`.

## Initializing the PostgresTransactionLogger In Our Web Service

That's really it; our `PostgresTransactionLogger` is pretty much complete. Now let's go ahead and integrate it into our web service.

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

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

which becomes...

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

Yup. That's it. Really.

Because we built a complete implementation of our `TransactionLogger` interface, everything else stays exactly the same. We 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 our transaction logger, and has lots of room for improvement. Including, but certainly not limited to:

- We assume that the database and table already exist, and will 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.

While there are rich tools and established security best practices for traditional environments, 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

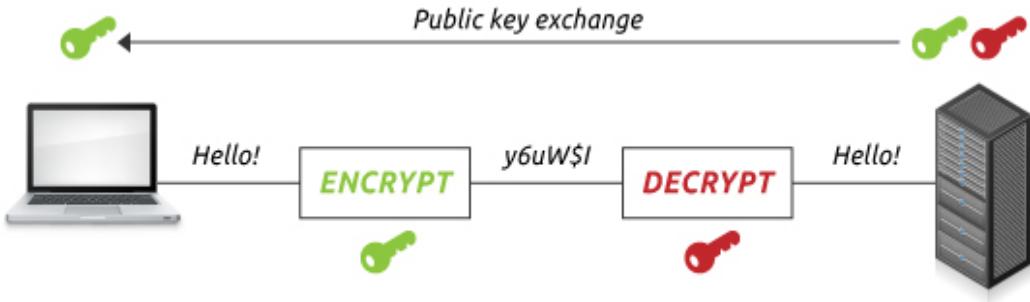
transmitted across a network, opening it up to eavesdropping and manipulation.

The subject of security can take up an entire book of its own<sup>14</sup>, 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 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<sup>15</sup>.



*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<sup>16</sup>.

Instead, every public key is associated with a *digital certificate*, an electronic document used to prove the key’s 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 “trust” 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.

## WARNING

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<sup>17</sup> 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

PEM (Privacy Enhanced Mail) 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<sup>18</sup>.

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 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 key common formats.

## Securing Our 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](#)", you might recall that the `net/http` package contains a function, `ListenAndServe`, which in its most basic form it 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](#)”, 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 Transport Layer Security, 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 our service's communications without a great deal of effort.

## Containerizing Our Key-Value Store

A *container* is a lightweight operating-system level virtualization<sup>19</sup> 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<sup>20</sup>: 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<sup>21</sup>, 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 above, 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.

### *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 taking advantage of them.

Just to name a (very) few<sup>22</sup>. In other words, containerization is super, super useful.

For this book we'll be using Docker to build our container images. Alternative build tools like Bazel 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 use 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 Media). You should read it too.

## 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) from <https://docs.docker.com/v17.09/engine/installation>.

### The Dockerfile

Dockerfiles are essentially build files that describe the steps required to build an image. We demonstrate a very minimal — but complete

— example below.

```
# The parent image. At built 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

# 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?](#)” 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 at <https://docs.docker.com/engine/reference/builder>.

As you may be able to infer, the above example starts with an existing Linux distribution image (Ubuntu 18.04) 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 above example into it.

## Building Your Container Image

Now that we have our simple image, 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 will 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   123da79a90ec  29 seconds ago
152MB
ubuntu          18.04    775349758637  4 weeks ago
64.2MB
```

If all has gone as planned, you'll see at least two images listed: our parent image `ubuntu:18.04`, and our own `my-nginx:latest` image. Next step: running our 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 my own company may be named something like `docker.flatiron.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 (<https://hub.docker.com>).

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.

## Running Your Container Image

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

```
$ docker run --detach --publish 8080:80 --name nginx my-nginx  
4cce9201f484ed2e174a85439ccce3cce47adad9fe082b87f29508bd1d89b6ec
```

This instructs Docker to run a container using our `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 maps a container port to a port on the host.

## Running Your Container Image

To verify that our container is running and is doing what we expect, we 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 output above has been edited for brevity (you may notice that it's missing the COMMAND and CREATED columns), but your own 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?](#)"). 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 id 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 re-use a name, you'll first have to delete the unwanted container.

## Issuing a Request to a Published Container Port

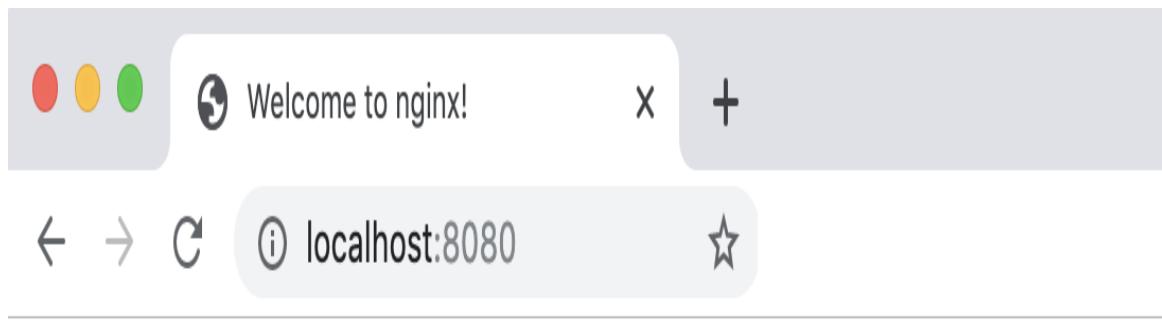
If you've gotten this far then your docker ps output should show a container named nginx that appears to have port 8080 published

and forwarding to the container's port 80. If so, then we're ready to send a request to our running container. But which port will we query?

Well, the Nginx container is listening on port 80. Can we reach that? Actually, no. Importantly, 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
```

We haven't published to port 80, but we *have* published port 8080 and forwarded it to the container's port 80. We 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 below:



# Welcome to nginx!

If you see this page, the nginx web server is successfully installed and working. Further configuration is required.

For online documentation and support please refer to [nginx.org](https://nginx.org).  
Commercial support is available at [nginx.com](https://nginx.com).

*Thank you for using nginx.*

*Figure 5-3. Welcome to nginx!*

## Running Multiple Containers

One of the “killer features” of containerization is that 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 we wanted to run an httpd

container alongside our already-running `my-nginx` container, we 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, we can actually have as many containers as we 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 we want to run the stock `httpd` image, we can run it by using the `docker run` command again, as long as we 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

Great, now you’ve successfully run your container, you’ll probably need to stop and delete it at some point, particularly if you want to re-run 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 the following:

```
$ 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.

#### TIP

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. Go ahead and run `docker ps --all` again. You shouldn't see the container listed at all anymore.

# Building Our Key-Value Store Container

Now that we have the basics down, we 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 493MB `openjdk:14` for Java or the 917MB `python:3.7` for Python<sup>23</sup>, 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 Our Binary To a FROM scratch image

To do this, we'll need our Dockerfile. The example below 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 our service binary.
FROM scratch

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

# 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 one above, 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 our binary to be usable, it has to meet a few criteria:

- It has to be compiled (or cross-compiled) for Linux,

- It has to be statically linked, and
- It has to be named kvs because that's what our Dockerfile is expecting.

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

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

Let's walk through what this does:

- CGO\_ENABLED=0 is a flag that forces cgo off. We won't go into what this is other than that this enforces static linking, but I encourage you to look at <https://golang.org/cmd/cgo> 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
```

Great! Now lets build our image, and see what we get:

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

$ docker images
REPOSITORY      TAG          IMAGE ID          CREATED
SIZE
```

kvs	latest	7b1fb6fa93e3	About a minute ago
7.92MB			
openjdk	14	7c4714f0b8d3	37 hours ago
493MB			
python	3.7	3786ccc05c04	5 days ago
917MB			
node	13	1a77bcb355eb	12 days ago
933MB			

Less than 8MB! That's roughly two orders of magnitude smaller than the relatively massive images for other language's runtimes. This can come in quite handy when you're operating at scale and have regularly pull your image onto a couple hundred nodes.

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

## Iteration 2: Using A Multi-Stage Build

In the last section, we created a simple Dockerfile that would take an existing Linux binary and wrap it in a bare-bones "scratch" image.

But what if we could do the *entire* build — compilation and all — in Docker?

One approach might be to use the `golang` image as our parent image. If we did that, we could write a Dockerfile that compiled our source code and ran the resulting binary at deploy time. This would certainly work, and would have the benefit of being able to build on hosts that don't have the Go compiler installed. However, the resulting image would also be saddled with an additional 803MB of entirely unnecessary build machinery.

Another approach might be to use two Dockerfiles, one for building and generating the binary, and another that containerizes the output of the first build. This is a lot closer to where we want to be, and was actually once the “correct” way to solve this problem. It does, however, require two distinct Dockerfiles that need be sequentially called or managed by a separate script.

A better solution became available with the introduction of multi-stage Docker builds, which allow multiple distinct builds — even with entirely different base images — to be chained together so that artifacts in one stage can be selectively copied into another, leaving behind everything we don't want in the final image. Using this approach we define a build with two-stages: a “compile” stage that generates our binary, and an “image” stage that uses that binary to produce the final image.

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

```
FROM golang:1.13 as build
```

Once we have builds with names, we can use the `COPY` instruction in our Dockerfile to copy *from the previous build*. Our “image” stage might have an instruction like the following, which copies the file

`/go/src/kvs/kvs` from the build stage to the current working directory:

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

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

```
# Stage 1: Compile the binary in a containerized Golang environment
#
FROM golang:1.15 as build

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

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

# Download the dependency code
RUN go get github.com/gorilla/mux

# 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 as image

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

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

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

Now that we have our complete Dockerfile, we can build it in precisely the same way as before. We'll tag it as `multipart` this time though, so that we 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
7.92MB
kvs             multipart b83b9e479ae7  4 minutes ago
7.9MB
```

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

We don't need to stop there though. If you wanted to, you could add other stages as well, such as a `test` stage that ran 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 we *want* our data to outlive our containers.

For example, the ability to mount externally-managed files directly into the filesystem of an otherwise general-purpose container can decouple configuration files from images so you doesn't need to rebuild them whenever you alter a configuration file. 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 resource type — ConfigMap — dedicated to it.

Similarly we might want data generated in the 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<sup>24</sup>, Kubernetes provides a Volume abstraction 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 at <https://kubernetes.io/docs> and equally excellent *Kubernetes: Up and Running* by Brendan Burns, Joe Beda, and Kelsey Hightower (O'Reilly Media).

## Summary

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

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 (Transport Layer Security) as one part of a larger security strategy, and implemented HTTPS (TLS-over-HTTP) 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.

So yes, we covered a lot of things, none of which in any more detail than we needed to introduce and implement it. Everything we touched on could be the subject of books of their own — most actually are — so if you’re sufficiently interested and motivated I encourage you dig in and learn everything you can about any and all of them.

Going forward, we’ll be extending on our key-value service with each subsequent chapter, so stay tuned. Things are about to get even more interesting.

---

<sup>1</sup> Schieber, Philip. “The Wit and Wisdom of Grace Hopper.” *OCLC Newsletter*, March/April, 1987, No. 167.

<sup>2</sup> For some definition of “love”.

<sup>3</sup> If it does, something is very wrong.

<sup>4</sup> Or, like my wife, were only *pretending* not to hear you.

<sup>5</sup> “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)

<sup>6</sup> <https://www.gorillatoolkit.org/>

<sup>7</sup> It’s a good thing too. Mutexes can be pretty tedious to implement correctly!

<sup>8</sup> Didn’t I tell you that we’d make it more complicated?

<sup>9</sup> That’s a lie. There are probably lots of better names.

<sup>10</sup> What makes a transaction log “good” anyway?

- 11 Naming is hard.
- 12 After all this time, I still think that's pretty neat.
- 13 You're welcome.
- 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.
- 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
- 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.
- 22 The initial draft had several more, but this chapter is already pretty lengthy.
- 23 To be fair, these are 244MB and 338MB compressed, respectively.
- 24 I'm intentionally ignoring solutions like Amazon's Elastic Block Store, which can help, but have issues of their own.

# Chapter 6. It's All About Dependability

---

## A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 6th chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at [ablevins@oreilly.com](mailto:ablevins@oreilly.com).

*The most important property of a program is whether it accomplishes the intention of its user<sup>1</sup>.*

—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<sup>2</sup> 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<sup>3</sup> 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* — who decide whether a program is performing correctly. How inconvenient that the intentions a program's user aren't always the same as those of its creator!

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 being 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 straight-forward: 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". Cummins is absolutely brilliant, and has said a lot of absolutely brilliant things<sup>4</sup>, but I have to disagree with her on this one.

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 can go down at night "for maintenance". Users today have come to rely heavily on the services they use, and they expect those services to be always 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.

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*<sup>5</sup>.

All of the patterns and techniques that we associate with "cloud native" — *every single one* — exist to allow services to be deployed, operated, observed, maintained at scale in unreliable environments,

driven by the need to produce dependable services that keep users happy.

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 over their life cycles.

The concept of dependability in a computing context was first rigorously defined by Jean-Claude Laprie about 35 years ago<sup>6</sup>, 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:

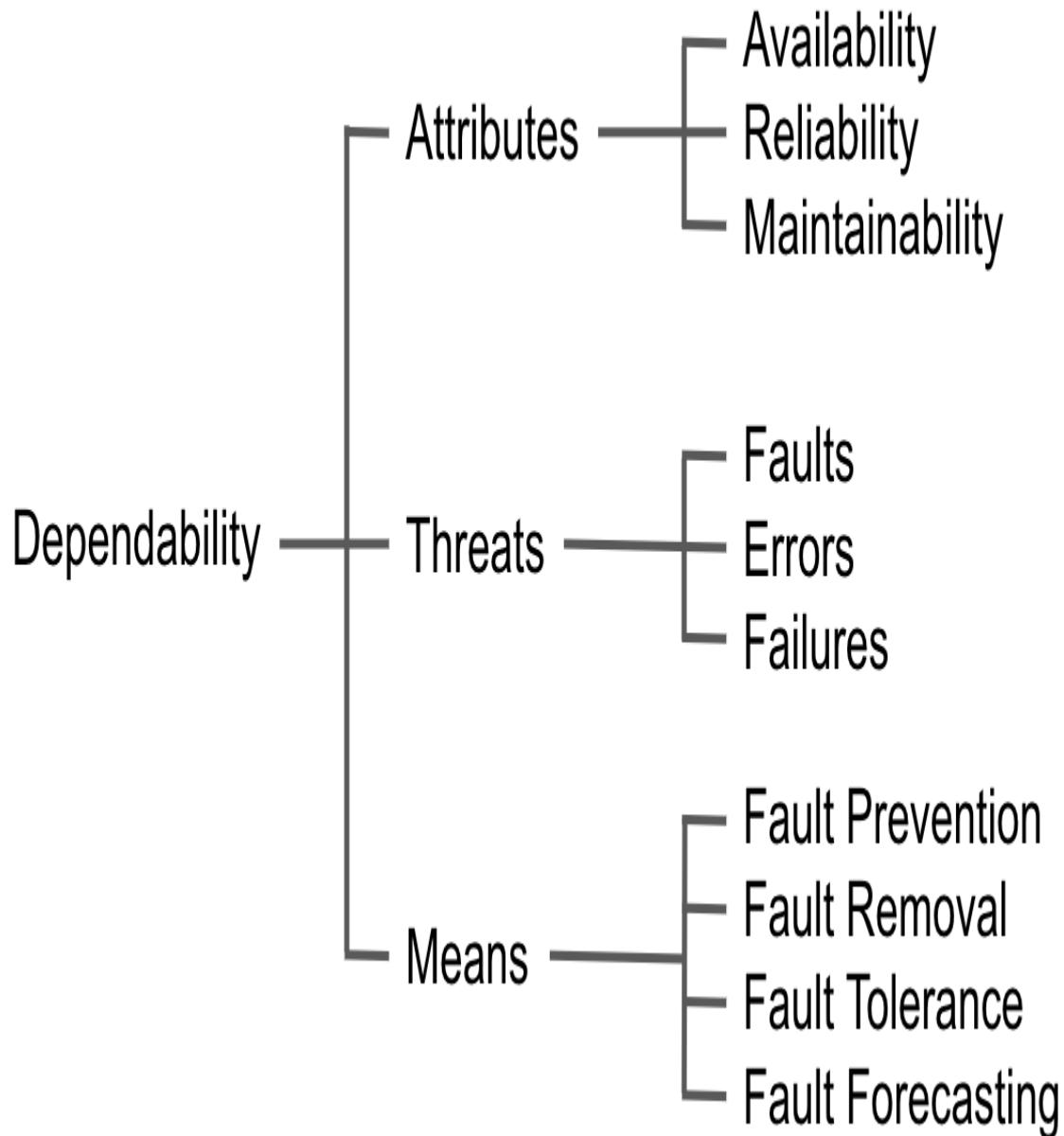
*The dependability of a computer system is its ability to avoid failures that are more frequent or more severe, and outage durations that are longer, than is acceptable to the user(s)<sup>7</sup>.*

—Algirdas Avižienis, et al.

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 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 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 be to provide correct service 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*<sup>8</sup>

The ability of a system to provide correct service for a given time interval. This is often expressed as the probability that a system will perform its required function for a specified period of 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.

#### **NOTE**

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

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

But 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 scalable set of rules around it.

Reliability, on the other hand, is quantitative. Given a robust definition<sup>9</sup> for what it means for a system to provide “correct” service, it becomes relatively straight-forward 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.

Fortunately, over the past decade or so — coinciding with the DevOps movement — a new wave of technologies have 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 met. Everything is software now.

The ever-increasing demand from users for service dependability has driven the creation of a whole new generation of cloud native technologies. The effect 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 good and well, but without

actionable advice for how to achieve dependability the entire discussion it 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 avoid of the occurrence or introduction of faults.

### *Fault Tolerance*

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

### *Fault Removal*

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

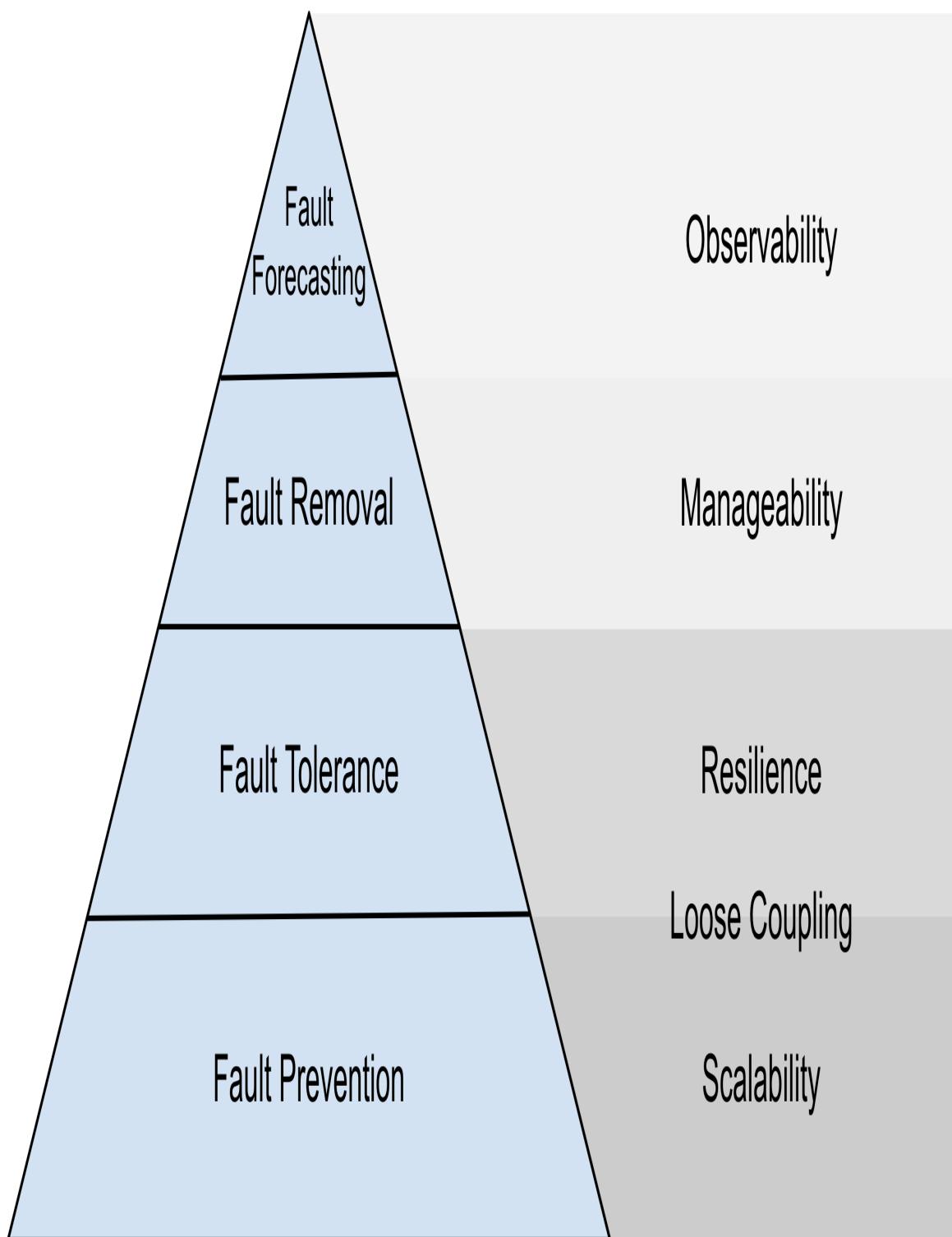
### *Fault Forecasting*

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](#).

## Means of Dependability

## Cloud Native Attributes



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

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. 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, mature observability practices naturally contribute to the ability to forecast faults. 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 now come full-circle.

## 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 development and code review practices. Many such techniques can really be grouped into what might be considered to be “good programming practice”, which we won’t explicitly cover here.

## Language Features

Your choice of language can also greatly affect to 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 cloud native services take a look 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 re-sizing existing resources, and horizontal scaling (scaling out) by adding (or removing) service instances — and some of the pros and cons of each.

In [Chapter 7](#) we’ll go far deeper into each of these, especially into the gotchas and downsides. Having to scale your service adds quite

a bit of overhead, including but not limited to cost, complexity, and debugging. Also state. Definitely state<sup>10</sup>.

While scaling resources eventually is 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, such as memory leaks and lock contention, that tend to plague systems at scale.

## Loose Coupling

Loose coupling, which we first defined in “[Loose coupling](#)”, is the system property and design strategy in which a system’s components have minimal knowledge of any other components. 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 — and increasingly, in recent months — there have been dissenters who point to the difficulty and complexity of deploying and maintaining a microservice-based system. To an extent I even agree with them, almost entirely because it’s so easy to build a distributed monolith that’s saddled with all of the complexity of microservices and all of the dependencies and entanglements of a monolith. You know you have a distributed monolith when you have to deploy most of your services together or when a failed health check sends cascading failures through your entire system.

Like any good relationship, the key to loosely coupled services is good communication and firm 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.

## 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 means to dependability, 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, a system can often 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 below.

## 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 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 "[Manageability](#)", 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 plugin-defined behaviors.

Clearly, manageability is a broad topic. We'll review a few of the mechanisms Go provides for it in [\[Link to Come\]](#).

## 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 [Link to Come], 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 2010's 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 felt were systemic problems in modern application development, they drafted *The Twelve-Factor App*, 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) that<sup>11</sup>

- 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;
- Minimize divergence between development and production, enabling continuous deployment for maximum agility;
- And 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, Codebase

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

Having multiple services that share 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 build and deploy phases of your service's life cycle.

## II. Dependencies

*Explicitly declare and isolate (code) dependencies.*

—The Twelve-Factor App, Dependencies

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<sup>12</sup>. 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 in the service's repository.

### III. Configuration

*Store configuration in the environment.*

—The Twelve-Factor App, Configuration

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.

## 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.

A common means of extracting configuration from code is by *externalizing* them into some configuration file — often YAML<sup>13</sup> — 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.

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.

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("BURROW")

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 upper-cased automatically
viper.SetDefault("id", "13")    // Default value is "13"

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

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

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

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 [Link to Come].

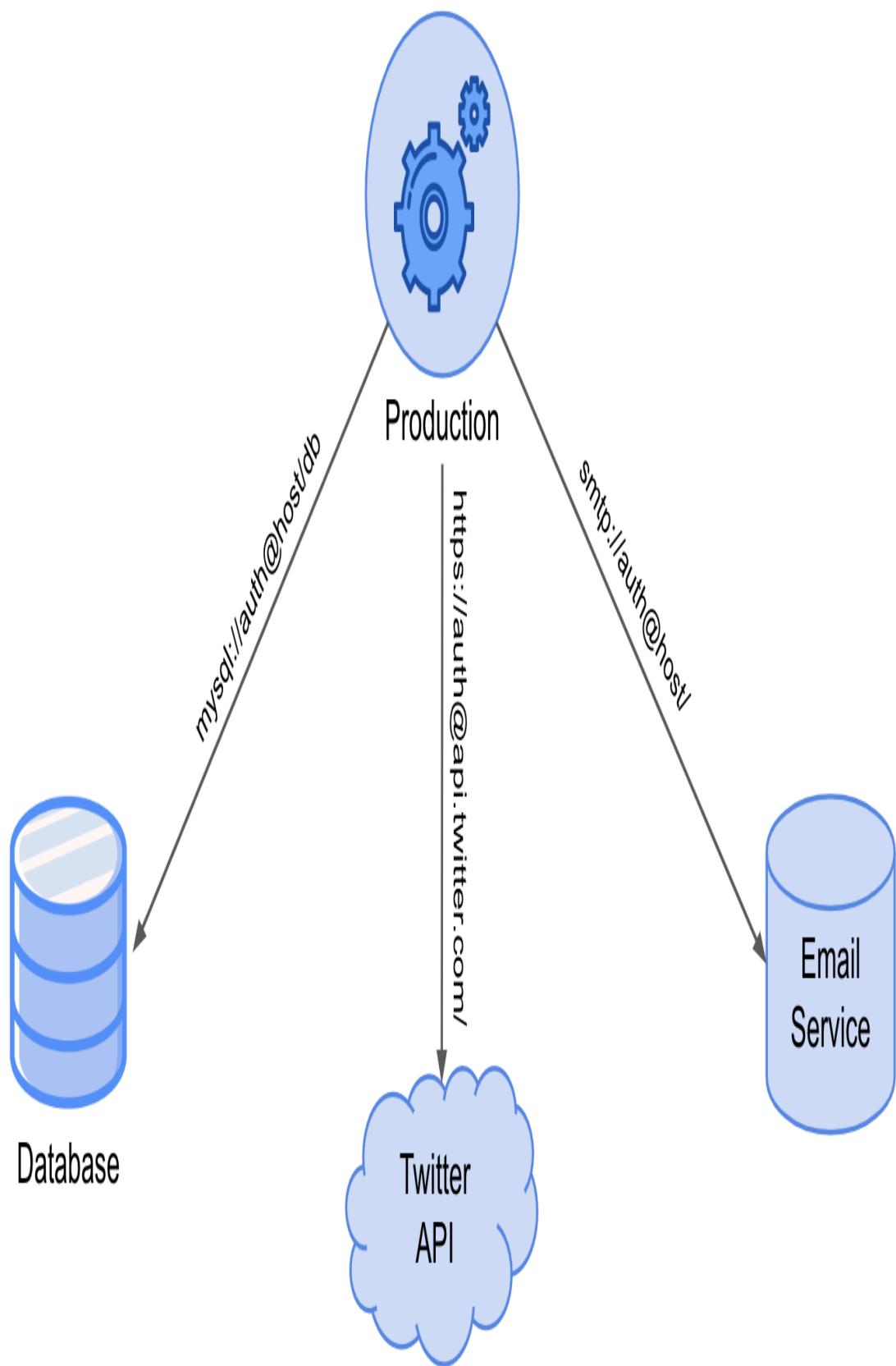
## IV. Backing Services

*Treat backing services as attached resources.*

—The Twelve-Factor App, Backing Services

A service should make no distinction between *backing services* — any upstream dependency (see “[Upstream and Downstream Dependencies](#)”) the service consumes across the network as part of its normal operation — of the same type. Whether it’s an internal service that’s managed with 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 on [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.

An 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, Build Release Run

Each (non-development) 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's 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 below:

### *Build*

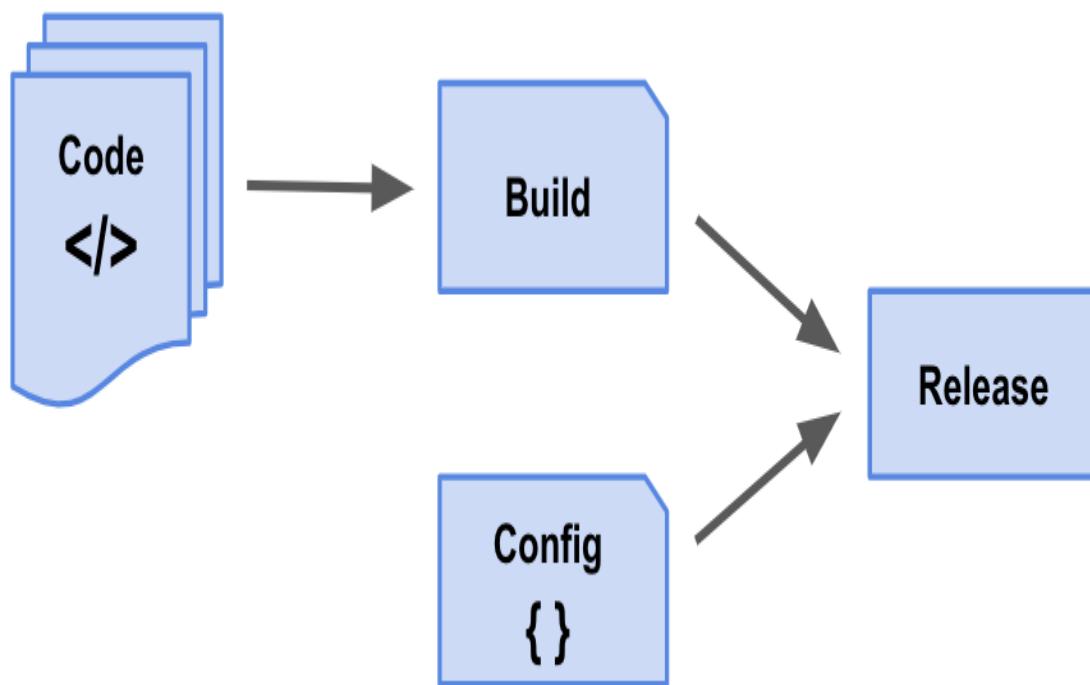
In the *build* stage, an automated process retrieves a specific version of the code, fetches dependencies, and compiles an executable artifact we call a *build*. Every build should always have a unique identifier, typically a timestamp or an incrementing build number.

## *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.



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

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

## VI. Processes

*Execute the app as one or more stateless processes.*

—The Twelve-Factor App, Processes

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](#)".

## VII. Data Isolation

*Each service manages its own data.*

—The Twelve-Factor App, Data Isolation

Each service should be entirely *self-contained*. That is, it should manage its own data, and only 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](#)".

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 title of this section in *The Twelve Factor App*, as originally written, is “Port Binding”, summarized as “export services via port binding”<sup>14</sup>.

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

While many (or even most) web application 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 Media).

And finally, although this is something you don’t see in the Go world, some languages allow (or even encourage!) runtime injection of an application server into the execution environment to create a web-facing service. However, 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, Concurrency

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<sup>15</sup>. Both scenarios end with a lot of unhappy users.

We'll discuss scalability quite a bit more in [Chapter 7](#).

## IX. Disposability

*Maximize robustness with fast startup and graceful shutdown.*

—The Twelve-Factor App, Disposability

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 re-deployed) 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 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 Our Key-Value Store](#)”.

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, Dev/Prod Parity

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. Hours later is good. Minutes later is better. 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 counter-productive 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

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 counter-intuitive) 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 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 [Link to Come].

## XII. Administrative Processes

*Run administrative/management tasks as one-off processes.*

—The Twelve-Factor App, Administrative Processes

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](#)”.

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:

1. 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.
2. 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 3am, 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<sup>16</sup> 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 is, often, “computer system that works in the cloud”. But “work” can be 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; and
- 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, I think that there's a lot of important, foundational information here to inform the motivations and means used to achieve what we call "cloud native".

---

1 Hoare, C.A.R. "[An Axiomatic Basis for Computer Programming](#)".

*Communications of the ACM*, vol. 12, no. 10, October 1969, pp. 576–583.

*Communications of the ACM*, vol. 12, no. 10, October 1969, pp. 576–583.

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.

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

5 Do you remember what Walt did to Jane that time? That was so messed up.

6 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.

- 7 A. Avižienis, J.-C. Laprie, and B. Randell, “[Fundamental Concepts of Dependability](#)”, *Research Report No. 1145, LAAS-CNRS*, April 2001.
- 8 If you’ve read any of O’Reilly’s SRE books, you’ll be very familiar with the concept of reliability already.
- 9 Many organizations use service-level objectives (SLOs) for precisely this purpose.
- 10 Application state is hard, and when done wrong it’s poison to scalability.
- 11 Wiggins, Adam. “[The Twelve-Factor App](#)”. *The Twelve-Factor App*, Adam Wiggins, 2011.
- 12 Although it was for too long!
- 13 The world’s worst configuration language (except for all the other ones).
- 14 Wiggins, Adam. “[Port Binding](#)”. *The Twelve-Factor App*, 2011.
- 15 Probably at 3 in the morning.
- 16 “Baking” is a term sometimes used to refer to the process of creating a new container or server image.

# Chapter 7. Scalability

---

## A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 7th chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at [ablevins@oreilly.com](mailto:ablevins@oreilly.com).

*Some of the best programming is done on paper, really. Putting it into the computer is just a minor detail<sup>1</sup>.*

—Max Kanat-Alexander, *Code Simplicity: The Fundamentals of Software* (March 2012)

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 when we found it 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.

So what did we learn from this, other than the benefits of autoscaling<sup>2</sup>?

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.

## What Is Scalability?

We first 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. 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 load increases in load.

Note that this definition<sup>3</sup> 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.

So how do we handle high demand without adding resources? As we'll discuss in "[Scaling Postponed: Efficiency](#)", systems built with *efficiency* in mind are 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".

## The Focus of This Chapter

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. So while we will briefly discussing things like scalable architecture and messaging patterns, most 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.

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).

## Bottlenecks and Scaling Targets

There are several different resources that might be considered scalability targets — whether by actually scaling the resource or by using it more efficiently — but nearly all scaling 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<sup>4</sup>, but network I/O is particularly amenable to various optimization strategies that we'll discuss below.

Why these four? Because as the load 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 to be bottlenecked *again* by another resource. For example, a database might avoid disk I/O bottlenecking by storing data in memory, or a service might scale up by adding more instances whose communication overhead puts additional strain on the network.

Furthermore, highly concurrent systems can often become victims of their own inner workings as the demand on them increases and phenomena like lock contention come into play.

## **Horizontal vs. Vertical 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, each with its own associated pros and cons:

### *Vertical scaling*

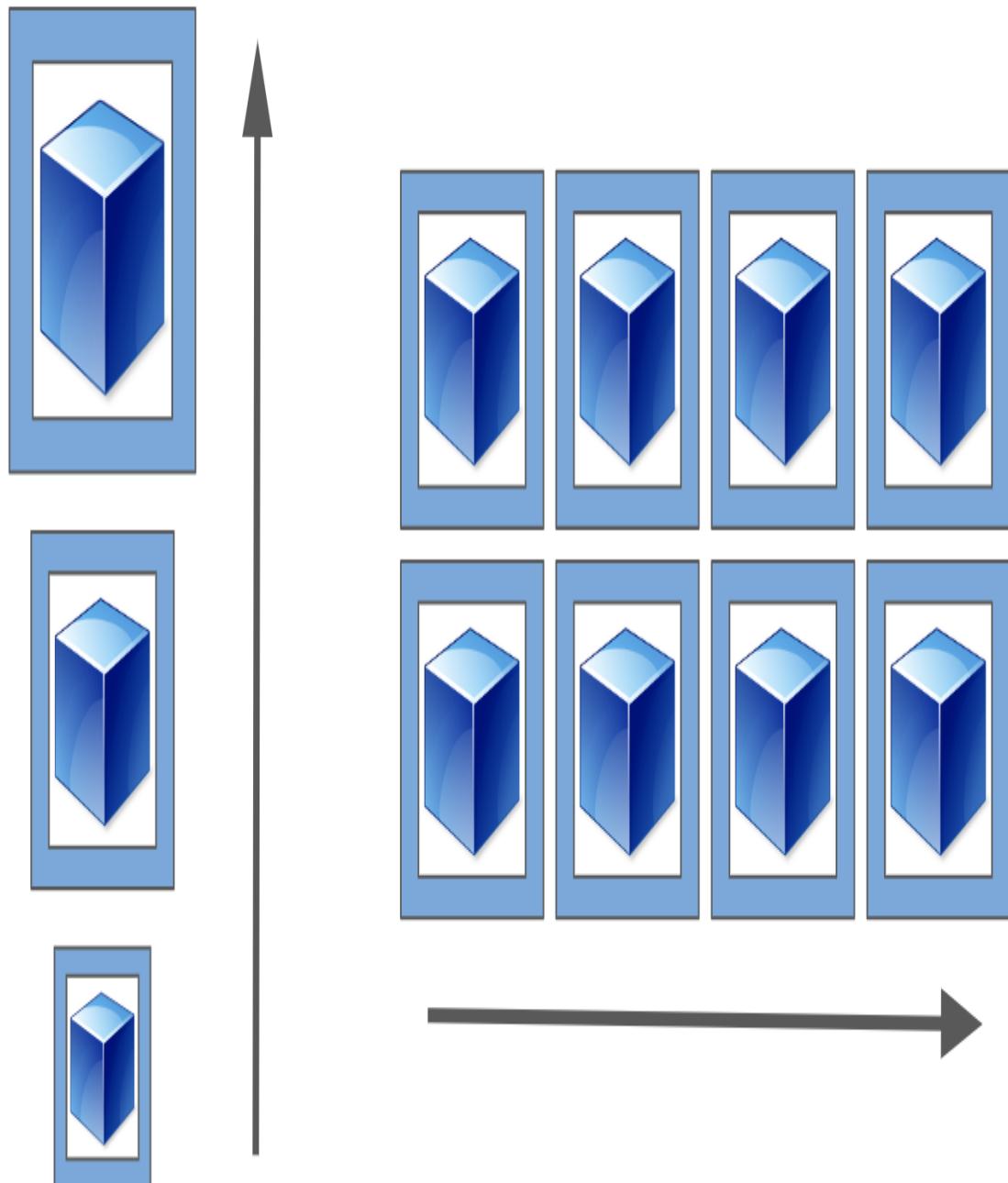
A system can be *vertically scaled* (or *scaled up*) by increasing the hardware resources that are already allocated to it. For example, by adding memory or CPU to a database that's running on a dedicated compute instance. In most public clouds an existing server can be vertically scaled by changing its instance size, or in a data center by replacing it with a new, more powerful appliance. Vertical scaling has the benefit of being technically relatively straight-forward, making it useful as a short-term fix. An instance can only be up-sized so much, however, so you may still eventually require a longer-term horizontal scaling strategy.

### *Horizontal scaling*

A system can be *horizontally scaled* (or *scaled out*) by adding service instances, splitting workloads between servers to limit the burden on any individual server. Examples include increasing the number of service nodes behind a load balancer or containers in Kubernetes or other container orchestration system. This strategy has a number of advantages, including redundancy and relative freedom from the limits of available instance sizes. However, more replicas means greater design and management complexity, and not all services can be horizontally scaled.

Vertical Scaling  
(increased instance size)

Horizontal Scaling  
(increased instance count)



*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.*

Given that there are two ways scaling a service — up or out — does that mean that any service whose hardware can be up-scaled is “scalable”? If you want to split hairs, then sure, to a point. But how scalable is it? Vertical scaling is inherently limited by the size of available computing resources, so in actuality a service that can only be scaled *up* isn’t very scalable at all. If you want to be able to scale by ten times, or a hundred, or a thousand, your service really has to be horizontally scalable.

## State and Statelessness

We briefly touched on statelessness in “[Application State vs Resource State](#)” 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<sup>5</sup>.

## Application State vs. Resource State

Not all state is created equal.

Any time an application needs to remember an event locally, that’s *application state*. A *stateful* application is one that’s designed to use this kind of local state; conversely a *stateless* application is free of any local persistent data, often because its state is stored in some external data store.

“Local” is an operative word here. Saying that an application is “stateless” doesn’t mean that it doesn’t have any data, rather that it’s been designed in such a way that any data that it needs doesn’t live

locally. Data that's the same for every client and which has nothing to do with the actions of clients, such as data stored in external data store or managed by configuration management, or is distinctly different from application state. This kind of data is sometimes referred to as *resource state*. Resource state is “good state” in the same way that “good cholesterol” is good.

To illustrate, imagine an application that tracks client sessions, associating them with some application context. If each user’s session data is maintained locally, by the application, it would be considered “application state”. If the data was stored in an external database, however, then it could be treated as a remote resource: 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 having to... well... manage their state<sup>6</sup>. Not being burdened with the need to maintain service-side state synchronization, consistency, and recovery logic<sup>7</sup> 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.

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.

## 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* straight-forward. 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.

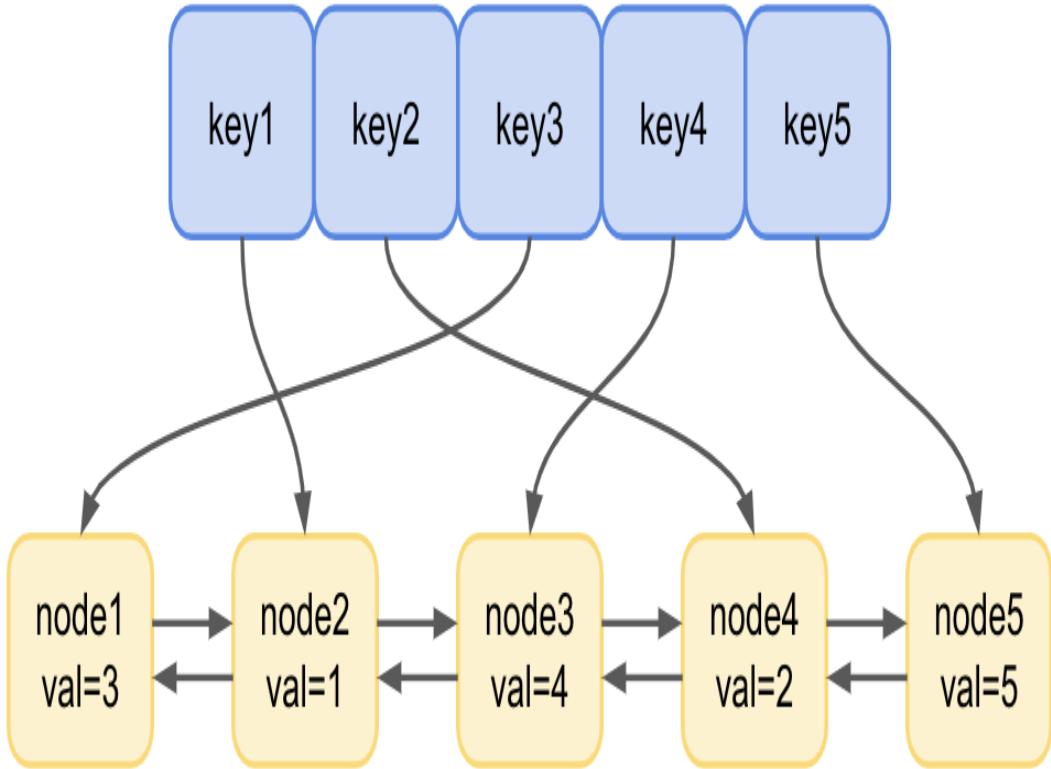
What happens as the number of cores and goroutines increases? Since you didn't consider concurrency, you'll soon find your modifications stepping on one another, leading to some unpleasant results. Also, since we forgot to remove anything from our map, it'll continue growing indefinitely until it consumes all of our memory.

So what we need is a cache that:

- supports concurrent read, write, and delete operations;
- scales well as the number of cores and goroutines increase; and
- 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 pre-defined 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.



*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.*

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.

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 2. 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, which is 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, 1)           // adds 1
    cache.Add(2, 2)           // adds 2; cache is now at capacity

    fmt.Println(cache.Get(1)) // returns (1, true); 1 now most
    // recently used

    cache.Add(3, 3)           // adds 3, evicts key 2

    fmt.Println(cache.Get(2)) // returns (nil, false) (not found)
}

```

In the above program, we create cache with a capacity of 2, which means that the addition of a third value will force the eviction of the least recently used value.

After adding the values 1 and 2 to the cache, we call `cache.Get(1)`, which makes 1 more recently used than 2. So when we add 3 in the next step, 2 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
1 true
evicted: key=2 value=2
<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. 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 on the *Dgraph Blog* entitled "[The State of Caching in Go](#)".

## 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 all-together.

(If you’re a bit fuzzy on the details of channels and goroutines, don’t stress. Take a moment to flip back to [“Goroutines”](#). It’s okay. We’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"<sup>8</sup> — to demonstrate this truism and show how Go channels can make concurrency safer and easier to reason about.

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 above, 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 one below 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.

```

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 above examples to highlight the relevant bits. For a walkthrough of a complete, idiomatic Go program that uses these ideas, see the Codewalk [Share Memory By Communicating](#).

## 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 "[Channel Buffering](#)", its 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 very especially useful for handling "bursty" loads. In fact, we already used this strategy in [Chapter 5](#) when we initialized our `FileTransactionLogger`. If you take a look back at "[Appending Entries to the Transaction Log](#)", you'll see something like 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 (edited to highlight the relevant bits), 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 the anonymous goroutine completed a receive operation. Most of the time that would probably be fine, 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 Vertical Sharding

As lovely as channels are, as we mentioned in “[Efficient Synchronization](#)” 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<sup>9</sup>.

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 Our Data Structure Concurrency-Safe](#)”. 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 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 “[Sharding](#)”. 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 haven’t done the research<sup>10</sup>, but based purely on my own personal experience I 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<sup>11</sup>, 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<sup>12</sup>. Whether by design or by accident, however, not every goroutine actually returns. For example:

```
func leaky() {
    ch := make(chan string)

    go func() {
        s := <-ch
        fmt.Println("Message:", s)
    }()
}
```

In the above 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 very 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.”<sup>13</sup>

## 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.

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. So 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.

## Service Architectures

The concept of the *microservice* first appeared in the early 2010's 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<sup>14</sup>.

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.

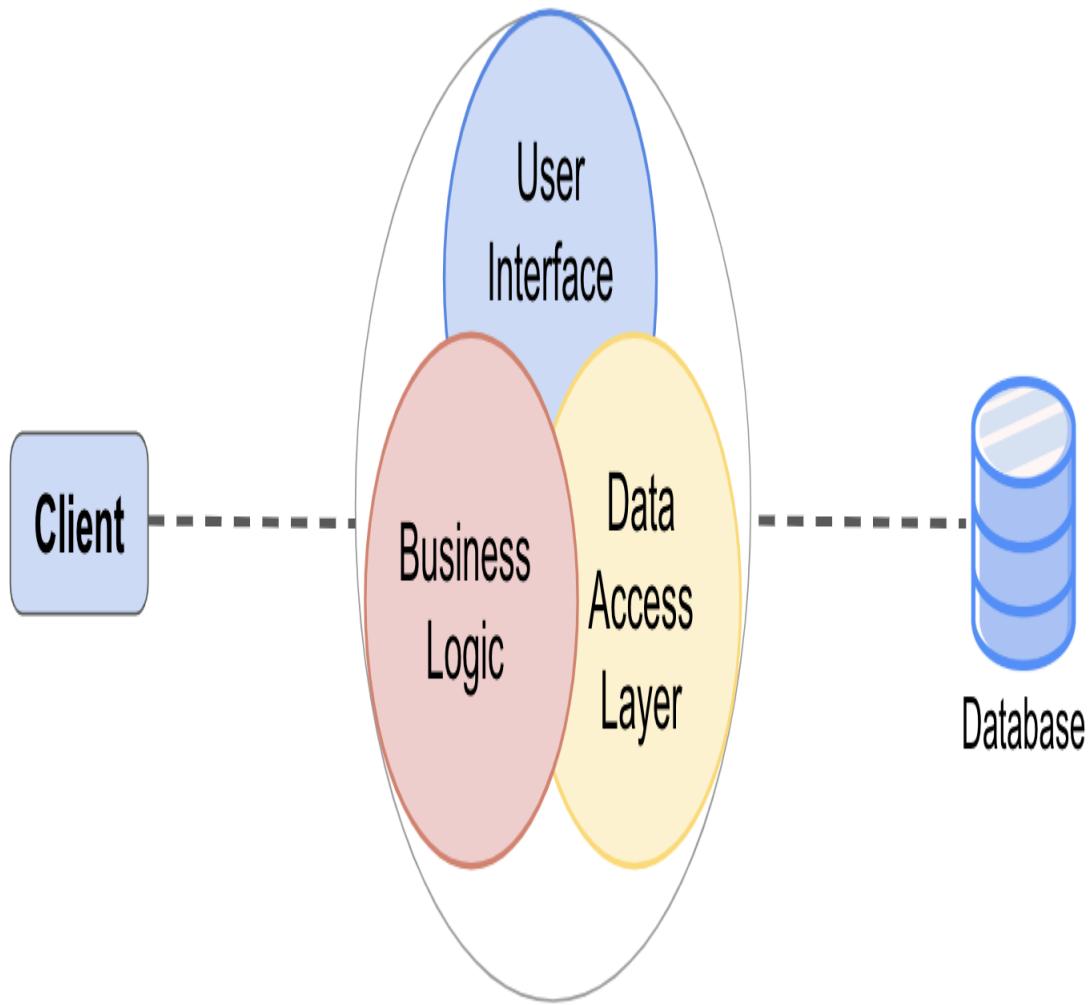
If you ask most programmers to compare monoliths to microservices, most of the answers you get will probably be something about how monoliths are slow, sluggish, and bloated, while microservices are small, agile, and the new hotness. Sweeping generalizations are always wrong, though, so let's take a moment to ask ourselves whether this is true, and whether monoliths might sometimes be the right choice.

So let's start 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:

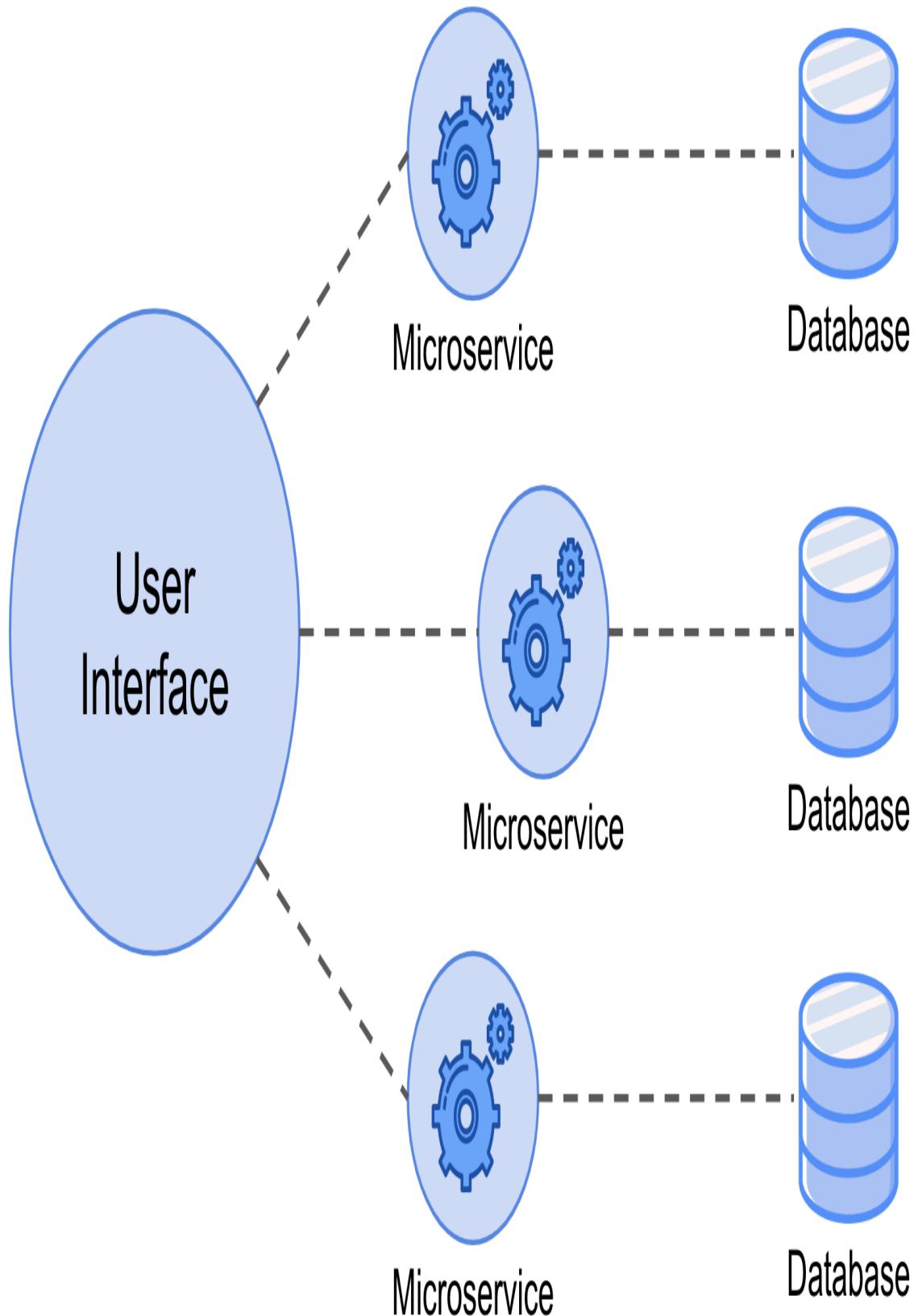
- Monoliths are usually deployed as a single artifact, so making even a small change generally requires a new version of the entire monolith to be built, tested, and deployed.
- Despite even the best of intentions and efforts, monolith code tends to decrease in modularity over time, making it harder 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 2000’s 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: “microservice 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 sub-services.*

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.

### WARNING

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<sup>15</sup> — 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 to weigh 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 working on it increases. For this reason many authors advocate starting with a monolith and decomposing it later.

On a personal note, I will also mention that I've never seen any organization successfully break apart a large monolith, but I've seen many try. So if you're pretty sure your application is going to get large and complex enough to require the kind of scale that make microservices appropriate, just start with microservices.

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 pre-defined 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 below, 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 Garner, which specializes in studying IT and technology trends, serverless infrastructure is hovering at or near the "Peak of Inflated Expectations"<sup>16</sup> 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 "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<sup>17</sup>. 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 under-utilized) servers.

### *Productivity*

In a FaaS model, the unit of work is an event-driven function. Because each function call is its own independent process, programmers can spend less time thinking about things like concurrency. Furthermore, because this model encourages a

“function first” mindset, the resulting code is 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 must be “spun up” by the cloud provider, which 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 function isn’t called for several minutes — the exact time varies between providers — it’s “spun down” by the provider so that has to endure another cold start when it’s called again. This 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 straight-forward 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.

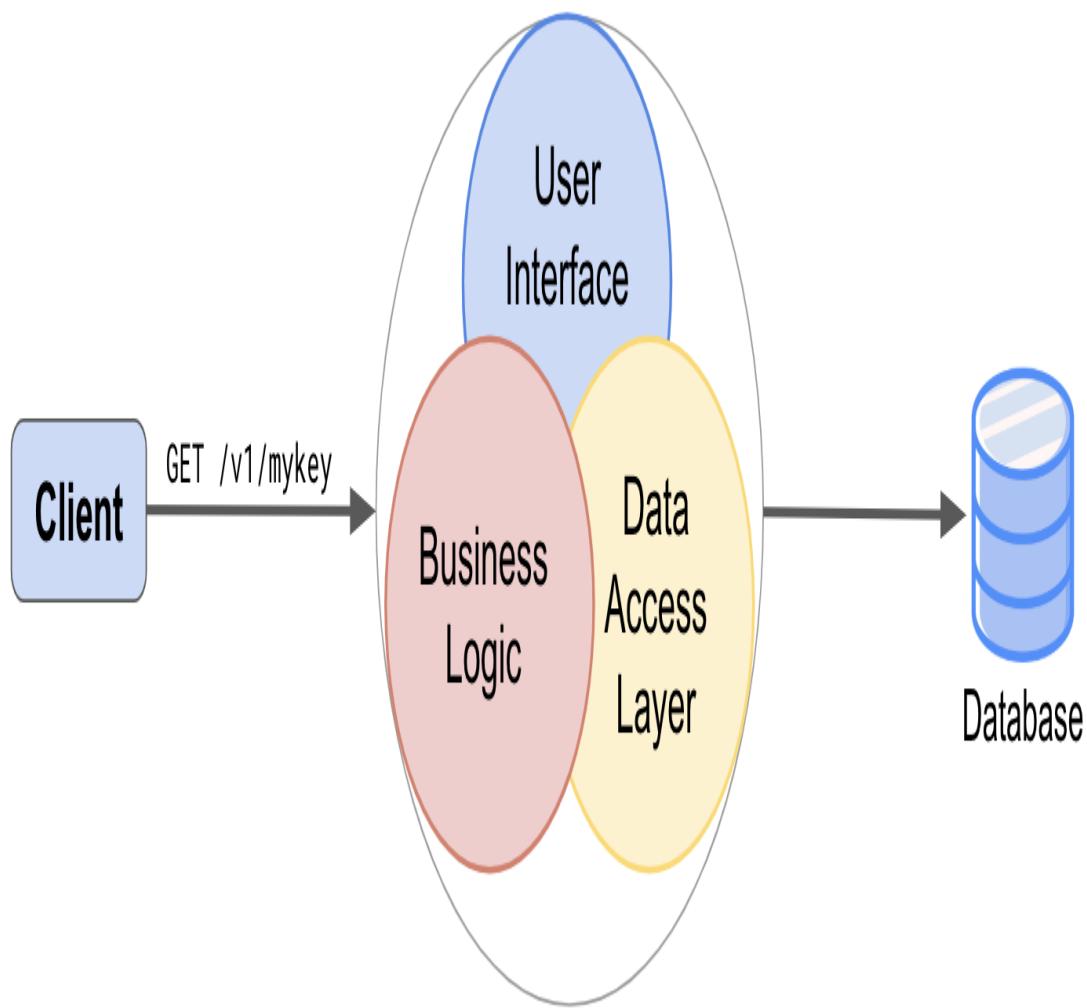
Clearly, there's quite a lot to consider — on both sides — and while there *is* a great deal of hype around serverless at the moment, to some degree I think it's merited. However, while serverlessness promises (and largely delivers) scalability and reduced costs, it does have quite a few gotchas, including but not limited to testing and debugging challenges. Not to mention the increased burden on operations around observability<sup>18</sup>!

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 does make it a bit harder to rapidly prototype and iterate.

## Serverless Services

As mentioned above, 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.

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 [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 [Figure 7-6](#), below.

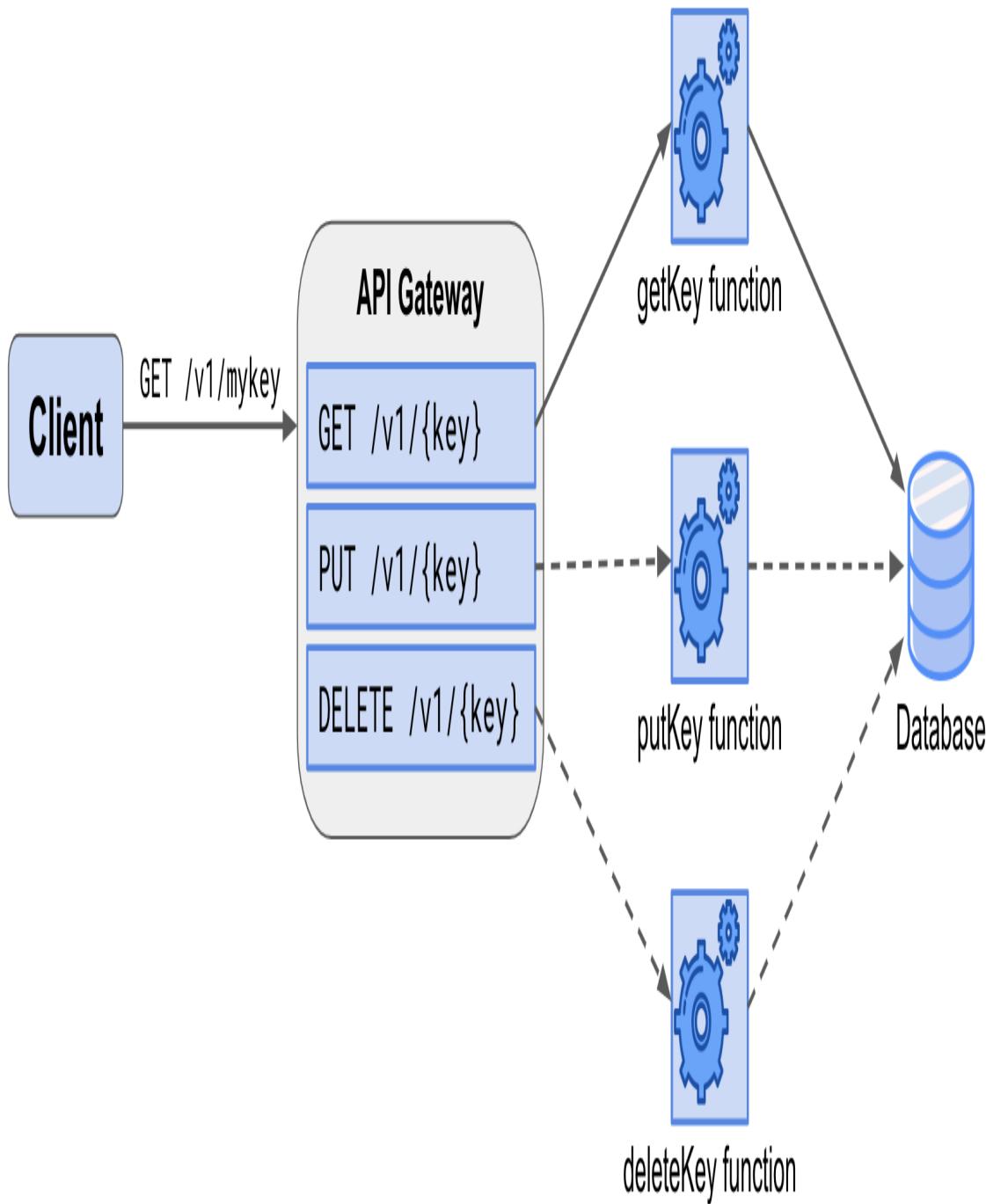


Figure 7-6. An API gateway routes HTTP calls to serverless handler functions.

In this example, we've replaced each of the server-side 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?

Some may argue that need for up-front planning is a *good thing*, but in any case it can make rapid iteration and prototyping more of a challenge.

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, though, 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 avoid memory leaks.
- We compared and contrasted monolithic, microservice, and serverless architectures.

That's quite a lot, and although I regret not being able to drill down in much detail, I'm pleased that we were able to touch on the things we did.

---

<sup>1</sup> 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 even wouldn't remember that this happened.
- 3 This is my definition. I acknowledge that it diverges from other common definitions.
- 4 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.
- 5 If you have a better definition, let me know. I'm already thinking about the 2nd edition.
- 6 I know I said the word "state" a bunch of times there. Writing is hard.
- 7 See also: idempotence.
- 8 Gerrand, Andrew. "Share Memory By Communicating." *The Go Blog*, 13 July 2010, <https://blog.golang.org/share-memory-by-communicating>. 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.
- 9 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.
- 10 Translation: I don't actually know what I'm talking about.
- 11 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.
- 12 There's a very good article by Vincent Blanchon on the subject of goroutine recycling entitled *How Does Go Recycle Goroutines?*
- 13 Cheney, Dave. "Never Start a Goroutine without Knowing How It Will Stop." dave.cheney.net, 22 Dec. 2016, <https://dave.cheney.net/2016/12/22/never-start-a-goroutine-without-knowing-how-it-will-stop>.
- 14 Not that they've gone away.
- 15 Yes, even Go.
- 16 Bowers, Daniel, et al. "Hype Cycle for Compute Infrastructure, 2019." Gartner, Gartner Research, 26 July 2019, <https://www.gartner.com/en/documents/3953649>.
- 17 It's *right in the name!*
- 18 Sorry, there's no such thing as No Ops.

# Chapter 8. Loose Coupling

---

## A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 8th chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at [ablevins@oreilly.com](mailto:ablevins@oreilly.com).

*We build our computers the way we build our cities — over time, without a plan, on top of ruins.<sup>1</sup>*

—Ellen Ullman, The Dumbing-down of Programming  
(May 1998)

Coupling is one of those fascinating topics that seem straightforward in theory but are actually quite challenging in practice. As we’ll discuss, there are lots of ways in which coupling can be introduced in a systems, which means it’s also a *big* subject. As you might imagine, this chapter is an ambitious one, and we cover a lot of ground.

First, we’ll introduce the subject, diving more deeply into the concept of “coupling”, and discussing the relative merits of “loose” vs “tight” coupling. We’ll present some of the most common coupling mechanisms, and how some kinds of tight coupling can lead to the dreaded “distributed monolith”.

Next, we'll talk about inter-service communications, and how fragile exchange protocols are a very common way of introducing tight coupling to distributed systems. We'll cover some of the common protocols in use today to minimize the degree of coupling between two services.

In the third part, we'll change directions for a bit, away from distributed systems and into the implementations of the services themselves. We'll talk about services as code artifacts, subject to coupling resulting from mingling implementations and violating separation of concerns, and present the use of plugins as a way to dynamically add implementations.

Finally, we'll close with a discussion of Hexagonal architecture, an architectural pattern that makes loose coupling the central pillar of its design philosophy.

Throughout the chapter, we'll do our best to balance theory, architecture, and implementation. Most of the chapter will be spent on the fun stuff: discussing a variety of different strategies for managing coupling, particularly (but not exclusively) in the distributed context, and demonstrating by extending our example key/value store.

## Loose and Tight Coupling

“Coupling” is a somewhat romantic term describing the degree of direct knowledge between components. For example, a client that sends requests to a service is by definition coupled to that service. The degree of that coupling can vary considerably, however, falling anywhere between two extremes.

“Tightly coupled” components have a great deal of knowledge about another component. Perhaps both require the same version of a shared library to communicate, or maybe the client needs an understanding of the server's architecture or database schema. It's

easy to build tightly coupled systems when optimizing for the short term, but they have a huge downside: the more tightly coupled two components are, the more likely that a change to one component will necessitate corresponding changes to the other. As a result, tightly coupled systems lose many of the benefits of a microservice architecture.

In contrast, “loosely coupled” components have minimal direct knowledge of one another. They’re relatively independent, typically interacting via a change-robust abstraction. Systems designed for loose coupling require more up-front planning, but they can be more freely upgraded, re-deployed, or even entirely rewritten without greatly affecting the systems that depend on them.

Put simply, if you want to know how tightly coupled your system is, ask how many and what kind of changes can be made to one component without adversely affecting another.

#### NOTE

Some amount of coupling is inevitable, and isn’t necessarily a bad thing, especially in early days of a system. It’s always tempting to over-abstract and over-complicate, and premature optimization is *still* the root of all evil.

## COUPLING IN DIFFERENT COMPUTING CONTEXTS

The term “coupling” in the computing context predates microservices and service-oriented architecture by quite a bit, and has been used for many years to describe the degree of knowledge that one component has about another.

- In programming, code can be tightly coupled when a dependent class directly references a concrete implementation instead of an abstraction (such as an interface; see “[Interfaces](#)”). In Go this might be a function that requires an `os.File` when an `io.Reader` would do.
- Multiprocessor systems that communicate by sharing memory can be said to be tightly coupled. In a loosely coupled system, components are connected through a MTS (Message Transfer System) (see “[Efficient Synchronization](#)” for a refresher on how Go solves this problem with channels).

It’s important to note that tight coupling isn’t always a bad thing. Eliminating abstractions and other intermediate layers can reduce overhead, which can be a useful optimization if speed is a critical system requirement.

Since this book is largely about distributed architectures, we’ll focus on coupling between services that communicate across a network, but keep in mind that there are other ways that software can be tightly coupled to resources in its environment.

## Tight Coupling Takes Many Forms

There’s no limit to the ways components in a distributed system can find themselves tightly coupled. However, while these all share one fundamental flaw — they all depend on some property of another

component that they wrongly assume won't change — most can be grouped into a few broad classes:

## Fragile Exchange Protocols

Remember SOAP? Statistically speaking, probably not<sup>2</sup>. SOAP was a messaging protocol developed in the late 1990s that was designed for extensibility and implementation neutrality. SOAP services provided a *contract* that clients could follow to format their requests<sup>3</sup>. This was something of a breakthrough at the time, but if the contract changed in any way the clients had to change along with it. In effect, clients were tightly coupled to services.

It didn't take long for people to realize that this was a problem, and SOAP quickly lost its luster. It has since been largely replaced by REST, which, while an considerable improvement, can often introduce its own tight coupling. In 2016, Google released gRPC (gRPC Remote Procedure Calls<sup>4</sup>), an open source framework with a number of useful features, including, importantly, allowing loose coupling between components.

We'll discuss some of these more contemporary options in ["Communications Between Services"](#), where we'll demonstrate how to use Go's `net/http` package to build a REST/HTTP client, and we'll extend our key/value store with a gRPC frontend.

## Shared Dependencies

In 2016, Facebook's Ben Christensen [gave a talk](#) at the Microservices Practitioner Summit where he spoke about another increasingly common mechanism for tightly coupling distributed services, introducing the term "distributed monolith" in the process.

Ben described an anti-pattern in which services were *required* to use specific libraries — and versions of libraries — in order to launch and interact with one another. Such systems find themselves saddled with a fleet-wide dependency, such that upgrading these shared libraries can force all services to have to upgrade in lockstep.

As we'll discuss in "[Distributed Monoliths](#)", this isn't the only way that you can find yourself with a distributed monolith, but it's probably one of the most popular ways.

## Shared Point-in-Time

Often systems are designed in such a way that clients expect an immediate response from services. Systems using this *request-response messaging* pattern implicitly assume that a service is present and ready to promptly respond. But if it's not, the request will fail. It can be said that they're *coupled in time*.

Coupling in time isn't necessarily bad practice, though. It might even be preferable, particularly when there's a human waiting for a timely response. We even detail how to construct such a client in the section "[Request-Response Messaging](#)".

But if the response isn't necessarily time constrained then a safer approach may be to send messages to an intermediate queue that recipients can retrieve from when they're ready, a messaging pattern commonly referred to as *publish-subscribe messaging* ("pub-sub" for short).

## Fixed Addresses

It's the nature of microservices that they need to take to one another. But to do that, they first have to find each other. This process of locating services on a network is called *service discovery*.

Traditionally, services lived at relatively fixed, well-known network locations that could be discovered by referencing some centralized registry. Initially this took the form of manually-maintained hosts.txt files, but as networks scaled up so did the adoption of DNS and URLs.

Traditional DNS works well for long-lived services whose locations on the network rarely change, but the increased popularity of ephemeral, microservice-based applications has ushered in a world

in which the lifespans of service instances is often measurable in seconds or minutes rather than months or years. In such dynamic environments URLs and traditional DNS become just another form of tight-coupling.

This need for dynamic, fluid service discovery has driven the adoption of entirely new strategies like the *service mesh*, a dedicated layer for facilitating service-to-service communications between resources in a distributed system.

Unfortunately, we won't be able to cover service discovery or service meshes in this book, but they're a fascinating and fast-developing subject. But the service mesh field is rich, with a number of mature open-source projects with active communities — such as [Envoy](#) (a CNCF-graduated project), [Linkerd](#) (a CNCF-incubating project), and [Istio](#) — and even commercial offerings like [Hashicorp's Consul](#).

## Distributed Monoliths

In [Chapter 7](#) we made the case that monoliths, at least for complex systems with multiple distinct functions, are (generally) undesirable and microservices are (generally) the way to go<sup>5</sup>. Of course, that's easier said than done, in large part because it's so easy to accidentally create a *distributed monolith*: a microservice-based system containing tightly coupled services.

In a distributed monolith, even small changes to one service can necessitate changes to others, often triggering unintended consequences. Services often can't be deployed independently, so deployments have to be carefully orchestrated, and errors in one component can send faults rippling through the entire system. Rollbacks are functionally impossible.

In other words, a distributed monolith is a “worst of all worlds” system that pairs the management and complexity overhead of multiple services with the dependencies and entanglements of a

monolith, losing many of the benefits of microservices in the process. Avoid at all costs.

## Communications Between Services

Communication and message passing is a critical function of distributed systems, and all distributed systems depend on some form of messaging to receive instructions and directions, exchange information, and provide results and updates. Of course, a message is useless if the recipient can't understand it.

In order for services to communicate, they must first establish an implicit or explicit *contract* that defines how messages will be structured. While such a contract is necessary, it also effectively couples the components that depend on it.

It's actually very easy to introduce tight coupling in this way, the degree of which reflected in the protocol's ability to change safely. Does it allow backward- and forward-compatible changes, like protocol buffers and gRPC, or do minor changes to the contract effectively break communications, as is the case with SOAP?

Of course, the data exchange protocol and its contract isn't the only variable in inter-service communications. There are, in fact, two broad classes of messaging patterns:

### *Request-Response (Synchronous)*

A two-way message exchange in which a requester (the client) issues a request of a receiver (the service) and waits for a response. A textbook example is HTML.

### *Publish-Subscribe (Asynchronous)*

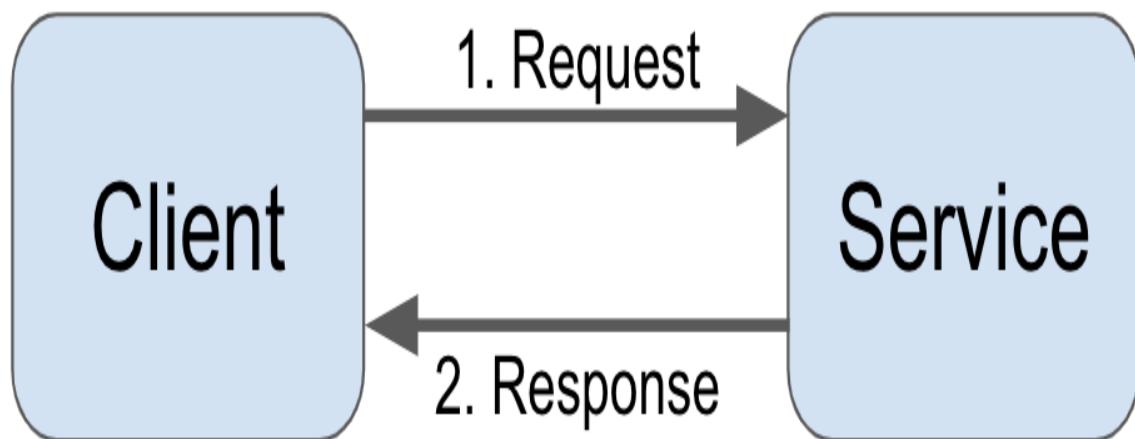
A one-way message exchange in which a requester (the publisher) issues a message to an event bus or message exchange, rather than directly to a specific receiver. Messages

can be retrieved asynchronously and acted upon by one or more services (subscribers).

Each of these patterns has a variety of implementations and particular use-cases, each with their own pros and cons. While we won't be able to cover every possible nuance, we'll do our best to provide a usable survey and some direction about how they may be implemented in Go.

## Request-Response Messaging

As its name suggests, systems using a *request-response* or *synchronous* messaging pattern communicate using a series of coordinated requests and responses, in which a requester (or client) submits a request to a receiver (or service) and waits until the receiver responds (hopefully) with the requested data or service.



*Figure 8-1. Systems using a request-response messaging pattern communicate using a series of coordinated requests and responses.*

The most obvious example of this pattern might be HTTP, which is so ubiquitous and well-established that it's been extended beyond its original purpose, and now underlies common messaging protocols like REST and GraphQL.

The request-response pattern has the advantages of being relatively easy to reason about and straight-forward to implement, and has long been considered the default messaging pattern, particularly for public-facing services. However, it's also "point-to-point"—involving exactly one requester and receiver—and requires the requesting process to pause until it receives a response.

Together, these properties make the request-response pattern a good choice for straight-forward exchanges between two endpoints where a response can be expected in reasonably short amount of time, but less than ideal when a message has to be sent to multiple receivers or when a response might take longer than a requester might want to wait.

## Common Request-Response Implementations

### *REST*

You're likely already very familiar with REST, which we discussed in some detail in "[Building an HTTP Server With `net/http`](#)". REST has some things going for it. It's human-readable and easy to implement, making it a good choice for outward-facing services (which is why we chose it in [Chapter 5](#)). We'll discuss a little more in "[Issuing HTTP Requests With `net/http`](#)".

### *GraphQL*

GraphQL is a query and manipulation language generally considered an alternative to REST, and is particularly powerful when working with complex datasets. We don't discuss GraphQL in much detail in this book, but we encourage you to look into it the next time you're designing an outward-facing API.

### *Remote Procedure Calls (RPC)*

Remote procedure call (RPC) frameworks allow programs to execute procedures in a different address space, often on

another computer. Go provides a standard Go-specific RPC implementation in the form of `net/rpc`. There are also two big language-agnostic RPC players: Apache Thrift and gRPC. While similar in design and usage goals, gRPC seems to have taken the lead with respect to adoption and community support. We'll discuss gRPC in much more detail in "["Remote Procedure Calls With gRPC"](#)".

## Issuing HTTP Requests With `net/http`

HTTP is perhaps the most common request-response protocol, particularly for public-facing services, underlying popular API formats like REST and GraphQL. If you're interacting with an HTTP service, you'll need some way to programmatically issue requests to the service and retrieve the response.

Fortunately, the Go standard library comes with excellent HTTP client and server implementations in the form of the `net/http` package. You may remember `net/http` from "["Building an HTTP Server With `net/http`"](#)", where we used it to build the first iteration of our key/value store.

The `net/http` includes, among other things, convenience functions for GET, HEAD, and POST methods. The signatures for the first of these, `http.Get` and `http.Head`, are shown below:

```
// Get issues a GET to the specified URL
func Get(url string) (*http.Response, error)

// Head issues a HEAD to the specified URL
func Head(url string) (*http.Response, error)
```

The above functions are very straight forward, and are both used similarly: each accepts a `string` that represents the URL of interest, and each returns an `error` value and a pointer to an `http.Response` struct.

The `http.Response` struct is particularly useful because it contains all kinds of useful information about the service's response to our request, including the returned status code and the response body.

A small selection of the `http.Response` struct is below:

```
type Response struct {
    Status      string      // e.g. "200 OK"
    StatusCode int         // e.g. 200

    // Header maps header keys to values.
    Header Header

    // Body represents the response body.
    Body io.ReadCloser

    // ContentLength records the length of the associated content. The
    // value -1 indicates that the length is unknown.
    ContentLength int64

    // Request is the request that was sent to obtain this Response.
    Request *Request
}
```

There are some useful things in there! Of particular interest is the `Body` field, which provides access to the HTTP response body. It's a `ReadCloser` interface, which tells us two things: that the response body is streamed on demand as it's read, and that it has a `Close` method that we're expected to call.

Below we demonstrate several things: how to use the `Get` convenience function, how to close the response body, and how to use `ioutil.ReadAll` to read the *entire* response body as a string (if you're into that kind of thing).

```
package main

import (
    "fmt"
    "io/ioutil"
    "net/http"
```

```

)
func main() {
    resp, err := http.Get("http://example.com")      // Send an HTTP
GET
    if err != nil {
        panic(err)
    }
    defer resp.Body.Close()                         // Close your
response!

    body, err := ioutil.ReadAll(resp.Body)          // Read body as
[]byte
    if err != nil {
        panic(err)
    }

    fmt.Println(string(body))
}

```

In the above example, we use the `http.Get` function to issue a GET to the URL `http://example.com`, which returns a pointer to a `http.Response` struct and an `error` value.

As we mentioned above, access to the HTTP response body is provided via the `resp.Body` variable, which implements `io.ReadCloser`. Note how we defer the call `resp.Body.Close()`. This is very important: failing to close your response body can sometimes lead to some unfortunate memory leaks.

Because `Body` implements `io.Reader`, we have many different standard means to retrieve its data. In this case we use the very reliable `ioutil.ReadAll`, which conveniently returns the entire response body as a `[]byte` slice, which we simply print.

### WARNING

Always remember to use `Close()` to close your response body!  
Not doing so can lead to some unfortunate memory leaks.

We've already seen the Get and Head functions, but how do we issue POSTs? Fortunately, similar convenience functions exist for them too. Two, in fact: `http.Post` and `http.PostForm`. The signatures for each of these are shown below:

```
// Post issues a POST to the specified URL
func Post(url, contentType string, body io.Reader) (*Response, error)

// PostForm issues a POST to the specified URL, with data's keys
// and values URL-encoded as the request body
func PostForm(url string, data url.Values) (*Response, error)
```

The first of these, `Post`, expects an `io.Reader` that provides the body — such as a file of a JSON object — of the post. We demonstrate below how to upload JSON text in a POST:

```
package main

import (
    "fmt"
    "io/ioutil"
    "net/http"
    "strings"
)

const json = `{"name":"Matt", "age":44}` // This is our JSON

func main() {
    in := strings.NewReader(json)           // Wrap JSON with an
    io.Reader

    // Issue HTTP POST, declaring our content-type as "text/json"
    resp, err := http.Post("http://example.com/upload", "text/json",
    in)
    if err != nil {
        panic(err)
    }
    defer resp.Body.Close()                // Close your
    response!

    message, err := ioutil.ReadAll(resp.Body)
    if err != nil {
```

```
        panic(err)
    }

    fmt.Printf(string(message))
}
```

## A POSSIBLE PITFALL OF CONVENIENCE FUNCTIONS

We've been referring to the Get, Head, Post, and PostForm functions as "convenience functions", but what does that mean?

It turns out that, under the hood, each is actually calling a method on a default `*http.Client` value, a concurrency-safe type that Go uses to manage the internals of communicating over HTTP.

The code for the Get convenience function, for example, is actually a call to the default client's `http.Client.Get` method:

```
func Get(url string) (resp *Response, err error) {
    return DefaultClient.Get(url)
}
```

As you can see, when you use `http.Get`, you're actually using `http.DefaultClient`. Because `http.Client` is concurrency safe, it's possible to have only one of these, pre-defined as a package variable.

The source code for the creation of `DefaultClient` itself is somewhat plain, creating a zero-value `http.Client`:

```
var DefaultClient = &Client{}
```

Generally, this is perfectly fine. However, there's a potential issue here, and it involves timeouts. The `http.Client` methods are capable of asserting timeouts that terminate long-running requests. This is super useful. Unfortunately, the default timeout value is 0, which Go interprets as "no timeout".

Okay, so Go's default HTTP client will never time out. Is that a problem? *Usually* not, but what if it connects to a server that doesn't respond and doesn't close the connection? The result would be an especially nasty and non-deterministic memory leak.

But how do we fix this? Well, as it turns out `http.Client` *does* support timeouts, we just have to enable that functionality by creating a custom Client and setting a timeout:

```
var client = &http.Client{
    Timeout: time.Second * 10,
}
response, err := client.Get(url)
```

Take a look at the [net/http package documentation](#) for more information about `http.Client` and its possible settings.

## Remote Procedure Calls With gRPC

gRPC is an efficient, polyglot data exchange framework that was originally developed by Google as the successor to *Stubby*, a general-purpose RPC (Remote Procedure Call) framework that had been in use internally at Google for over a decade. It was open sourced in 2015 under the name gRPC, and taken over by the Cloud Native Computing Foundation in 2017.

Unlike REST, which is essentially a set of unenforced best practices, gRPC is a fully-featured data exchange framework, which, like other RPC frameworks such as SOAP, Apache Thrift, Java RMI, and CORBA (to name a few) allows a client to execute specific methods implemented on different systems as if they were local functions.

This approach has a number of advantages over REST, including but not limited to:

- **Conciseness.** Its messages are more compact, consuming less network I/O.
- **Speed.** Its binary exchange format is much faster to marshal and unmarshal.

- **Strong-typing.** It's natively strongly typed, eliminating a lot of boilerplate and removing a common source of errors.
- **Feature-rich.** It has a number of built-in features like authentication, encryption, timeout, and compression (to name a few) that you would otherwise have to implement yourself.

That's not to say that gRPC is always the best choice. Compared to REST:

- **Contract-driven.** gRPC's contracts make it less suitable for external-facing services.
- **Binary format.** gRPC data isn't human readable, making it harder to inspect and debug.

#### TIP

gRPC is a very large and rich subject that this modest section can't fully do justice to. If you're interested in learning more we recommend the official [Introduction to gRPC](#) and the excellent [gRPC: Up and Running by Kasun Indrasiri and Danesh Kuruppu](#) (O'Reilly Media).

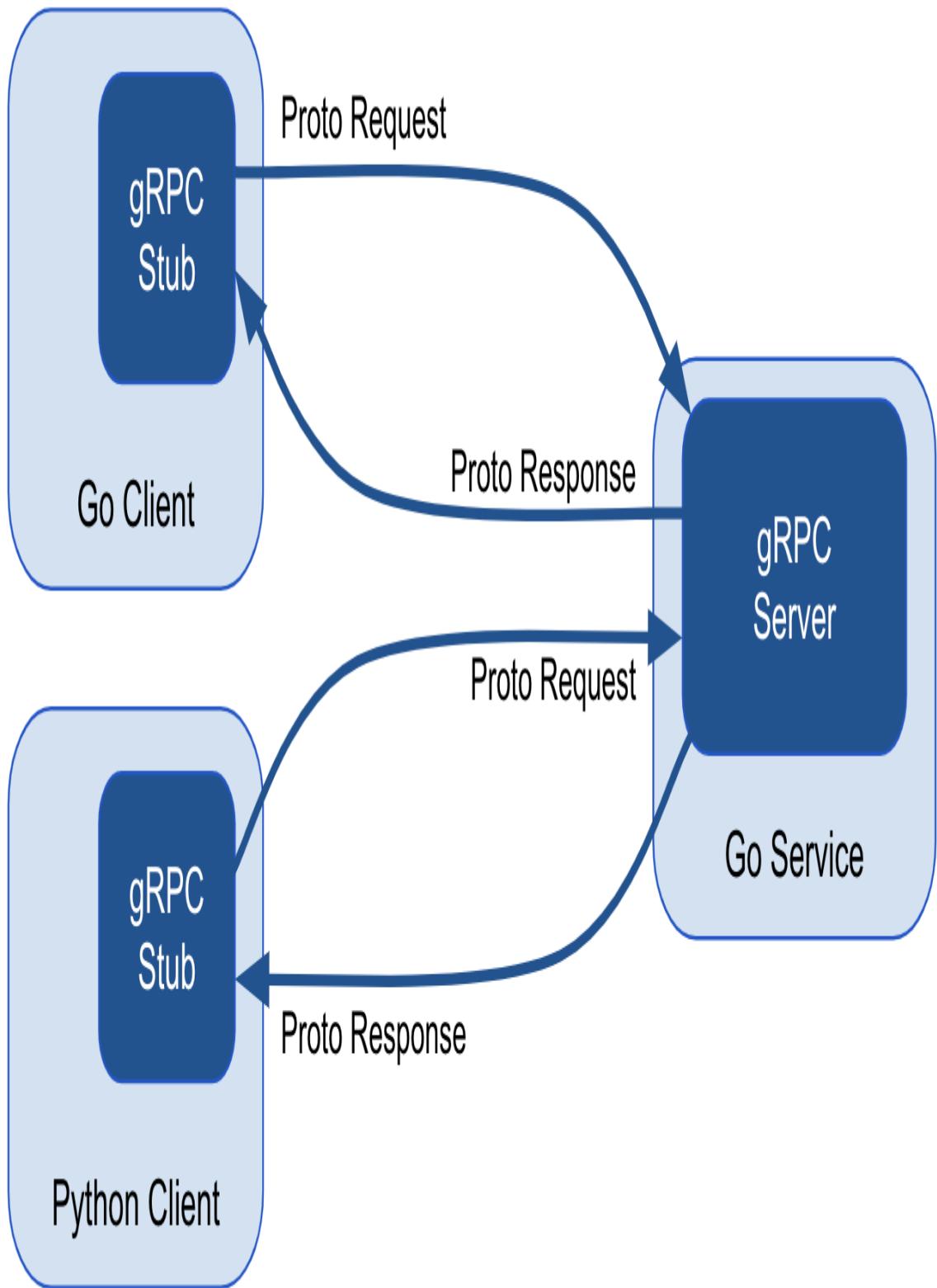
## Interface Definition with Protocol Buffers

As is the case with most RPC frameworks, gRPC requires you to define a *service interface*. By default, gRPC uses [protocol buffers](#) for this purpose, though it's possible to use an alternative Interface Definition Language (IDL) like JSON if you want.

To define a service interface, the author uses the protocol buffers schema to describe the service methods that can be called remotely by a client in a `.proto` file. This is then be *compiled* into a language-specific interface; Go code, in our case.

As illustrated in [Figure 8-2](#), gRPC servers implement the resulting source code to handle client calls, while the client has a stub that provides the same methods as the server.

Yes, this seems very hand-wavey and abstract right now. Keep reading for some more details!



*Figure 8-2. By default, gRPC uses protocol buffers as both its Interface Definition Language and its underlying message interchange format. Servers and clients can be written in **any supported language**.*

## Installing the Protocol Compiler

Before we proceed, we'll first need to install the protocol buffer compiler, protoc, and the Go protocol buffers plugin. We'll use these to compile .proto files into Go service interface code.

1. If you're using Linux or MacOS, the simplest and easiest way to install protoc is to use a package manager, as follows:

**Linux.** To install on a Debian-flavored Linux you can use apt or apt-get:

```
$ apt install -y protobuf-compiler  
$ protoc --version
```

**MacOS.** The easiest way to install on MacOS is to use Homebrew:

```
$ brew install protobuf  
$ protoc --version
```

2. Run the following command to install the Go protocol buffers plugin:

```
$ go install google.golang.org/protobuf/cmd/protoc-gen-go
```

The compiler plugin protoc-gen-go will be installed in \$GOBIN, defaulting to \$GOPATH/bin. It must be in your \$PATH for protoc to find it.

## WARNING

This book uses protocol buffers version 3. Be sure to check the version of protoc after installation to make sure that it's version 3 or higher.

If you're using another OS, your chosen package manager has an old version, or if you just want to make sure you have the latest and greatest, you can find the instructions for installing the pre-compiled binaries on gRPC's [Protocol Buffer Compiler Installation](#) page.

## The Message Definition Structure

Protocol buffers are a language-neutral mechanism for serializing structured data. You can think of it as a binary version of XML<sup>6</sup>. Protocol buffer data is structured as *messages* where each message is a small record of information containing a series of name-value pairs called *fields*.

The first step when working with protocol buffers is to define the message structure by defining it in a .proto file. A basic example is presented below:

*Example 8-1. An example .proto file. The message definitions define remote procedure payloads.*

---

```
syntax = "proto3";

option go_package = "github.com/cloud-native-go/ch08/point";

// Point represents a labeled position on a 2-dimensional surface
message Point {
    int32 x = 1;
    int32 y = 2;
    string label = 3;
}

// Line contains start and end Points
message Line {
    Point start = 1;
    Point end = 2;
```

```
    string label = 3;
}

// Polyline contains any number (including zero) of Points
message Polyline {
    repeated Point point = 1;
    string label = 2;
}
```

You'll may have noticed that the protocol buffer syntax is reminiscent of C/C++, complete with its semicolon and commenting syntax.

The first line of the file specifies that you're using proto3 syntax: if you don't do this the protocol buffer compiler will assume you are using proto2. This must be the first non-empty, non-comment line of the file.

The second line uses the option keyword to specify the full import path of the Go package that will contain the generated code.

Finally, we have three message definitions, which describe the structure of the payload messages. In this example, we have three message of increasing complexity:

- Point, which contains x and y integer values, and a label string,
- Line, which contains exactly two Point values, and
- Polyline which uses the repeated keyword to indicate that it can contain any number of Point values.

Each message contains zero or more fields that have a name and a type. Note that each field in a message definition has a *field number* that is unique for that message type. These are used to identify fields in the message binary format, and should not be changed once your message type is in use.

If this raises a “tight coupling” red flag in your mind, you get a gold star for paying attention. For this reason, protocol buffers provide

explicit support for [updating message types](#), including marking a field as [reserved](#) so that it can't be accidentally reused.

This example is incredibly simple, but don't let that fool you: protocol buffers are capable of some very sophisticated encodings. See the [Protocol Buffers Language Guide](#) for more information.

## The Key-Value Message Structure

So how do we make use of this to extend the example key-value store that we started in [Chapter 5](#)?

Let's say that we want to implement gRPC equivalents to the Get, Put, and Delete functions we are already exposing via RESTful methods. The message formats for that might look something like the following .proto file:

*Example 8-2. keyvalue.proto—The messages that will be passed to and from our key-value service procedures.*

---

```
syntax = "proto3";

option go_package = "github.com/cloud-native-go/ch08/keyvalue";

// GetRequest represents a request to the key-value store for the
// value associated with a particular key
message GetRequest {
    string key = 1;
}

// GetResponse represents a response from the key-value store for a
// particular value
message GetResponse {
    string value = 1;
}

// PutRequest represents a request to the key-value store for the
// value associated with a particular key
message PutRequest {
    string key = 1;
    string value = 2;
}
```

```
// PutResponse represents a response from the key-value store for a
// Put action.
message PutResponse {}

// DeleteRequest represents a request to the key-value store to delete
// the record associated with a key
message DeleteRequest {
    string key = 1;
}

// DeleteResponse represents a response from the key-value store for a
// Delete action.
message DeleteResponse {}
```

### TIP

Don't let the names of the message definitions confuse you: they represent *messages* — nouns — that will be passed to and from functions — verbs — that we'll define in the next section.

In the above .proto file, which we'll call `keyvalue.proto`, we have three Request message definitions describing messages that will be sent from the client to the server, and three Response message definitions describing the server's response messages.

You may have noticed that we don't include error or status values in the message response definitions. As you'll see in "[Implementing the gRPC Client](#)", these are unnecessary because they're included in the return values of the gRPC client functions.

## Defining Our Service Methods

So now that we've completed our message definitions, we'll need to describe the methods that'll use them.

To do that, we extend our `keyvalue.proto` file, using the `rpc` keyword to define our service interfaces. Compiling the modified .proto file will generate Go code that includes the service interface code and client stubs.

### *Example 8-3. keyvalue.proto—The procedures for our key-value service.*

---

```
service KeyValue {  
    rpc Get(GetRequest) returns (GetResponse);  
  
    rpc Put(PutRequest) returns (PutResponse);  
  
    rpc Delete(DeleteRequest) returns (DeleteResponse);  
}
```

#### TIP

In contrast to the messages defined in [Example 8-2](#), the `rpc` definitions represent functions — verbs — which will send and receive messages — nouns.

In this example, we add three methods to our service: `Get` which accepts a `GetRequest` and returns a `GetResponse`, `Put` which accepts a `PutRequest` and returns a `PutResponse`, and `Delete` which accepts a `DeleteRequest` and returns a `DeleteResponse`. Note that we don't actually implement the functionality here. We do that later.

The above methods are all examples of *unary RPC* definitions, in which a client sends a single request to the server and gets a single response back. This is the simplest of the four service methods types. Various streaming modes are also supported, but these are beyond the scope of this simple primer. The [gRPC documentation](#) discusses these in more detail.

## Compiling your Protocol Buffers

Now that you have a `.proto` file complete with message and service definitions, the next thing you need to do is generate the classes you'll need to read and write messages. To do this, you need to run the protocol buffer compiler `protoc` on our `keyvalue.proto`.

If you haven't installed the protoc compiler and Go protocol buffers plugin, follow the directions in "[Installing the Protocol Compiler](#)" to do so.

Now you can run the compiler, specifying the source directory (`$SOURCE_DIR`) where your application's source code lives (defaults to the current directory), the destination directory (`$DEST_DIR`; often the same as `$SOURCE_DIR`), and the path to your `keystore.proto`. Because we want Go code, you use the `--go_out` option. Similar options are provided for other supported languages.

In this case, we would invoke:

```
$ protoc --proto_path=$SOURCE_DIR \
    --go_out=$DEST_DIR --go_opt=paths=source_relative \
    --go-grpc_out=$DEST_DIR --go-grpc_opt=paths=source_relative \
    $SOURCE_DIR/keyvalue.proto
```

The `go_opt` and `go-grpc_opt` flags tell `protoc` to place the output files in the same relative directory as the input file. Our `keyvalue.proto` file results in two files, named `keyvalue.pb.go` and `keyvalue_grpc.pb.go`.

Without these flags, the output files are placed in a directory named after the Go package's import path. Our `keyvalue.proto` file, for example, would result in a file named `github.com/cloud-native-go/ch08/keyvalue/keyvalue.pb.go`.

## Implementing the gRPC Service

To implement our gRPC server, we'll need to implement the generated service interface, which defines the server API for our key-value service. It can be found in the `keyvalue_grpc.pb.go` as `KeyValueServer`:

```
type KeyValueServer interface {
    Get(context.Context, *GetRequest) (*GetResponse, error)
    Put(context.Context, *PutRequest) (*PutResponse, error)
```

```
        Delete(context.Context, *DeleteRequest) (*PutResponse, error)
    }
```

As you can see, the `KeyValueServer` interface specifies our `Get`, `Put`, and `Delete` methods: each accepts a `context.Context` and a request pointer, and returns a response pointer and an `error`.

### TIP

As a side-effect of its simplicity, it's dead easy to mock requests to and responses from a gRPC server implementation.

To implement our server, we'll make use of a generated struct that provides a default implementation for the `KeyValueServer` interface, which in our case is named `UnimplementedKeyValueServer`. It's so named because it includes default "unimplemented" versions of all of our client methods attached, which look something like the following:

```
type UnimplementedKeyValueServer struct {}

func (*UnimplementedKeyValueServer) Get(context.Context, *GetRequest)
    (*GetResponse, error) {

    return nil, status.Errorf(codes.Unimplemented, "method not
implemented")
}
```

By embedding the `UnimplementedKeyValueServer`, we're able to implement our key-value gRPC server. This is demonstrated with the following code, in which we implement the `Get` method. The `Put` and `Delete` methods are omitted for brevity.

```
package main

import (
    "context"
    "log"
```

```

"net"

pb "github.com/cloud-native-go/ch08/keyvalue"
"google.golang.org/grpc"
)

// server is used to implement KeyValueServer. It MUST embed the generated
// struct pb.UnimplementedKeyValueServer
type server struct {
    pb.UnimplementedKeyValueServer
}

func (s *server) Get(ctx context.Context, r *pb.GetRequest)
    (*pb.GetResponse, error) {

    log.Printf("Received GET key=%v", r.Key)

    // The local Get function is implemented back in Chapter 5
    value, err := Get(r.Key)

    // Return expects a GetResponse pointer and an err
    return &pb.GetResponse{Value: value}, err
}

func main() {
    // Create a gRPC server and register our KeyValueServer with it
    s := grpc.NewServer()
    pb.RegisterKeyValueServer(s, &server{})

    // Open a listening port on 50051
    lis, err := net.Listen("tcp", ":50051")
    if err != nil {
        log.Fatalf("failed to listen: %v", err)
    }

    // Start accepting connections on the listening port
    if err := s.Serve(lis); err != nil {
        log.Fatalf("failed to serve: %v", err)
    }
}

```

In the above code, we implement and start our service in four steps:

1. **Create the server struct.** Our server struct embeds `pb.UnimplementedKeyValueServer`. This is not optional: gRPC requires your server struct to similarly embed its generated `UnimplementedXXXServer`.
2. **Implement the service methods.** We implement the service methods defined in the generated `pb.KeyValueServer` interface. Interestingly, because the `pb.UnimplementedKeyValueServer` includes stubs for all of these service methods, we don't have to implement them all right away.
3. **Register our gRPC server.** In the `main` function, we create a new instance of the server struct and register it with the gRPC framework. This is similar to how we registered handler functions in "[Building an HTTP Server With net/http](#)", except we register an entire instance rather than individual functions.
4. **Start accepting connections.** Finally, we open a listening port<sup>7</sup> using `net.Listen`, which we pass to the gRPC framework via `s.Serve` to begin listening.

It could be argued that gRPC provides the best of both worlds by providing the freedom to implement any desired functionality without having to be concerned with building many of the tests and checks usually associated with a RESTful service.

## Implementing the gRPC Client

Because all of the client code is generated, making use of a gRPC client is fairly straight forward.

The generated client interface will be named `XXXClient`, which in our case will be `KeyValueClient`, shown below.

```
type KeyValueClient interface {
    Get(ctx context.Context, in *GetRequest, opts ...grpc.CallOption)
```

```

        (*GetResponse, error)

    Put(ctx context.Context, in *PutRequest, opts ...grpc.CallOption)
        (*PutResponse, error)

    Delete(ctx context.Context, in *DeleteRequest, opts
...grpc.CallOption)
        (*PutResponse, error)
}

```

All of the methods described in our source .proto file are specified here, each accepting a request type pointer, and returning a response type pointer and an error.

Additionally, each of the methods accepts a context.Context (if you're rusty on what this is or how it's used, take a look at [“The Context Package”](#)), and zero or more instances of grpc.CallOption. CallOption is used to modify the behavior of the client when it executes its calls. More detail can be found [in the gRPC API documentation](#).

We demonstrate how to create and use a gRPC client below:

```

package main

import (
    "context"
    "log"
    "os"
    "strings"
    "time"

    pb "github.com/cloud-native-go/ch08/keyvalue"
    "google.golang.org/grpc"
)

func main() {
    // Set up a connection to the gRPC server
    conn, err := grpc.Dial("localhost:50051",
        grpc.WithInsecure(), grpc.WithBlock(),
    grpc.WithTimeout(time.Second))
    if err != nil {

```

```
        log.Fatalf("did not connect: %v", err)
    }
    defer conn.Close()

    // Get a new instance of our client
    client := pb.NewKeyValueClient(conn)

    var action, key, value string

    // Expect something like "set foo bar"
    if len(os.Args) > 2 {
        action, key = os.Args[1], os.Args[2]
        value = strings.Join(os.Args[3:], " ")
    }

    // Use context to establish a 1-second timeout.
    ctx, cancel := context.WithTimeout(context.Background(),
time.Second)
    defer cancel()

    // Call client.Get() or client.Put() as appropriate.
    switch action {
    case "get":
        r, err := client.Get(ctx, &pb.GetRequest{Key: key})
        if err != nil {
            log.Fatalf("could not get value for key %s: %v\n", key,
err)
        }
        log.Printf("Get %s returns: %s", key, r.Value)

    case "put":
        _, err := client.Put(ctx, &pb.PutRequest{Key: key, Value:
value})
        if err != nil {
            log.Fatalf("could not put key %s: %v\n", key, err)
        }
        log.Printf("Put %s", key)

    default:
        log.Fatalf("Syntax: go run [get|put] KEY VALUE...")
    }
}
```

The above example parses command line values to determine whether it should do a Get or a Put operation.

First, it establishes a connection with the gRPC server using the `grpc.Dial` function, which takes a target address string, and one or more `grpc.DialOption` arguments that configure how the connection gets set up. In our case we use:

- `WithInsecure`, which disables transport security for this `ClientConn`. *Don't use insecure connections in production.*
- `WithBlock`, which makes `Dial` a block until a connection is established, otherwise the connection will occur in the background.
- `WithTimeout`, which makes a blocking `Dial` throw an error if it takes longer than the specified amount of time.

Next, it uses `NewKeyValueClient` to get a new `KeyValueClient`, and gets the various command line arguments.

Finally, based on the `action` value, we call either `client.Get` or `client.Put`, both of which return an appropriate return type and an error.

Once again, these functions look and feel exactly like local function calls. No checking status codes, hand-building our own clients, or any other funny business.

## Loose Coupling Local Resources With Plugins

At first glance, the topic of loose coupling of local — as opposed to remote or distributed — resources might seem mostly irrelevant to a discussion of “cloud native” technologies. But you might be surprised how often such patterns come in handy.

For example, it’s often useful to build services or tools that can accept data from different kinds of input sources (such as REST

interface, a gRPC interface, and a chatbot interface) or generate different kinds of outputs (such as generating different kinds of logging or metric formats). As an added bonus, designs that support such modularity can also make mocking resources for testing dead simple.

As we'll see in "[Hexagonal Architecture](#)", entire software architectures have even been built around this concept.

No discussion of loose coupling would be complete without a review of plugin technologies.

## In-Process Plugins With the `plugin` Package

Go provides a native plugin system in the form of the standard [plugin package](#). This package is used to open and access Go plugins, but it's not necessary to actually build the plugins themselves.

As we'll demonstrate below, the requirements for building and using a Go plugin are pretty minimal. It doesn't have to even know it's a plugin or even import the `plugin` package. A Go plugin has three real requirements: it must be in the `main` package, it must export one or more functions or variables, and it must be compiled using the `-buildmode=plugin` build flag. That's it, really.

## GO PLUGIN CAVEATS

Before we get too deep into the subject of Go plugins, it's important to mention some caveats up front.

1. As of Go version 1.14.6, Go plugins are only supported on Linux, FreeBSD, and MacOS.
2. The version of Go used to build a plugin must match the program that's using it *exactly*. Plugins built with Go 1.14.5 won't work with Go 1.14.6.
3. Similarly, the versions of any packages used by both the plugin and program must also match *exactly*.
4. Finally, building plugins forces `CGO_ENABLED`, making cross-compiling more complicated.

These conditions make Go plugins most appropriate when you're building plugins for use within the same codebase, but it adds considerable obstacles for creating distributable plugins.

## Plugin Vocabulary

Before we continue we need to define a few terms that are particular to plugins. Each of the following describes a specific plugin concept, and each has a corresponding type or function implementation in the plugin package. We'll go into all of these in more detail in our example.

### *Plugin*

A *plugin* is a Go `main` package with one or more exported functions and variables that has been built with the `-buildmode=plugin` build flag. It's represented in the `plugin` package by the `Plugin` type.

### *Open*

*Opening* a plugin is the process of loading it into memory, validating it, and discovering its exposed symbols. A plugin at a known location in the file system can be opened using the `Open` function, which returns a `*Plugin` value.

```
func Open(path string) (*Plugin, error)
```

## *Symbol*

A plugin *symbol* is any variable or function that's exported by the plugin's package. Symbols can be retrieved by "looking them up", and are represented in the plugin package by the `Symbol` type.

```
type Symbol interface{}
```

## *Look up*

*Looking up* describes the process of searching for and retrieving a symbol exposed by a plugin. The `plugin` package's `Lookup` method provides that functionality, and returns a `Symbol` value.

```
func (p *Plugin) Lookup(symName string) (Symbol, error)
```

In the next section we present a toy example that demonstrates how these resources are used, and dig into a little detail in the process.

## A Toy Plugin Example

You can only learn so much from a review of the API, even one as minimal as the `plugin` package. So let's build ourselves a toy example: a program that tells you about various animals<sup>8</sup>, as implemented by plugins.

For this example we'll be creating three independent packages with the following package structure:

```
~/cloud-native-go/ch08/go-plugin
└── duck
    └── duck.go
└── frog
    └── frog.go
└── main
    └── main.go
```

The `duck/duck.go` and `frog/frog.go` files each contain the source code for one plugin. The `main/main.go` file contains our example's `main` function, which will load and use the plugins we'll generate by building `frog.go` and `duck.go`.

The complete source code for this example is available in [this book's companion GitHub repository](#).

## The Sayer Interface

In order for a plugin to be useful, the functions that access it need to know what symbols to look up and what contract those symbols conform to.

One convenient — but by no means required — way to do this is to use an interface that a symbol can be expected to satisfy. In our particular implementation our plugins will expose just one symbol — `Animal` — which we'll expect to conform to the below `Sayer` interface.

```
type Sayer interface {
    Says() string
}
```

This interface describes only one method, `Says`, which returns a string that says what an animal says.

## The Go Plugin Code

We have source for two separate plugins in `duck/duck.go` and `frog/frog.go`. The first of these, `duck/duck.go`, is shown in its

entirety below, and displays all of the requirements of a plugin implementation:

```
package main

type duck struct{}

func (d duck) Says() string {
    return "quack!"
}

// Animal is exported as a symbol.
var Animal duck
```

As described in the introduction to this section, the requirements for Go plugin are really, really minimal: it just has to be a `main` package that exports one or more variables or functions.

The above plugin code describes and exports just one feature — `Animal` — that satisfies the above Sayer interface. Recall that exported package variables and symbols are exposed on the plugin as shared library symbols that can be looked up later. In this case, our code will have to look specifically for the exported `Animal` symbol.

In this example we have only one symbol, but there's no explicit limit to the number of symbols we can have. We could export many more features, if we wanted to.

We won't show the `frog/frog.go` file here because it's essentially the same. But it's important to know that the internals of a plugin don't matter as long as it satisfies the expectations of its consumer. In our case, these are that:

- The plugin exposes a symbol named `Animal`.
- The `Animal` symbol adheres to the contract defined by the Sayer interface.

## Building the Plugins

Building a Go plugin is very similar to building any other Go main package, except that you have to include the `-buildmode=plugin` build parameter.

To build our `duck/duck.go` plugin code, we do the following:

```
$ go build -buildmode=plugin -o duck/duck.so duck/duck.go
```

The result is a shared object (`.so`) file in ELF (Executable Linkable Format) format.

```
$ file duck/duck.so
duck/duck.so: Mach-O 64-bit dynamically linked shared library x86_64
```

ELF files are commonly used for plugins because once they're loaded into memory by the kernel they expose symbols in a way that allows for easy discovery and access.

## Using Our Go Plugins

Now that we've built our plugins, which are patiently sitting there with their `.so` extensions, we need to write some code that'll load and use them.

Note that even though we have our plugins fully built and in place, we haven't had to reach for the `plugin` package yet. However, now that we want to actually use our plugins, we get to change that now.

The process of finding, opening, and consuming a plugin requires several steps, which we demonstrate below.

### *Import the plugin Package*

First things first: we have to import the `plugin` package, which will provide us the tools we need to open and access our plugins.

In this example, we import four packages: `fmt`, `log`, `os`, and most relevant to this example, `plugin`.

```
import (
    "fmt"
    "log"
    "os"
    "plugin"
)
```

## *Find Our Plugin*

To load a plugin, we have to find its relative or absolute file path. For this reason, plugin binaries are usually named according to some pattern and placed somewhere where they can be easily discovered, like the user's command path or other standard fixed location.

For simplicity, our implementation assumes that our plugin has the same name as the user's chosen animal and lives in a path relative to the execution location.

```
if len(os.Args) != 2 {
    log.Fatal("usage: run main/main.go animal")
}

// Get the animal name, and build the path where we expect to
// find the corresponding shared object (.so) file.
name := os.Args[1]
module := fmt.Sprintf("./%s/%s.so", name, name)
```

Importantly, this approach means that our plugin doesn't need to be known—or even exist—at compile time. In this manner, we're able to implement whatever plugins we want at any time, and load and access them dynamically as we see fit.

## *Open Our Plugin*

Now that we think we know our plugin's path we can use the `Open` function to “open” it, loading it into memory and discovering its available symbols. The `Open` function returns a `*Plugin` value that can then be used to look up any symbols exposed by the plugin.

```
// Open our plugin and get a *plugin.Plugin.  
p, err := plugin.Open(module)  
if err != nil {  
    log.Fatal(err)  
}
```

When a plugin is first opened by the `Open` function, the `init` functions of all packages that aren't already part of the program are called. The package's `main` function is *not* run.

When a plugin is opened, a single canonical `*Plugin` value representation of it is loaded into memory. If a particular path has already been opened, subsequent calls to `Open` will return the same `*Plugin` value.

A plugin can't be loaded more than once, and can't be closed.

### *Look up your symbol*

To retrieve a variable or function exported by our package — and therefore exposed as a symbol by the plugin — we have to use the `Lookup` method to find it. Unfortunately, the `plugin` package doesn't provide any way to list all of the symbols exposed by a plugin, so you we to know the name of our symbol ahead of time.

```
// Lookup searches for a symbol named "Animal" in plugin p.  
symbol, err := p.Lookup("Animal")  
if err != nil {  
    log.Fatal(err)  
}
```

If the symbol exists in the plugin `p`, then `Lookup` returns a `Symbol` value. If the symbol doesn't exist in `p`, then a non-nil error is returned instead.

### *Assert and Use Your Symbol*

Now that we have our `Symbol`, we can convert it into the form we need and use it however we want. To make things nice and easy for

us, the `Symbol` type is essentially a re-branded `interface{}` value. From the plugin source code:

```
type Symbol interface{}
```

This means that as long as we know what our symbol's type is, we can use type assertion to coerce it into a concrete type value that can be used however we see fit.

```
// Asserts that the symbol interface holds a Sayer.  
animal, ok := symbol.(Sayer)  
if !ok {  
    log.Fatal("that's not a Sayer")  
}  
  
// Now we can use our loaded plugin!  
fmt.Printf("A %s says: %q\n", name, animal.Says())
```

In the above code, we assert that the `symbol` value satisfies the `Sayer` interface. If it does, we print what our animal says. If it doesn't we're able to exit gracefully.

## Executing Our Example

Now that we've written our main code that attempts to open and access the plugin, we can run it like any other Go `main` package, passing the animal name in the arguments.

```
$ go run main/main.go duck  
A duck says: "quack!"  
  
$ go run main/main.go frog  
A frog says: "ribbit!"
```

We can even implement arbitrary plugins later without changing our main source code.

```
$ go run main/main.go fox  
A fox says: "ring-ding-ding-ding-dingeringed!"
```

## HashiCorp's Go Plugin System Over RPC

HashiCorp's **Go plugin system** has been in wide use — both internally to HashiCorp and elsewhere — since at least 2016, predating the release of Go's standard plugin package by about a year.

Unlike Go plugins, which use shared libraries, HashiCorp's plugins are standalone processes that are executed by using `exec.Command`, which has some obvious benefits over shared libraries:

- **They can't crash your host process.** Because they're separate processes, a panic in a plugin doesn't automatically crash the plugin consumer.
- **They're more version-flexible.** Go plugins are famously version-specific. HashiCorp plugins are far less so, expecting only that plugins adhere to a contract. It also supports explicit protocol versioning.
- **They're relatively secure.** HashiCorp plugins only have access to the interfaces and parameters passed to them, as opposed to the entire memory space of the consuming process.

They do have a couple of downsides, though:

- **More verbose.** HashiCorp plugins require more boilerplate than Go plugins.
- **Lower performance.** Because all data exchange with HashiCorp plugins occurs over RPC, communication with Go plugins is generally more performant.

That being said, let's take a look at what it takes to assemble a simple plugin.

### Another Toy Plugin Example

So we can compare apples to apples, we’re going to work through a toy example that’s functionally identical to the one for the standard plugin package in “[A Toy Plugin Example](#)”: a program that tells you what various animals say.

As before, we’ll be creating several independent packages with the following structure:

```
~/cloud-native-go/ch08/hashicorp-plugin
├── commons
│   └── commons.go
├── duck
│   └── duck.go
└── main
    └── main.go
```

As before, the `duck/duck.go` file contains the source code for a plugin, and the `main/main.go` file contains our example’s `main` function that loads and uses the plugin. Because both of these are independently compiled to produce executable binaries, both files are in the `main` package.

The `commons` package is new. It contains some resources that are shared by the plugin and the consumer, including the service interface and some RPC boilerplate.

As before, the complete source code for this example is available in [this book’s companion GitHub repository](#).

## Common Code

The `commons` package contains some resources that are shared by both the plugin and the consumer, so in our example it’s imported by both the plugin and client code.

It contains the RPC stubs that are used by the underlying `net/rpc` machinery to define the service abstraction for the host and allow the plugins to construct their service implementations.

## *The Sayer Interface*

The first of these is the Sayer interface. This is our service interface, which provides the service contract that the plugin service implementations must conform to and that the host can expect.

It's identical to the interface that we used in “[The Sayer Interface](#)”:

```
type Sayer interface {
    Says() string
}
```

The Sayer interface only describes one method: Says. Although this code is shared, as long as this interface doesn't change, the shared contract will be satisfied and the degree of coupling is kept fairly low.

## *The SayerPlugin Struct*

The more complex of the common resources is the SayerPlugin struct, shown below. It's an implementation of plugin.Plugin, the primary plugin interface from the [github.com/hashicorp/go-plugin](https://github.com/hashicorp/go-plugin) package.

### WARNING

The package declaration inside the [github.com/hashicorp/go-plugin](https://github.com/hashicorp/go-plugin) repository is plugin, not go-plugin, as its path might suggest. Adjust your imports accordingly!

The Client and Server methods are used to describe our service according to the expectations of Go's standard net/rpc package. We won't cover that package in this book, but if you're interested, you can find a wealth of information in [the Go documentation](#).

```
type SayerPlugin struct {
    Impl Sayer
}
```

```

func (SayerPlugin) Client(b *plugin.MuxBroker, c *rpc.Client)
    (interface{}, error) {

    return &SayerRPC{client: c}, nil
}

func (p *SayerPlugin) Server(*plugin.MuxBroker) (interface{}, error) {
    return &SayerRPCServer{Impl: p.Implementation}, nil
}

```

Both methods accept a `plugin.MuxBroker`, but you can ignore that for now. It's used to create more multiplexed streams on a plugin connection and is a more advanced use case.

### *The SayerRPC Client Implementation*

SayerPlugin's `Client` method provides an implementation of our `Sayer` interface that communicates over an RPC client—the appropriately named `SayerRPC` struct—shown below.

```

type SayerRPC struct{ client *rpc.Client }

func (g *SayerRPC) Says() string {
    var resp string

    err := g.client.Call("Plugin.Says", new(interface{}), &resp)
    if err != nil {
        panic(err)
    }

    return resp
}

```

`SayerRPC` uses Go's RPC framework to remotely call the `Says` method implemented in the plugin. It invokes the `Call` method attached to the `*rpc.Client`, passing in any parameters (`Says` doesn't have any parameters, so we pass an empty `interface{}`) and retrieves the response, which it puts it into the `resp` string.

### *The Handshake Configuration*

`HandshakeConfig` is used by both the plugin and host to do a basic handshake between the host and the plugin. If the handshake fails — if the plugin was compiled with a different protocol version, for example — a user friendly error is shown. This prevents users from executing bad plugins or executing a plugin directly. Importantly, this is a UX feature, not a security feature.

```
var HandshakeConfig = plugin.HandshakeConfig{
    ProtocolVersion: 1,
    MagicCookieKey: "BASIC_PLUGIN",
    MagicCookieValue: "hello",
}
```

### *The SayerRPCServer Server Implementation*

`SayerPlugin`'s `Server` method provides a definition of an RPC server — the `SayerRPCServer` struct — to serve the actual methods in a way that's consistent with `net/rpc`.

```
type SayerRPCServer struct {
    Impl Sayer // Impl contains our actual implementation
}

func (s *SayerRPCServer) Says(args interface{}, resp *string) error {
    *resp = s.Impl.Says()
    return nil
}
```

`SayerRPCServer` doesn't implement the `Sayer` service. Instead, its `Says` method calls into a `Sayer` implementation — `Impl` — that we'll provide when we use this to build our plugin.

## Our Plugin Implementation

Now that we've assembled the code that's common between the host and plugins — the `Sayer` interface and the RPC stubs — we can build our plugin code. The code in this section represents the entirety of our `main/main.go` file.

Just like standard Go plugins, HashiCorp plugins are compiled into standalone executable binaries, so they must be in the `main` package. Effectively, every HashiCorp plugin is a small, self-contained RPC server.

```
package main
```

We have to import our `commons` package, as well as the `hashicorp/go-plugin` package, whose contents we'll reference as `plugin`.

```
import (
    "github.com/cloud-native-go/ch08/hashicorp-plugin/commons"
    "github.com/hashicorp/go-plugin"
)
```

In our plugins we get to build our real implementations. We can build it however we want<sup>9</sup>, as long as it conforms to the `Sayer` interface that we define in the `commons` package.

```
type Duck struct{}

func (g *Duck) Says() string {
    return "Quack!"
}
```

Finally, we get to our `main` function. It's somewhat "boilerplate-y" but it's essential:

```
func main() {
    // Create and initialize our service implementation.
    sayer := &Duck{}

    // pluginMap is the map of plugins we can dispense.
    var pluginMap = map[string]plugin.Plugin{
        "sayer": &commons.SayerPlugin{Impl: sayer},
    }

    plugin.Serve(&plugin.ServeConfig{
```

```
        HandshakeConfig: handshakeConfig,
        Plugins:           pluginMap,
    })
}
```

The `main` function does three things. First, it creates and initializes our service implementation, a `*Duck` value, in this case.

Next, it maps the service implementation to the name “sayer” in the `pluginMap`. If we wanted to, we could actually implement several plugins, listing them all here with different names.

Finally, we call `plugin.Serve`, which starts the RPC server that will handle any connections from the host process, allowing the handshake with the host to proceed and the service’s methods to be executed as the host sees fit.

## Our Host Process

Finally, we have our host process; the main command that acts as a client that finds, loads, and executes the plugin processes.

As you’ll see, using HashiCorp plugins isn’t all that different from the steps that described for Go plugins in “[Using Our Go Plugins](#)”.

### *Import the `hashicorp/go-plugin` and `commons` Packages*

A usual, we start with our package declaration and imports. The imports mostly aren’t interesting, and their necessity should be clear from examination of the code.

The two that *are* interesting (but not surprising) are `github.com/hashicorp/go-plugin`, which we once again have to reference as `plugin`, and our `commons` package, which contains the interface and handshake configuration, both of which must be agreed upon by the host and the plugins.

```
package main

import (
```

```

    "fmt"
    "log"
    "os"
    "os/exec"

    "github.com/cloud-native-go/ch08/hashicorp-plugin/commons"
    "github.com/hashicorp/go-plugin"
)

```

## *Find Our Plugin*

Since our plugin is an external file, we have to find it. Again, for simplicity, our implementation assumes that our plugin has the same name as the user's chosen animal and lives in a path relative to the execution location.

```

func main() {
    if len(os.Args) != 2 {
        log.Fatal("usage: run main/main.go animal")
    }

    // Get the animal name, and build the path where we expect to
    // find the corresponding executable file.
    name := os.Args[1]
    module := fmt.Sprintf("./%s/%s", name, name)

    // Does the file exist?
    _, err := os.Stat(module)
    if os.IsNotExist(err) {
        log.Fatal("can't find an animal named", name)
    }
}

```

It bears repeating that the value of this approach is that our plugin—and its implementation—doesn't need to known or exist at compile time. We're able to implement whatever plugins we want at any time, and use them dynamically as we see fit.

## *Create Our Plugin Client*

The first way that a HashiCorp RPC plugin differs from a Go plugin is the way that it retrieves the implementation. Where Go plugins have

to be “opened” and their symbol “looked up”, HashiCorp plugins are built on RPC, and therefore require an RPC client.

This actually requires two steps, and two clients: one client — a `*plugin.Client` — that manages the lifecycle of the plugin subprocess, and a protocol client — a `plugin.ClientProtocol` implementation — that can communicate with the plugin subprocess.

This awkward API is mostly historical, but is used to split the client that deals with subprocess management and the client that does RPC management.

```
// pluginMap is the map of plugins we can dispense.
var pluginMap = map[string]plugin.Plugin{
    "sayer": &commons.SayerPlugin{},
}

// Launch the plugin process!
client := plugin.NewClient(&plugin.ClientConfig{
    HandshakeConfig: commons.HandshakeConfig,
    Plugins:         pluginMap,
    Cmd:             exec.Command(module),
})
defer client.Kill()

// Connect to the plugin via RPC
rpcClient, err := client.Client()
if err != nil {
    log.Fatal(err)
}
```

Most of the above consists of defining the parameters of the plugin that we want in the form of a `plugin.ClientConfig`. The **complete list of available client configurations** is lengthy. This example uses only three:

- **HandshakeConfig** — The handshake configuration. This has to match the plugin’s own handshake configuration or we’ll get an error in the next step.

- **Plugins** — A map that specifies the name and type of the plugin we want.
- **Cmd** — An `*exec.Cmd` value that represents the command for starting the plugin subprocess.

With all of the configuration stuff is out of the way, we can first use `plugin.NewClient` to retrieve a `*plugin.Client` value, which we call `client`.

Once we have that, we can use `client.Client` to request a protocol client. We call this `rpcClient` because it knows how to use RPC to communicate with the plugin subprocess.

### *Connect To Our Plugin and Dispense Our Sayer*

Now that we have our protocol client, we can use it to dispense our `Sayer` implementation.

```
// Request the plugin from the client
raw, err := rpcClient.Dispense("sayer")
if err != nil {
    log.Fatal(err)
}

// We should have a Sayer now! This feels like a normal interface
// implementation, but is actually over an RPC connection.
sayer := raw.(commons.Sayer)

// Now we can use our loaded plugin!
fmt.Printf("A %s says: %q\n", name, sayer.Says())
}
```

Using the protocol client's `Dispense` function, we're able to finally retrieve our `Sayer` implementation as an `interface{}`, which we can assert as a `commons.Sayer` value and immediately use exactly like using a local value.

Under the covers, our `sayer` is in fact a `SayerRPC` value, and calls to its functions trigger RPC calls that are executed in our plugin's

address space.

## Hexagonal Architecture

*Hexagonal architecture* — also known as the “ports and adapters” pattern — is an architectural pattern that uses loose coupling and *inversion of control* as its central design philosophy to establish clear boundaries between business and peripheral logic.

In a hexagonal application, the core application doesn’t know any details at all about the outside world, operating entirely through loosely-coupled *ports* and technology-specific *adapters*.

This approach allows the application to, for example, expose different APIs (REST, gRPC, a test harness, etc.) or use different data sources (database, message queues, local files, etc.) without impacting its core logic or requiring major code changes.

### NOTE

It took me an embarrassingly long time to realize that the name “hexagonal architecture” doesn’t actually mean anything. Alistair Cockburn — the author of hexagonal architecture — chose the shape because it gave him enough room to illustrate the design.

## The Architecture

As illustrated in [Figure 8-3](#), hexagonal architecture is composed of three components conceptually arranged in and around a central hexagon.

### *The Core Application*

The application proper, represented by the hexagon. Contains all of the business logic, but has no direct reference to any technology, framework or real world device. The business logic shouldn’t depend on whether it exposes a REST or a gRPC API

or whether it gets data from a database or a CSV file. Its only view of the world should be through ports.

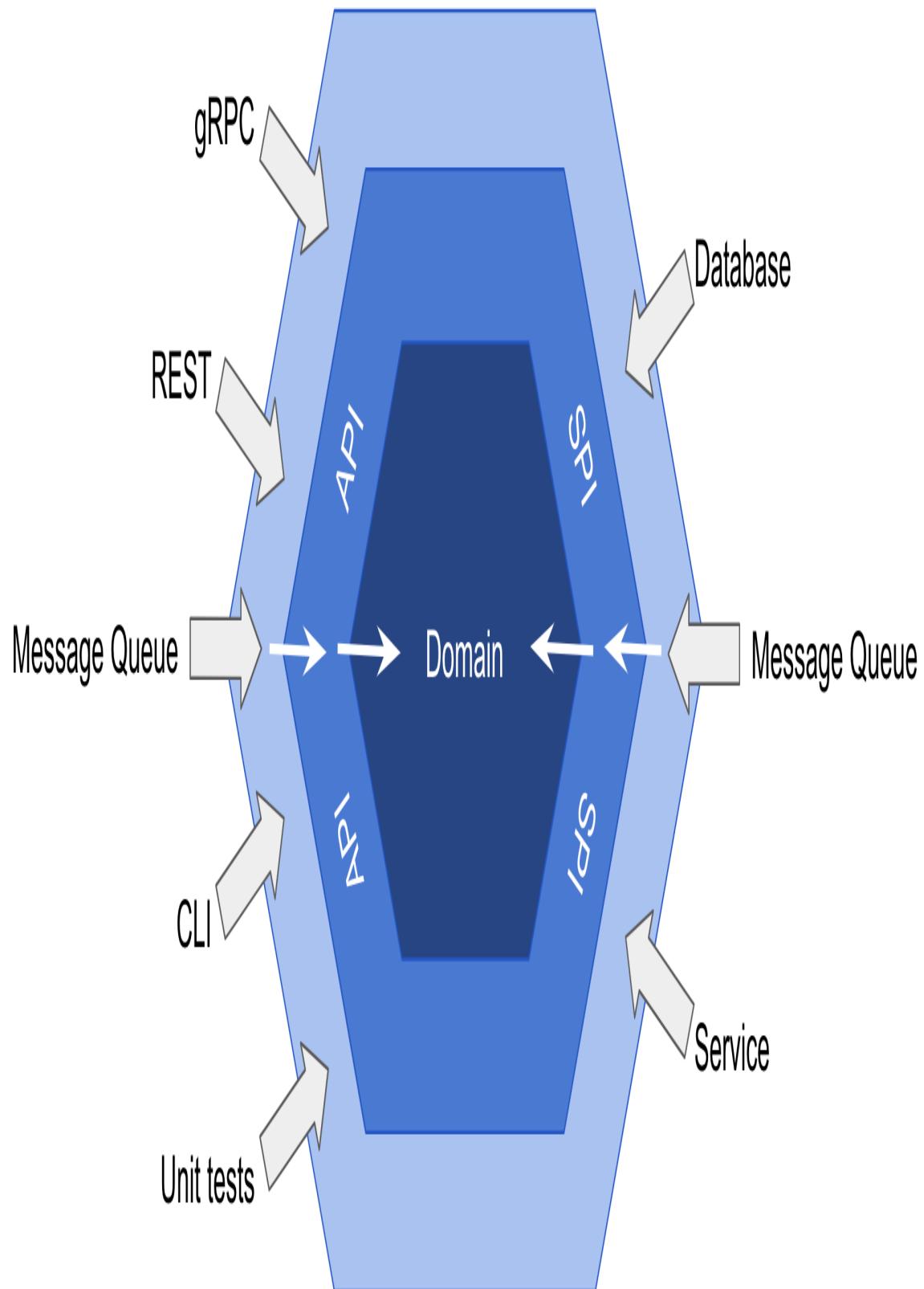
### *Ports and Adapters*

The ports and adapters are represented on the edge of the hexagon. Ports allow different kinds of actors to “plug in” and interact with the core service. Adapters can “plug into” a port and translate signals between the core application and an actor.

For example, your application might have a “data port” into which a “data adapter” might plug. One data adapter might write to a database, while another might use an in-memory datastore or automated test harness.

### *Actors*

The actors can be anything in the environment that interacts with the core application (users, upstream services, etc.) or that the core application interacts with (storage devices, downstream services, etc.). They exist outside the hexagon.



*Figure 8-3. In Hexagonal Architecture all dependencies point inward.*

In a traditional layered architecture all of the dependencies point in the same direction, each layer above depending on the one below.

In a hexagonal architecture, however, all dependencies point inward: the core business logic doesn't know any details about the outer world, the adapters know how to ferry information to and from the core, and the adapters in the outer world know how to interact with the actors.

## Example

To illustrate this, we're going to refactor our old friend the key-value store.

If you'll recall from [Chapter 5](#), the core application of our key-value store reads and writes to a in-memory map, which can be accessed via a RESTful (or gRPC) front end. Later in the same chapter we implemented a transaction logger, which knows how to write all transactions to *somewhere* and read them all back when the system restarts.

We'll reproduce important snippets of the service here, but if you want a refresher on what we did, go back and do so now.

By this point in the book, we've accumulated a couple of different implementations for a couple of different components of our service that seem like good candidates for ports and adapters in a hexagonal architecture:

### *The front end*

Back in "[Generation 1: The Monolith](#)" we implemented a REST front end, and then in "[Remote Procedure Calls With gRPC](#)" we implemented a separate gRPC front end. We can describe these with a single "driver" port into which we'll be able to plug either (or both!) as adapters.

## *The transaction logger*

In “[What’s a Transaction Log?](#)” we created two implementations of a transaction log. These seem like a natural choice for a “driven” port and adapters.

While all the logic for all of these already exist, we’ll need to do some refactoring to make this architecture “hexagonal”:

1. Our original core application — originally described in [“Generation 0: The Core Functionality”](#) — uses exclusively public functions. We’ll refactor those into struct methods to make it easier to use in a “ports and adapters” format.
2. Both the RESTful and gRPC front ends are already consistent with hexagonal architecture, since the core application doesn’t know or care about them, but they’re constructed in a `main` function. We’ll convert these into FrontEnd adapters into which we can pass our core application. This pattern is typical of a “driver” port.
3. The transaction loggers themselves won’t need much refactoring, but they’re currently embedded in the front end logic. When we refactor the core application, we’ll add a transaction logger port so that the adapter can be passed into the core logic. This pattern is typical of a “driven” port.

## *Our Refactored Components*

For the sake of this example, all of our components live under the [github.com/cloud-native-go/examples/ch08/hexarch](https://github.com/cloud-native-go/examples/ch08/hexarch) package.

```
~/cloud-native-go/ch08/hexarch/
├── core
│   └── core.go
└── frontend
    ├── grpc.go
    └── rest.go
```

```
└── main.go
└── transact
    ├── filelogger.go
    └── pglogger.go
```

- **core** — The core key-value application logic. Importantly, it has no dependencies outside of the Go standard libraries.
- **frontend** — Contains the REST and gRPC front end driver adapters. These have a dependency on **core**.
- **transact** — Contains the file and PostgreSQL transaction logger driven adapters. These also have a dependency on **core**.
- **main.go** — Makes the core application instance, into which it passes the driven components, and which it passes to the driver adapters.

The complete source code is also available in [the companion GitHub repository](#).

### *Our First Plug*

You may remember that we also implemented a *transaction log* to maintain a record of every time a resource is modified so that if our service crashes, is restarted, or otherwise finds itself in an inconsistent state, it can reconstruct its complete state by replaying the transactions.

In “[Our Transaction Logger Interface](#)” we represented a generic transaction logger with the `TransactionLogger`:

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

For brevity, we only define the `WriteDelete` and `WritePut` methods.

A common aspect of “driven” adapters is that the core logic acts *on them*, so the core application has to know about the port. As such, this code lives in the `core` package.

### *Our Core Application*

In our original implementation in “[Our Super Simple API](#)”, the transaction logger was used by the front end. In a hexagonal architecture we move the port—in the form of the `TransactionLogger` interface—into the core application.

```
package core

import (
    "errors"
    "log"
    "sync"
)

type KeyValueStore struct {
    m          map[string]string
    transact TransactionLogger
}

func NewKeyValueStore(tl TransactionLogger) *KeyValueStore {
    return &KeyValueStore{
        m:          make(map[string]string),
        transact: tl,
    }
}

func (store *KeyValueStore) Delete(key string) error {
    delete(store.m, key)
    store.transact.WriteDelete(key)
    return nil
}

func (store *KeyValueStore) Put(key string, value string) error {
    store.m[key] = value
    store.transact.WritePut(key, value)
    return nil
}
```

Comparing the above code with the original form in “[Generation 0: The Core Functionality](#)” you’ll see some significant changes.

First, Put and Delete aren’t pure functions anymore: they’re now methods on a new KeyValueStore struct, which also has the map data structure. We’ve also added a NewKeyValueStore function that initializes and returns a new KeyValueStore pointer value.

Finally, KeyValueStore now has a TransactionLogger, which Put and Delete act upon appropriately. This is our port.

### *Our TransactionLogger Adapters*

In [Chapter 5](#) we created two TransactionLogger implementations:

- In “[Implementing Our FileTransactionLogger](#)” we describe a file-based implementation.
- In “[Implementing Our PostgresTransactionLogger](#)” we describe a PostgreSQL-backed implementation.

Both of these have been moved to the `transact` package. They hardly have to change at all, except to account for the fact that the TransactionLogger interface and Event struct now live in the `core` package.

But how do we determine which one to load? Well, Go doesn’t have annotations or any fancy dependency injection features<sup>10</sup>, but there are still a couple of ways you can do this.

The first option is to use use a plugins of some kind (this is actually a primary use case for Go plugins). This might make sense if you want changing adapters to require zero code changes.

More commonly you’ll see some kind of “factory” function<sup>11</sup> that’s used by the initializing function. While this still requires code changes to add adapters, they’re isolated to a single, easily-modified location. A more sophisticated approach might use accept a parameter or configuration value to choose which adapter to use.

An example of a TransactionLogger factory function might look like the following:

```
func NewTransactionLogger(logger string) (*core.TransactionLogger, error) {
    switch logger {
    case "file":
        return NewFileTransactionLogger(os.Getenv("TLOG_FILENAME"))

    case "postgres":
        return NewPostgresTransactionLogger(
            PostgresDbParams{
                dbName: os.Getenv("TLOG_DB_HOST"),
                host: os.Getenv("TLOG_DB_DATABASE"),
                user: os.Getenv("TLOG_DB_USERNAME"),
                password: os.Getenv("TLOG_DB_PASSWORD"),
            }
        )

    case "":
        return nil, fmt.Errorf("transaction logger type not defined")

    default:
        return nil, fmt.Errorf("no such transaction logger %s", s)
    }
}
```

In this example, the NewTransactionLogger function accepts a string that specifies the desired implementation, returning either one of our implementations or an error. We use the `os.Getenv` function to retrieve the appropriate parameters from environment variables.

### *Our FrontEnd Port*

But what about our front ends? If you'll recall, we now have two front end implementations:

- In “[Generation 1: The Monolith](#)” in [Chapter 5](#) we built a RESTful interface using `net/http` and `gorilla/mux`.
- In “[Remote Procedure Calls With gRPC](#)”, earlier in this chapter, we built an RPC interface with gRPC.

Both of these implementations include a `main` function where we configure and start the service to listen for connections.

Since they're "driver" ports, we need to pass the core application to them, so let's refactor both front ends into structs according to the following interface:

```
package frontend

type FrontEnd interface {
    Start(kv *core.KeyValueStore) error
}
```

The `FrontEnd` interface serves as our "front end port", which all frontend implementations are expected to satisfy. The `Start` method accepts the core application API in the form of a `*core.KeyValueStore`, and will also include the setup logic that formerly lived in a `main` function.

Now that we have this, we can refactor both frontends so that they comply with the `FrontEnd` interface, starting with the RESTful frontend. As usual, the complete source code for this and the gRPC service refactor are available in [this book's companion GitHub repository](#).

```
package frontend

import (
    "net/http"

    "github.com/cloud-native-go/examples/ch08/hexarch/core"
    "github.com/gorilla/mux"
)

// restFrontEnd contains a reference to the core application logic,
// and complies with the contract defined by the FrontEnd interface.
type restFrontEnd struct {
    store *core.KeyValueStore
}
```

```

// keyValueDeleteHandler handles the logic for the DELETE HTTP method.
func (f *restFrontEnd) keyValueDeleteHandler(w http.ResponseWriter,
    r *http.Request) {

    vars := mux.Vars(r)
    key := vars["key"]

    err := f.store.Delete(key)
    if err != nil {
        http.Error(w, err.Error(), http.StatusInternalServerError)
        return
    }
}

// ...other handler functions omitted for brevity.

// Start includes the setup and start logic that previously
// lived in a main function.
func (f *restFrontEnd) Start(store *core.KeyValueStore) error {
    // Remember our core application reference.
    f.store = store

    r := mux.NewRouter()

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

    return http.ListenAndServe(":8080", r)
}

```

Comparing the above code to the code we produced in “[Generation 1: The Monolith](#)”, some differences stand out:

- All functions are now methods attached to a `restFrontEnd` struct.
- All calls to the core application go through the `store` value that lives in the `restFrontEnd` struct.
- Creating the router, defining the handlers, and starting the server now live in the `Start` method.

Similar changes will have been made for our gRPC front end implementation to make it consistent with the FrontEnd port.

This new arrangement makes it easier for a consumer to choose and plug in a “front end adapter”, as demonstrated below.

### *Putting It All Together*

Here, we have our `main` function, in which we plug all of the components into our application!

```
package main

import (
    "log"

    "github.com/cloud-native-go/examples/ch08/hexarch/core"
    "github.com/cloud-native-go/examples/ch08/hexarch/frontend"
    "github.com/cloud-native-go/examples/ch08/hexarch/transact"
)

func main() {
    // Create our TransactionLogger. This is an adapter that will plug
    // into the core application's TransactionLogger port.
    tl, err := transact.NewTransactionLogger(os.Getenv("TLOG_TYPE"))
    if err != nil {
        log.Fatal(err)
    }

    // Create Core and tell it which TransactionLogger to use.
    // This is an example of a "driven agent"
    store := core.NewKeyValueStore(tl)
    store.Restore()

    // Create the frontend.
    // This is an example of a "driving agent".
    fe, err := frontend.NewFrontEnd(os.Getenv("FRONTEND_TYPE"))
    if err != nil {
        log.Fatal(err)
    }

    log.Fatal(fe.Start(store))
}
```

First, we then create a transaction logger according to the environment TLOG\_TYPE. We do this first because the “transaction logger port” is “driven”, so we’ll need to provide it to the application to plug it in.

We then create our KeyValueStore value, which represents our core application functions and provides an API for ports to interact with, and provide it with any driven adapters.

Next, we create any “driver” adapters. Since these act on the core application API, we provide the API to the adapter instead of the other way around as we would with a “driven” adapter. This means we could also create multiple front ends here, if we wanted, by creating a new adapter and passing it the KeyValueStore that exposes the core application API.

Finally, we call Start on our frontend, which instructs it to start listening for connections. At last, we have a complete hexagonal service!

## Summary

We covered a lot of ground in this chapter, but really only scratched the surface of all the different ways that components can find themselves tightly coupled and all the different ways of managing each of those.

In the first half of the chapter, we focused the coupling that can result from how services communicate. We talked about the problems caused by fragile exchange protocols like SOAP, and demonstrated REST and gRPC, which are less fragile because they can be changed to some degree without necessarily forcing client upgrades. We also touched on coupling “in time”, in which one service implicitly expects a timely response from another, and how publish-subscribe messaging might be used to relieve this.

In the second half we addressed some of the ways that systems can minimize coupling to local resources. After all, even distributed services are just programs, subject to the same limitations of the architectures and implementations as any program. Plugins implementations and hexagonal architectures are two ways of doing this by enforcing separation of concerns and inversion of control.

Unfortunately, we didn't get to drill down into some other fascinating topics like service discovery, but sadly I had to draw a line somewhere before this subject got away from me even more than it did.

---

<sup>1</sup> Ullman, Ellen. "The Dumbing-down of Programming." *Salon*, Salon.com, 12 May 1998.

<sup>2</sup> Get off my lawn.

<sup>3</sup> In XML, no less. We didn't know any better at the time.

<sup>4</sup> At Google, even the acronyms are recursive.

<sup>5</sup> This is actually a pretty nuanced discussion. See "[Service Architectures](#)".

<sup>6</sup> If you're into that kind of thing.

<sup>7</sup> If you wanted to be creative, this could be a [FileListener](#), or even a stdio stream.

<sup>8</sup> Yes, I know the animal thing has been done before. Sue me.

<sup>9</sup> "So naturally we built a duck. Obviously."

<sup>10</sup> Good riddance.

<sup>11</sup> I'm sorry.

# Chapter 9. Resilience

---

## A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 9th chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at [ablevins@oreilly.com](mailto:ablevins@oreilly.com).

*A distributed system is one in which the failure of a computer you didn’t even know about can render your own computer unusable<sup>1</sup>.*

—Leslie Lamport, DEC SRC Bulletin Board (May 1987)

Late one September night, at just after two in the morning, a portion of Amazon’s internal network quietly stopped working<sup>2</sup>. This event was brief, and not particularly interesting, except that it happened to impact a sizable number of the servers that supported the DynamoDB service.

Most days, this wouldn’t be such a big deal. Any affected servers would just try to reconnect to the cluster by retrieving their membership data from a dedicated metadata service. If that failed, they would temporarily take themselves offline and try again.

But this time, when the network was restored, a small army of storage servers simultaneously requested their membership data from the metadata service, overwhelming it so that requests — even ones from previously unaffected servers — started to time out. Storage servers dutifully responded to the timeouts by taking themselves offline and retrying (again), further stressing the metadata service, causing even more servers to go offline, and so on. Within minutes the outage had spread to the entire cluster. The service was effectively down, taking a number of dependent services down with it.

To make matters worse, the sheer volume of retry attempts — a “retry storm” — put such a burden on the metadata service that it even became entirely unresponsive to requests to add capacity. The on-call engineers were forced to explicitly block requests to the metadata service just to relieve enough pressure to allow them to manually scale up.

Finally, nearly five hours after the initial network hiccup that triggered the incident, normal operations resumed, putting an end to what must have been a long night for all involved.

## Keeping On Ticking: Why Resilience Matters

So, what was the root cause of Amazon’s outage? Was it the network disruption? Was it the storage servers’ enthusiastic retry behavior? Was it the metadata service’s response time, or maybe its limited capacity?

Clearly, what happened that early morning didn’t have a single root cause. Failures in complex systems never do<sup>3</sup>. Rather, the system failed as complex systems do: with a failure in a subsystem, which triggered a latent fault in another subsystem causing *it* to fail, followed by another, and another, until eventually the entire system went down. What’s interesting, though, is that if any of the components in our story — the network, the storage servers, the

metadata service — had been able to isolate and recover from failures elsewhere in the system, the overall system likely would have recovered without human intervention.

Unfortunately, this is just one example of a common pattern. Complex systems fail in complex (and often surprising) ways, but they don't fail all at once: they fail one subsystem at a time. For this reason resilience patterns in complex systems take the form of bulwarks and safety valves that work to isolate failures at component boundaries. Frequently, a failure contained is a failure avoided.

This property, the measure of a system's ability to withstand and recover from errors and failures, is its *resilience*. A system can be considered *resilient* if it can continue operating correctly — possibly at a reduced level — rather than failing completely when one of its subsystems fails.

## RESILIENCE IS NOT RELIABILITY

The terms *resilience* and *reliability* describe closely-related concepts, and are often confused. But they aren't quite the same thing<sup>4</sup>.

- The *resilience* of a system is the degree to which it can continue to operate correctly in the face of errors and faults. Resilience, along with the other four cloud-native properties, is just one factor that contributes to reliability.
- The *reliability* of a system is its ability to behave as expected for a given time interval. Reliability, in conjunction with attributes like availability and maintainability, contributes to a system's overall dependability.

## What Does It Mean for a System to Fail?

*For want of a nail the shoe was lost,  
for want of a shoe the horse was lost;  
for want of a horse the rider was lost;  
all for want of care about a horse-shoe nail.*

—Benjamin Franklin, *The Way to Wealth* (1758)

If we want to know what it means for a system to fail, we first have to ask what a “system” is.

This is important. Bear with me.

By definition, a *system* is a set of components that work together to accomplish an overall goal. So far so good. But here’s the important part: each component of a system — a *subsystem* — is also a complete system unto itself that in turn is composed of still smaller subsystems, and so on, and so on.

Take a car, for example. Its engine is one of dozens of subsystems, but it — like all the others — is also a very complex system with a number of subsystems of its own, including a cooling subsystem, which includes a thermostat, which includes a temperature switch, and so on. That’s just some of thousands of components and sub-components and sub-sub-components. It’s enough to make the mind spin; so many things that can fail. But what happens when they do?

As we mentioned earlier — and discussed in some depth in [Chapter 6](#) — failures of complex systems don’t just happen all at once. They unravel in predictable steps:

1. All systems contain *faults*, which we lovingly refer to as “bugs” in the software world. A tendency for a temperature switch in a car engine to stick would be a fault. So would the metadata service’s limited capacity and the storage server’s retry behavior in the DynamoDB case study<sup>5</sup>. Under the right conditions, a fault can be exercised to produce an *error*.

2. An *error* is any discrepancy between the system's intended and actual behavior. Many errors can be caught and handled appropriately, but if they're not they can — singly or in accumulation — give rise to a *failure*. A stuck temperature switch in a car engine's thermostat is an error.
3. Finally, a system can be said to be experiencing a *failure* when it's no longer able to provide correct service<sup>6</sup>. A temperature switch that no longer responds to high temperatures can be said to have failed. A failure at the subsystem level becomes a fault at the system level.

This last bit bears repeating: *a failure at the subsystem level becomes a fault at the system level*. A stuck temperature switch causes a thermostat to fail, preventing coolant from flowing through the radiator, raising the temperature of the engine, causing it to stall and the car to stop<sup>7</sup>.

That's how systems fail. It starts with the failure of one component — one subsystem — which causes an error in one or more components that interact with it, and ones that interact with that, and so on, propagating upward until the entire system fails.

This isn't just academic. Knowing how complex systems fail — one component at a time — makes the means of resisting failures more clear: if a fault can be contained before it propagates all the way to the system level, the system may be able to recover (or at least fail on its own terms).

## Building For Resilience

In a perfect world it would be possible to rid a system of every possible fault, but this isn't realistic, and it's wasteful and unproductive to try. By instead assuming that all components are destined to fail eventually — which they absolutely are — and designing them to respond gracefully to errors when they do occur,

you can produce a system that's functionally healthy even when some of its components are not.

There are lots of ways to increase the resiliency of a system. Redundancy, such as deploying multiple components of the same type, is probably the most common approach. Specialized logic like circuit breakers and request throttles can be used to isolate specific kinds of errors, preventing them from propagating. Faulty components can even be reaped — or intentionally allowed to fail — to benefit the health of the larger system.

Resilience is a particularly rich subject. We'll explore several of these approaches — and more — over the remainder of the chapter.

## Cascading Failures

The reason the DynamoDB case study is so appropriate is that it demonstrates so many different ways that things that can go wrong at scale.

Take, for example, how the failure of a group of storage servers caused requests to the metadata service to time out, which in turn caused more storage servers to fail, which increased the pressure on the metadata service, and so on. This is an excellent example of a particular — and particularly common — failure mode known as a *cascading failure*. Once a cascading failure has begun, it tends to spread very quickly, often on the order of a few minutes.

The mechanisms of cascading failures can vary a bit, but one thing they share is some kind of positive feedback mechanism. One part of a system experiences a local failure — a reduction in capacity, an increase in latency, etc. — that causes other components to attempt to compensate for the failed component in a way that exacerbates the problem, eventually leading to the failure of the entire system.

The classic cause of cascading failures is overload, illustrated in [Figure 9-1](#). This occurs when one or more nodes in a set fails,

causing the load to be catastrophically re-distributed to the survivors. The increase in load overloads the remaining nodes, causing them to fail from resource exhaustion, taking the entire system down.

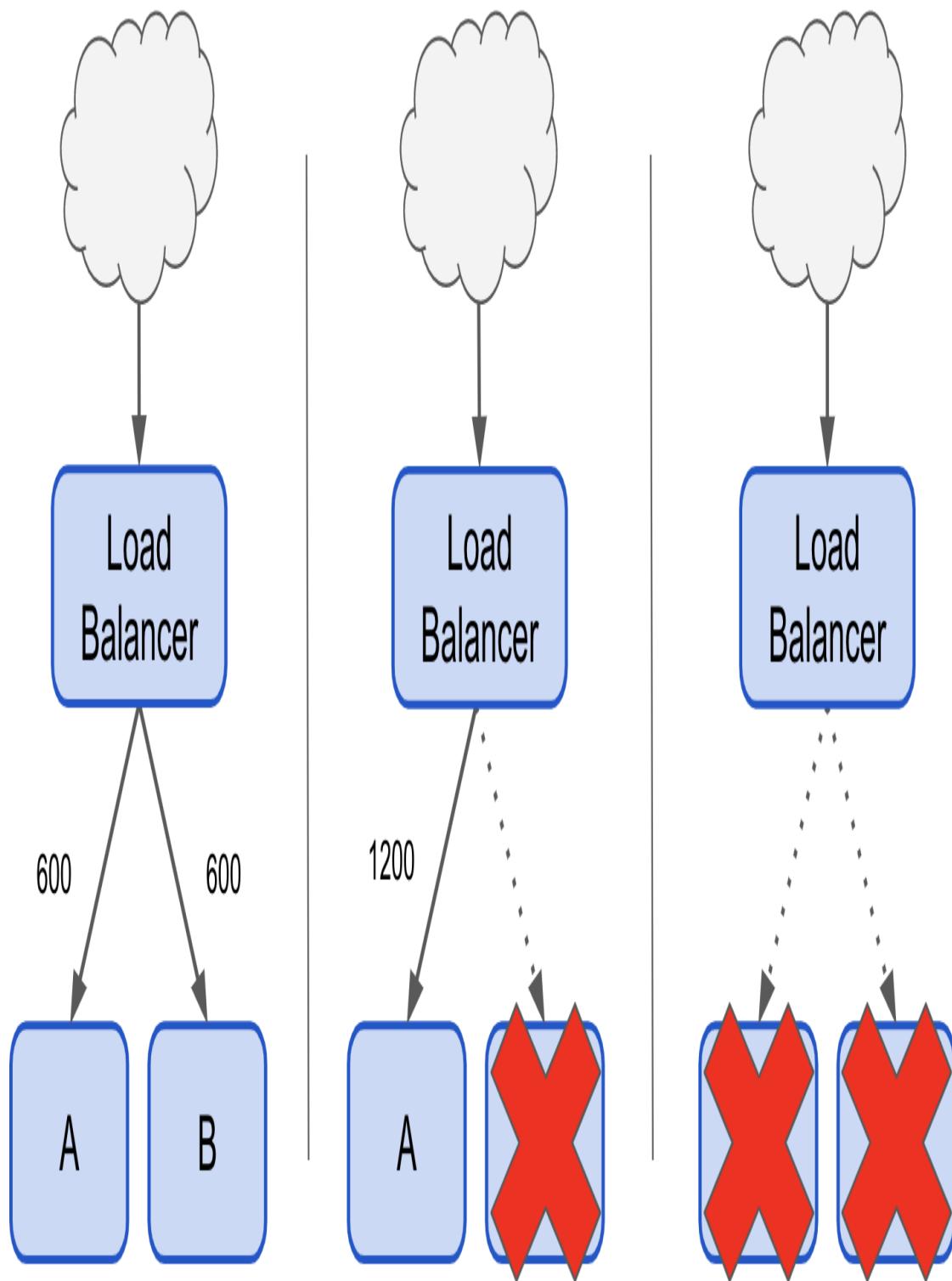


Figure 9-1. Server overload is a common cause of cascade failures. Each server handles 600 requests per second. When server B fails, server A is overloaded and

*also fails.*

The nature of positive feedback often makes it very difficult to scale your way out of a cascading failure by adding more capacity. New nodes can be overwhelmed as quickly as they come online, often contributing the feedback that took the system down in the first place. Sometimes, the only fix is to take your entire service down — perhaps by explicitly blocking the problematic traffic — in order to recover, and then slowly reintroduce load.

But how do you prevent cascading failures in the first place? This will be the subject of the next section (and, to some extent, most of this chapter).

## Preventing Overload

Every service, however well-designed and implemented, has its functional limitations. This is particularly evident in services intended to handle and respond to client requests<sup>8</sup>. For any such service there exists some request frequency, a threshold beyond which bad things will start to happen. So, how do we keep a large number of requests from accidentally (or intentionally!) bringing our service down?

Ultimately, a service that finds itself in such a situation has no choice but to reject — partially or entirely — some number of requests. There are two main strategies for doing this:

*Throttling* is a relatively straight-forward strategy that kicks in when requests come in faster than some pre-determined frequency, typically by just refusing to handle them. This is often used as a preventative measure by ensuring that no particular user consumes more resources than they would reasonably require.

*Load shedding* is a little more adaptive. Services using this strategy intentionally drop (“shed”) some proportion of load as they approach

overload conditions by either refusing requests or falling back into a degraded mode.

These strategies aren't mutually exclusive; a service may choose to employ either or both of them, according to its needs.

## Throttling

As we discussed in [Chapter 4](#), a throttle pattern works a lot like the throttle in a car, except that instead of limiting the amount of fuel entering an engine, it limits the number of requests that a user (human or otherwise) can make to a service in a set period of time.

The general-purpose throttle example that we provided in [“Throttle”](#) was relatively simple, and effectively global, at least as written.

However, throttles are also frequently applied on a per-user basis to provide something like a usage quota, so that no one caller can consume too much of a service's resources.

Below, we demonstrate a throttle implementation that, while still using a token bucket<sup>9</sup>, is otherwise quite different in several ways.

First, instead of having a single bucket that's used to gate all incoming requests, the implementation below throttles on a per-user basis, returning a function that accepts a “key” parameter, that's meant to represent a username or some other unique identifier.

Second, rather than attempting to “replay” a cached value when imposing a throttle limit, the returned function returns a boolean that indicates when a throttle has been imposed. Note that the throttle doesn't return an `error` when it's activated: throttling isn't an error condition, so we don't treat it as one.

Finally, and perhaps most interestingly, it doesn't actually use a timer (`a time.Ticker`) to explicitly add tokens to buckets on some regular cadence. Rather, it refills buckets on demand, based on the time elapsed between requests. This strategy means that we don't have to dedicate background processes to filling buckets until they're actually used, which will scale much more effectively.

```

// Effector is the function that you want to subject to throttling.
type Effector func(context.Context) (string, error)

// Throttled wraps an Effector. It accepts the same parameters, plus a
// "UID" string that represents a caller identity. It returns the
same,
// plus a bool that's true if the call is not throttled.
type Throttled func(context.Context, string) (bool, string, error)

// A bucket tracks the requests associated with a UID.
type bucket struct {
    tokens uint
    time   time.Time
}

// Throttle accepts an Effector function, and returns a Throttled
// function with a per-UID token bucket with a capacity of max
// that refills at a rate of refill tokens every d.
func Throttle(e Effector, max uint, refill uint, d time.Duration)
Throttled {
    // buckets maps UIDs to specific buckets
    buckets := map[string]*bucket{}

    return func(ctx context.Context, uid string) (bool, string,
error) {
        b := buckets[uid]

        // This is a new entry! It passes. Assumes that capacity >= 1.
        if b == nil {
            buckets[uid] = &bucket{tokens: max - 1, time: time.Now()}

            str, err := e(ctx)
            return true, str, err
        }

        // Calculate how many tokens we now have based on the time
// passed since the previous request.
        refillInterval := uint(time.Since(b.time) / d)
        tokensAdded := refill * refillInterval
        currentTokens := b.tokens + tokensAdded

        // We don't have enough tokens. Return false.
        if currentTokens < 1 {
            return false, "", nil
        }
    }
}

```

```

        // If we've refilled our bucket, we can restart the clock.
        // Otherwise, we figure out when the most recent tokens were
added.
        if currentTokens > max {
            b.time = time.Now()
            b.tokens = max - 1
        } else {
            deltaTokens := currentTokens - b.tokens
            deltaRefills := deltaTokens / refill
            deltaTime := time.Duration(deltaRefills) * d

            b.time = b.time.Add(deltaTime)
            b.tokens = currentTokens - 1
        }

        str, err := e(ctx)

        return true, str, err
    }
}

```

Like the example in “[Throttle](#)”, this Throttle function accepts a function literal that conforms to the Effector contract, plus some values that define the size and refill rate of the underlying token bucket.

Instead of returning another Effector, however, it returns a Throttled function, which in addition to wrapping the effector with the throttling logic adds a “key” input parameter, which represents a unique user identifier, and a boolean return value, which indicates whether the function has been throttled (and therefore not executed).

As interesting as you may (or may not) find the Throttle code, it’s still not production ready. First of all, it’s not entirely safe for concurrent use. A production implementation will probably want lock on the record values, and possibly the bucket map. Second, there’s no way to purge old records. In production, we’d probably want to use something like an LRU cache, like the one we described in “[Using an LRU Cache](#)”, instead.

Below, we show a toy example of how Throttle might be used in a RESTful web service.

```
var throttled = Throttle(getHostname, 1, 1, time.Second)

func getHostname(ctx context.Context) (string, error) {
    if ctx.Err() != nil {
        return "", ctx.Err()
    }

    return os.Hostname()
}

func throttledHandler(w http.ResponseWriter, r *http.Request) {
    ok, hostname, err := throttled(r.Context(), r.RemoteAddr)

    if err != nil {
        http.Error(w, err.Error(), http.StatusInternalServerError)
        return
    }

    if !ok {
        http.Error(w, "Too many requests", http.StatusTooManyRequests)
        return
    }

    w.WriteHeader(http.StatusOK)
    w.Write([]byte(hostname))
}

func main() {
    r := mux.NewRouter()
    r.HandleFunc("/hostname", throttledHandler)
    log.Fatal(http.ListenAndServe(":8080", r))
}
```

The above code creates a small web service with a single (somewhat contrived) endpoint at /hostname that does returns the service's hostname. When the program is run, the throttled var is created by wrapping the getHostname function — which provides the actual service logic — by passing it to Throttle, which we define above.

When the router receives a request for the `/hostname` endpoint, the request is forwarded to the `throttledHandler` function, which performs the calls to `throttled`, receiving a `bool` indicating throttling status, the hostname string, and an `error` value. A defined error causes us to return a `500 Internal Server Error`, and a throttled request gets a `429 Too Many Requests`. If all else goes well, we return the hostname and a status `200 OK`.

Note that the bucket values are stored locally, so this implementation can't really be considered production ready either. If you want this to scale out, you might want to store to record values in an external cache of some kind so that multiple service replicas can share them.

## Load Shedding

It's an unavoidable fact of life that as load on a server increases beyond what it can handle, something eventually has to give.

*Load shedding* is a technique used to predict when a server is approaching that saturation point and mitigate it by dropping some proportion of traffic in a controlled fashion. Ideally, this will prevent the server from overloading and failing health checks, serving with high latency, or just collapsing in a graceless, uncontrolled failure.

Unlike quota-based throttling, load shedding is reactive, typically engaging in response to depletion of a resource like CPU, memory, or request queue depth.

Perhaps the most straight-forward form of load-shedding is a per-task throttling that drops requests when one or more resources exceed a particular threshold. For example, if your service provides a RESTful endpoint, you might choose to return an HTTP 503 (service unavailable). The `gorilla/mux` web toolkit, which we found very effective in [Chapter 5](#) in the section "[Building an HTTP Server With gorilla/mux](#)", makes this fairly straight-forward by [supporting "middleware" handler functions](#) that are called on every request:

```

const MaxQueueDepth = 1000

// Middleware function, which will be called for each request.
// If queue depth is exceeded, it returns HTTP 503 (service
// unavailable).
func loadSheddingMiddleware(next http.Handler) http.Handler {
    return http.HandlerFunc(func(w http.ResponseWriter, r
    *http.Request) {
        // CurrentQueueDepth is fictional and for example purposes
        // only.
        if CurrentQueueDepth() > MaxQueueDepth {
            log.Println("load shedding engaged")

            http.Error(w,
                err.Error(),
                http.StatusServiceUnavailable)
            return
        }

        next.ServeHTTP(w, r)
    })
}

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

    // Register middleware
    r.Use(loadSheddingMiddleware)

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

```

Gorilla Mux middlewares are called on every request, each taking a request, doing something with it, and passing it down to another middleware or the final handler. This makes them perfect for implementing general request logging, header manipulation, ResponseWriter hijacking, or, in our case, resource-reactive load shedding.

Our middleware uses the fictional `CurrentQueueDepth()` (your actual function will depend on your implementation) to check the current queue depth, and rejects requests with an HTTP 503 (service

unavailable) if the value is too high. More sophisticated implementations might even be smarter about choosing which work is dropped by prioritizing particularly important requests.

## Graceful Service Degradation

Resource-sensitive load shedding works well, but in some applications it's possible to act a little more gracefully by significantly decreasing the quality of responses when the service is approaching overload. Such *graceful degradation* takes the concept of load shedding one step further by strategically reducing the amount of work that needed to satisfy each request instead of just rejecting requests.

There are as many ways of doing this as there are services, and not every service can be degraded in a reasonable manner, but common approaches include falling back on cached data or less expensive — if less precise — algorithms.

## Play It Again: Retrying Requests

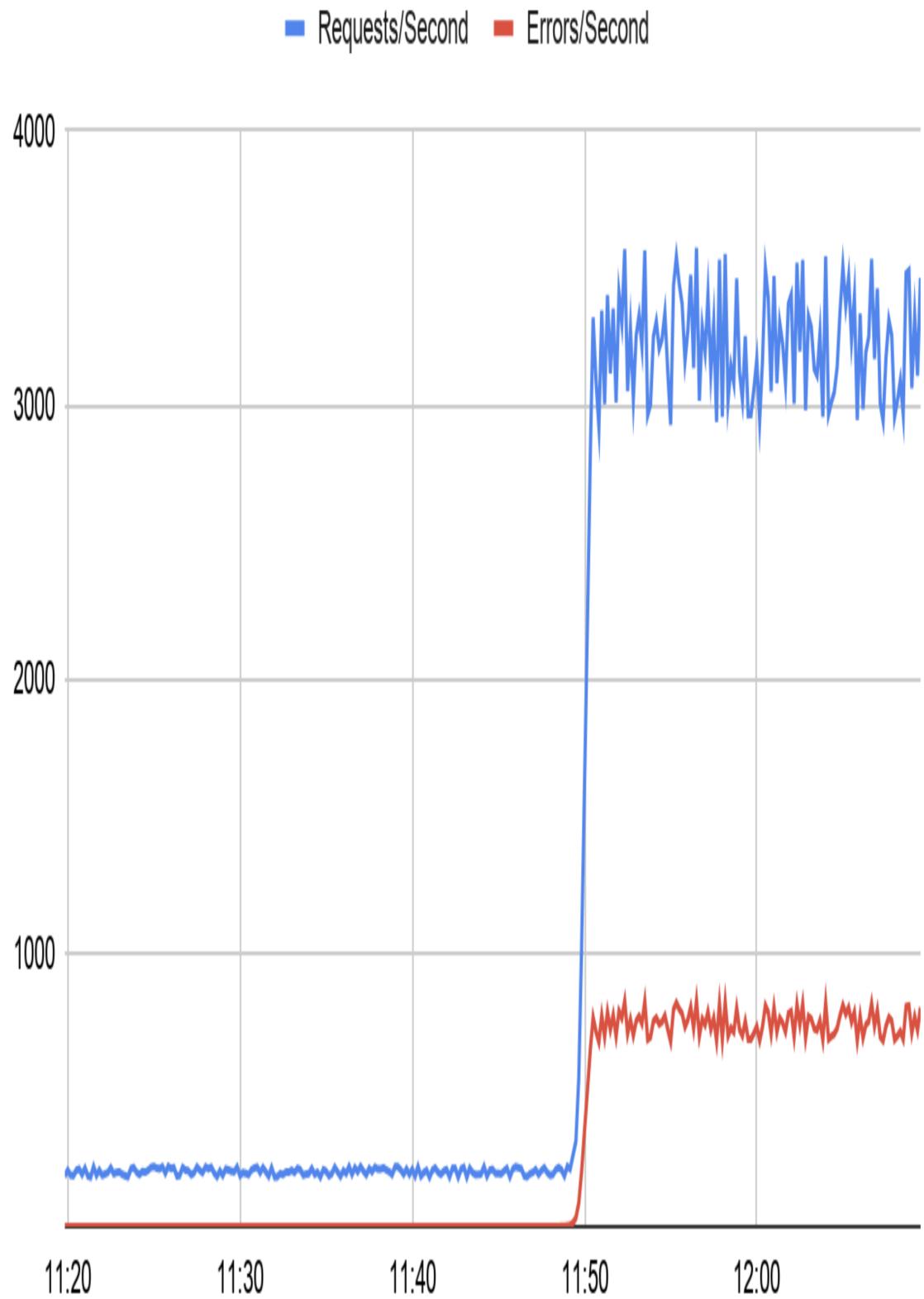
When a request receives an error response, or doesn't receive a response at all, it should just try again, right? Well, kinda. Retrying makes sense, but it's a lot more nuanced than that.

Take this snippet for example, a version of which I've found in a production system:

```
res, err := SendRequest()
for err != nil {
    res, err = SendRequest()
}
```

It seems seductively straight-forward, doesn't it? It *will* repeat failed requests, but that's also *exactly* what it will do. So when this logic was deployed to a few hundred servers and the service it was

issuing requests to went down, the entire system went with it. A review of the service metrics revealed this:



*Figure 9-2. The anatomy of a “retry storm”.*

It seems that when the downstream service failed, our service — every single instance of it — entered its retry loop, making *thousands* of requests per second and bringing the network to its knees so severely that we were forced to essentially restart the entire system.

This is actually a very common kind of cascading failure known as a *retry storm*. In a retry storm, well-meaning logic intended add resilience to a component acts against the larger system. Very often, even when the conditions that caused the downstream service to go down are resolved, it can’t come back up because it’s instantly brought under too much load.

But, retries are a good thing, right?

Yes, but whenever you implement retry logic, you should always include a *backoff algorithm*, which we’ll conveniently discuss in the next section.

## Backoff Algorithms

When a request to a downstream service fails for any reason, “best” practice is to retry the request. But how long should you wait? If you wait too long, important work may be delayed. Too little and you risk overwhelming the target, the network, or both.

The common solution is to implement a *backoff algorithm* that introduces a delay between retries to reduce the frequency of attempts to some safe and acceptable rate.

There are a variety of backoff algorithms available, the simplest of which is to include a short, fixed-duration pause between retries, as follows:

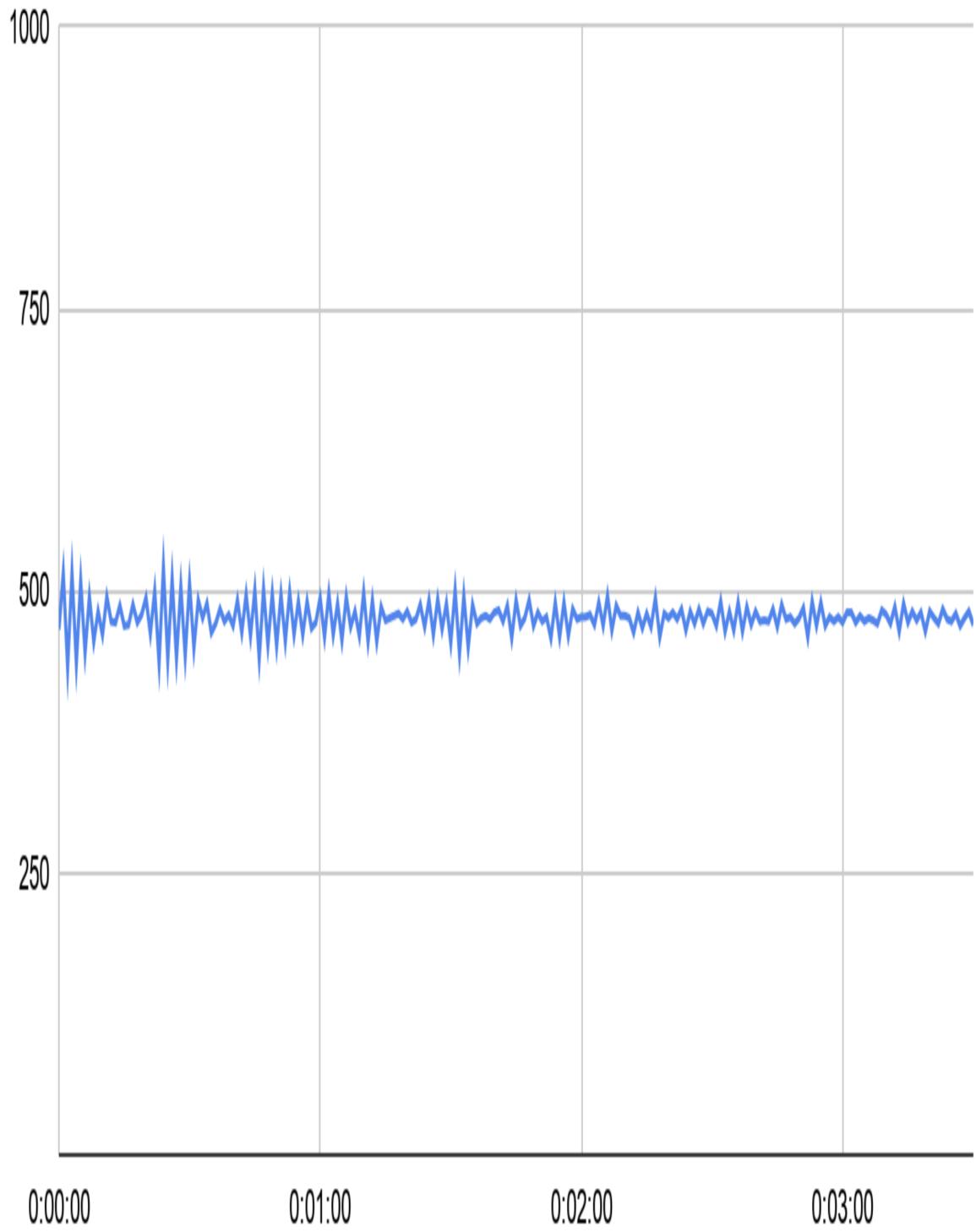
```
res, err := SendRequest()
for err != nil {
    time.Sleep(2 * time.Second)
```

```
    res, err = SendRequest()  
}
```

In the above snippet, `SendRequest` is used to issue a request, returning string and error values. However, if `err` isn't `nil`, the code enters a loop, sleeping for two seconds before retrying, repeating indefinitely until it receives a non-error response.

In [Figure 9-3](#) below, we illustrate the number of requests generated by 1000 simulated instances using this method<sup>10</sup>. As you can see, while the fixed-delay approach might reduce the request count compared to having no backoff at all, the overall number of requests is still quite consistently high.

■ Requests/Second



*Figure 9-3. Requests/second of 1000 simulated instances using a 2-second retry delay.*

A fixed-duration backoff delay might work fine if you have a very small number of retrying instances, but it doesn't scale very well, since a sufficient number of requestors can still overwhelm the network.

However, we can't always assume that any given service will have a small enough number of instances not to overwhelm the network with retries, or that our service will even be the only one retrying. For this reason, many backoff algorithms implement an *exponential backoff*, in which the durations of the delays between retries roughly doubles with each attempt up to some fixed maximum.

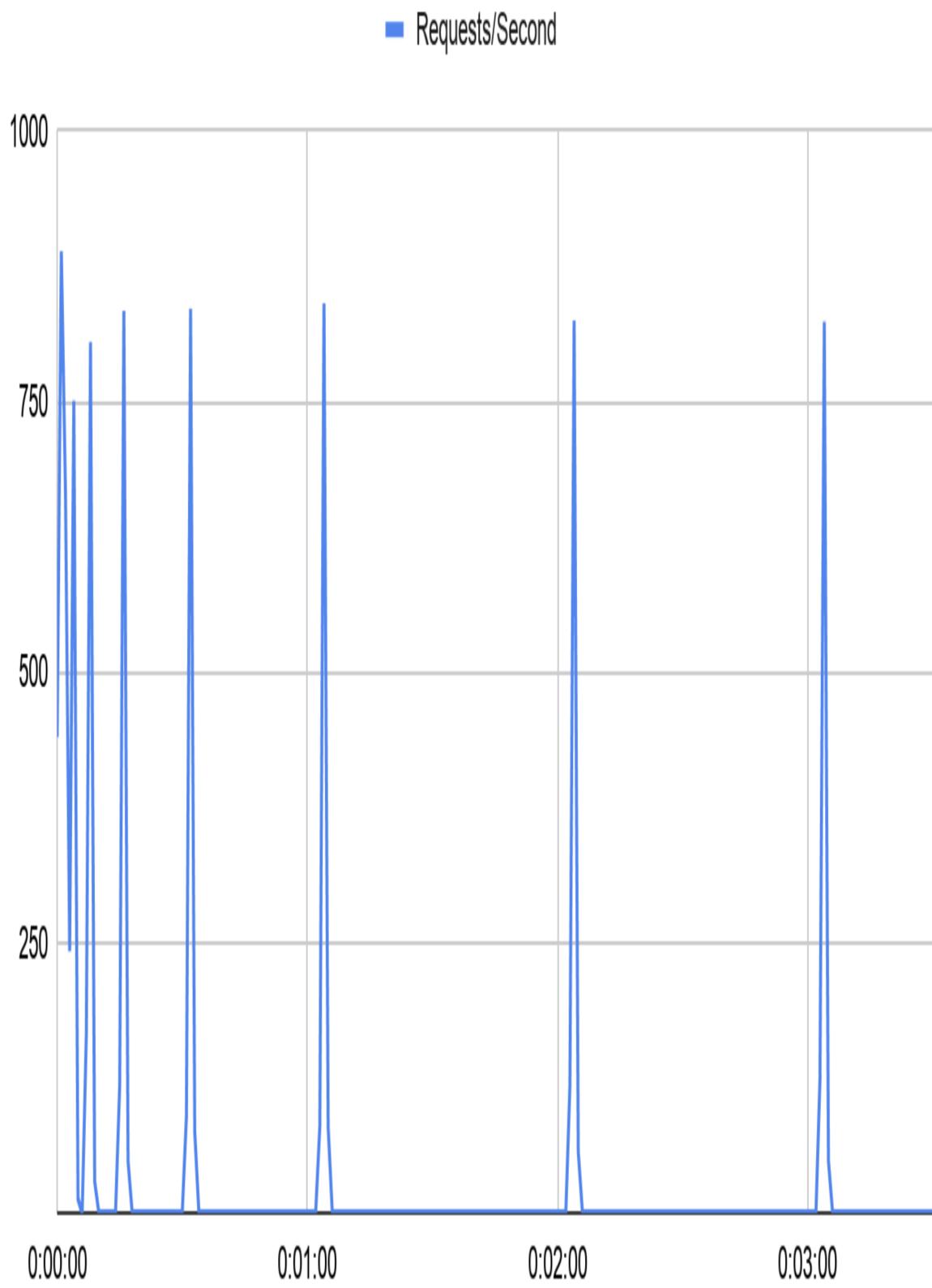
A very common (but flawed, as you'll soon see) exponential backoff implementation might look something like the following:

```
res, err := SendRequest()
base, cap := time.Second, time.Minute

for backoff := base; err != nil; backoff <= 1 {
    if backoff > cap {
        backoff = cap
    }
    time.Sleep(backoff)
    res, err = SendRequest()
}
```

In this snippet, we specify a starting duration, `base`, and a fixed maximum duration, `cap`. In the loop, the value of `backoff` starts at `base` and doubles each iteration to a maximum value of `cap`.

You would think that this logic would help to mitigate the network load and retry request burden on downstream services. Simulating this implementation for 1000 nodes, however, tells another story, illustrated in [Figure 9-4](#).



*Figure 9-4. Requests/second of 1000 simulated instances using an exponential backoff.*

It would seem that having 1000 nodes with exactly the same retry schedule still isn't optimal, since the retries are now clustering, possibly generating enough load in the process to cause problems. So in practice, pure exponential backoff doesn't necessarily help as much we'd like.

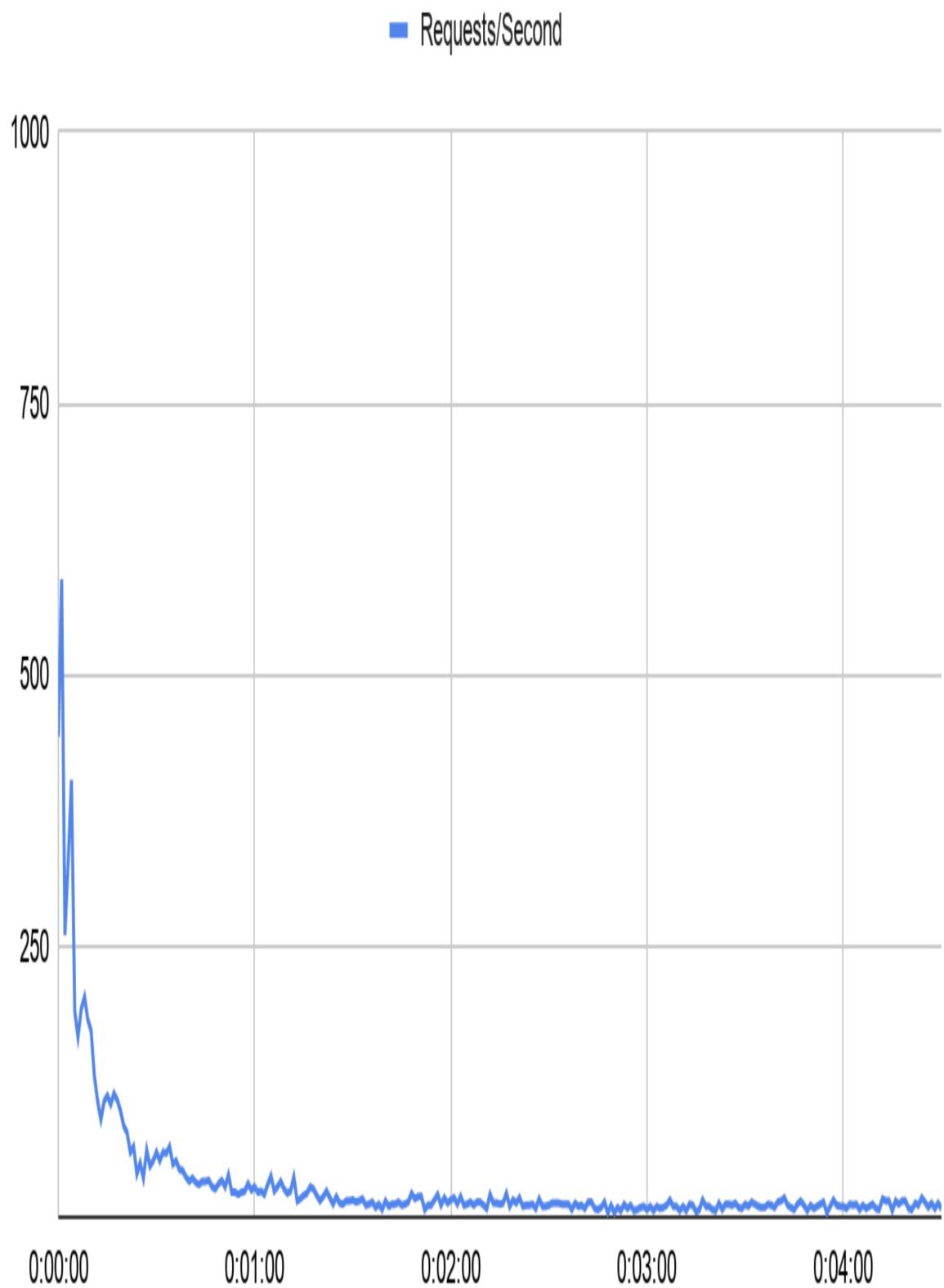
It would seem that we need some way to spread the spikes out so that the retries occur at a roughly constant rate. The solution is to include an element of randomness, called *jitter*. Adding jitter to our above backoff function results in something like the snippet below:

```
res, err := SendRequest()
base, cap := time.Second, time.Minute

for backoff := base; err != nil; backoff <= 1 {
    if backoff > cap {
        backoff = cap
    }

    jitter := rand.Int63n(int64(backoff * 3))
    sleep := base + time.Duration(jitter)
    time.Sleep(sleep)
    res, err = SendRequest()
}
```

Simulating running this code on 1000 nodes produces the pattern presented in [Figure 9-5](#).



*Figure 9-5. Requests/second of 1000 simulated instances using an exponential backoff with jitter.*

## WARNING

The `rand` package's top-level functions produce a deterministic sequence of values each time the program is run. If you don't use the `rand.Seed` function to provide a new seed value, they behave as if seeded by `rand.Seed(1)` and always produce the same "random" sequence of numbers.

When we use exponential backoff with jitter the number of retries both decreases over a short interval — so as not to overstress services that are trying to come up — but also spreads them out over time so that they occur at an approximately constant rate.

Who would have thought there was more to retrying requests than retrying requests?

## Circuit Breaking

We first introduced the Circuit Breaker pattern in [Chapter 4](#) as a function that degrades potentially failing method calls as a way to prevent larger or cascading failures. That definition still holds, and because we're not going to extend or change it much, we won't dig into it in *too* much detail here.

To review, the Circuit Breaker pattern tracks the number of consecutive failed requests made to a downstream component. If the failure count passes some threshold, the circuit is "opened", and all attempts to issue additional requests fail immediately (or return some defined fallback). After a waiting period, the circuit automatically "closes", resuming its normal state and allowing requests to be made normally.

## TIP

Not all resilience patterns are defensive.

Sometimes it pays to be a good neighbor.

A properly applied Circuit Breaker pattern can make the difference between system recovery and cascading failure. In addition to the obvious benefits of not wasting resources or clogging the network with doomed requests, a circuit breaker (particularly one with a backoff function) can give a malfunctioning service enough room to recover, allowing it to come back up and restore correct service.

The Circuit Breaker pattern was covered in some detail in [Chapter 4](#), so that's all we're going to say about it here. Take a look at "[Circuit Breaker](#)" for more background and code examples. The addition of jitter to the example's backoff function is left as an exercise for the reader<sup>11</sup>.

## WHAT'S THE DIFFERENCE BETWEEN CIRCUIT BREAKER AND THROTTLE?

At a quick glance, the Circuit Breaker pattern might seem to resemble a Throttle — after all they're both resilience patterns that rate requests — but they really are two quite different things.

- *Circuit Breaker* is generally applied only to *outgoing* requests. It usually doesn't care one bit about the request rate: it's only concerned with the number of failed requests, and only if they're consecutive.
- *Throttle* works like the throttle in a car by limiting a number of requests — regardless of success or failure — to some maximum rate. It's *typically* applied to incoming traffic, but there's no rule that says it has to be.

# Timeouts

The importance of timeouts isn't always appreciated. However, the ability for a client to recognize when a request is unlikely to be satisfied allows it to release resources that it—and any downstream requestors it might be acting on behalf of—might otherwise hold on to. This holds just as true for a service, which may find itself holding onto requests until long after a client has given up.

For example, imagine a basic service that queries a database. If that database should suddenly slow so that queries take a few seconds to complete, requests to the service—each holding on to a database connection—could accumulate, eventually depleting the connection pool. If the database is shared, it could even cause other services to fail, resulting in a cascading failure.

If the service had timed out instead of holding on to the database it could have degraded service instead of failing outright.

In other words, if you think you're going to fail, fail fast.

## Using Context for Service-Side Timeouts

We first introduced `context.Context` back in [Chapter 4](#) as Go's idiomatic means of carrying deadlines and cancelation signals between processes<sup>12</sup>. If you'd like a refresher, or just want to put yourself in the right frame of mind before continuing, go ahead and take a look at "[The Context Package](#)".

You might also recall that later in the same chapter, in "[Timeout](#)", we covered the *Timeout* pattern, which uses Context to not only allow a process to stop waiting for an answer once it's clear that a result may not be coming, but to also notify other functions with derived Contexts to stop working and release any resources that they might also be holding on to.

This ability to cancel not just local but sub-functions is so powerful that it's generally considered good form for functions to accept a

Context value if they have the potential to run longer than a caller might want to wait, which is almost always true if the call traverses a network.

For this reason, there are many excellent samples of Context-accepting functions scattered throughout Go's standard library. Many of these can be found in the `sql` package, which includes Context-accepting versions of many of its functions. For example, the `DB` struct's `QueryRow` method has an equivalent `QueryRowContext` that accepts a `Context` value.

A function that uses this technique to provide the username of a user based on an ID value might look something like the following:

```
func UserName(ctx context.Context, id int) (string, error) {
    const query = "SELECT username FROM users WHERE id=?"

    dctx, cancel := context.WithTimeout(ctx, 15*time.Second)
    defer cancel()

    var username string
    err := db.QueryRowContext(dctx, query, id).Scan(&username)

    return username, err
}
```

The above `UserName` function accepts a `context.Context` and an `id` integer, but it also creates its own derived `Context` with a rather long timeout. This approach provides a default timeout that automatically releases any open connections after 15 seconds — longer than many clients are likely to be willing to wait — while also being responsive to cancellation signals from the caller.

The responsiveness to outside cancellation signals can be quite useful. The `http` framework provides yet another excellent example of this, as demonstrated in the below `UserGetHandler` HTTP handler function:

```

func UserGetHandler(w http.ResponseWriter, r *http.Request) {
    vars := mux.Vars(r)
    id := vars["id"]

    // Get the request's context. This context is canceled when
    // the client's connection closes, the request is canceled
    // (with HTTP/2), or when the ServeHTTP method returns.
    rctx := r.Context()

    ctx, cancel := context.WithTimeout(rctx, 10*time.Second)
    defer cancel()

    username, err := UserName(ctx, id)

    switch {
    case errors.Is(err, sql.ErrNoRows):
        http.Error(w, "no such user", http.StatusNotFound)
    case errors.Is(err, context.DeadlineExceeded):
        http.Error(w, "database timeout", http.StatusGatewayTimeout)
    case err != nil:
        http.Error(w, err.Error(), http.StatusInternalServerError)
    default:
        w.Write([]byte(username))
    }
}

```

In `UserGetHandler`, the first thing we do is retrieve the request's Context via its `Context()` method. Conveniently, this Context is canceled when the client's connection closes, when the request is canceled (with HTTP/2), or when the `ServeHTTP` method returns.

From this we create a derived context, applying our own explicit timeout, which will cancel the Context after 10 seconds, no matter what.

Because the derived context is passed to the `UserName` function, we are able to draw a direct causative line between closing the HTTP request and closing the database connection: if the request's Context closes, all derived *Cons* close as well, ultimately ensuring that all open resources are released as well in a loosely-coupled manner.

## Timing Out HTTP/REST Client Calls

Back in “[A Possible Pitfall of Convenience Functions](#)” we presented one of the pitfalls of the http “convenience functions” like `http.Get` and `http.Post`: that they use the default timeout. Unfortunately, the default timeout value is 0, which Go interprets as “no timeout”.

The mechanism we presented at the time for setting timeouts for client methods was to create a custom `Client` value with a non-zero `Timeout` value, as follows:

```
var client = &http.Client{
    Timeout: time.Second * 10,
}

response, err := client.Get(url)
```

This works perfectly fine, and in fact will cancel a request in exactly the same way as if its `Context` is canceled. But what if you want to use an existing or derived `Context` value? For that you’ll need access to the underlying `Context`, which you can get by using `http.NewRequestWithContext`, the `Context`-accepting equivalent of `http.NewRequest`, which allows a programmer to specify a `Context` that controls the entire lifetime of the request and its response.

This isn’t as much of a divergence as it might seem. In fact, looking at the source code for the `Get` method on the `http.Client` shows that under the covers, it’s just using `NewRequest`.

```
func (c *Client) Get(url string) (resp *Response, err error) {
    req, err := NewRequest("GET", url, nil)
    if err != nil {
        return nil, err
    }

    return c.Do(req)
}
```

As you can see, the standard Get method calls NewRequest to create a \*Request value, passing it the method name and URL (the last parameter accepts an optional io.Reader for the request body, which we don't need here). A call to the Do function executes the request proper.

Not counting an error check and the return, the entire method consists of just one call. It would seem that if we wanted to implement similar functionality that also accepts a Context value, we could without much hassle.

One way to do this might be to implement a GetContext function that accepts a Context value.

```
type ClientContext struct {
    http.Client
}

func (c *ClientContext) GetContext(ctx context.Context, url string)
    (resp *http.Response, err error) {

    req, err := http.NewRequestWithContext(ctx, "GET", url, nil)
    if err != nil {
        return nil, err
    }

    return c.Do(req)
}
```

Our new GetContext function is functionally identical to the canonical Get, except that it also accepts a Context value, which it uses to call http.NewRequestWithContext instead of http.NewRequest.

Using our new ClientContext would be very similar to using a standard http.Client value, except instead of calling client.Get we'd call client.GetContext (and pass along a Context value, of course):

```
func main() {
    client := &ClientContext{}
```

```

    ctx, cancel := context.WithTimeout(context.Background(),
5*time.Second)
    defer cancel()

    response, err := client.GetContext(ctx, "http://www.example.com")
    if err != nil {
        log.Fatal(err)
    }

    bytes, _ := ioutil.ReadAll(response.Body)
    fmt.Println(string(bytes))
}

```

But does it work? It's not a *proper* test with a testing library, but we can manually kick the tires by setting the deadline to 0 and running it:

```

$ go run .
2020/08/25 14:03:16 Get "http://www.example.com": context deadline
exceeded
exit status 1

```

And it would seem that it does! Excellent.

## Timing Out gRPC Client Calls

Just like `http.Client`, gRPC clients default to “no timeout”, but also allow timeouts to be explicitly set.

As we saw in “[Implementing the gRPC Client](#)”, gRPC clients typically use the `grpc.Dial` function to establish a connection to a client, and that a list of `grpc.DialOption` values — constructed via functions like `grpc.WithInsecure` and `grpc.WithBlock` — can be passed to it to configure how that connection is set up.

Among these options is `grpc.WithTimeout`, which can be used to configure a client dialing timeout:

```

opts := []grpc.DialOption{
    grpc.WithInsecure(),
    grpc.WithBlock(),
    grpc.WithTimeout(5 * time.Second),
}

```

```
    }
    conn, err := grpc.Dial(serverAddr, opts...)
}
```

However, while `grpc.WithTimeout` might seem convenient on its face, it's actually been deprecated for some time, largely because its mechanism is inconsistent (and redundant) with the preferred Context timeout method. We show it here for the sake of completion.

### WARNING

The `grpc.WithTimeout` option is deprecated and will eventually be removed. Use `grpc.DialContext` and `context.WithTimeout` instead.

Instead, the preferred method of setting a gRPC dialing timeout is the very convenient (for us) `grpc.DialContext` function, which allows us to use (or reuse) a `context.Context` value. This is actually doubly useful, because gRPC service methods accept a `Context` value anyway, so there really isn't even any additional work to be done.

```
func TimeoutKeyValueGet() *pb.Response {
    // Use context to set a 5-second timeout.
    ctx, cancel := context.WithTimeout(context.Background(), 5 *
time.Second)
    defer cancel()

    // We can still set other options as desired.
    opts := []grpc.DialOption{grpc.WithInsecure(), grpc.WithBlock()}

    conn, err := grpc.DialContext(ctx, serverAddr, opts...)
    if err != nil {
        grpclog.Fatalf(err)
    }
    defer conn.Close()

    client := pb.NewKeyValueClient(conn)

    // We can re-use the same Context in the client calls.
    response, err := client.Get(ctx, &pb.GetRequest{Key: key})
    if err != nil {
```

```
        grpclog.Fatalf(err)
    }

    return response
}
```

As advertised, `TimeoutKeyValueGet` uses `grpc.DialContext` — to which we pass a `context.Context` value with a 5-second timeout — instead of `grpc.Dial`. The `opts` list is otherwise identical except, obviously, that it no longer includes `grpc.WithTimeout`.

Note the `client.Get` method call. As we mentioned above, gRPC service methods accept a `Context` parameter, so we simply re-use the existing one. Importantly, re-using the same `Context` value will constrain both operations under the same timeout calculation — a `Context` will time out regardless of how it's used — so be sure to take that into consideration when planning your timeout values.

## Idempotence

As we discussed at the top of [Chapter 4](#), cloud native applications by definition exist in and are subject to all of the idiosyncrasies of a networked world. It's a plain fact of life that networks — all networks — are unreliable, and messages sent across them don't always arrive at their destination on time (or at all).

What's more, if you send a message to but don't get a response, you have no way to know what happened. Did the message get lost on its way to the recipient? Did the recipient get the message, and the response get lost? Maybe everything is working fine, but the round trip is just taking a little longer than usual?

In such a situation, the only option is to send the message again. But it's not enough to cross your fingers and hope for the best. It's important to plan for this inevitability by making it safe to resend messages by designing the functions for *idempotence*.

You might recall that we briefly introduced the concept of idempotence in “[What Is Idempotence and Why Does It Matter?](#)”, in which we defined an idempotent operation as one that has the same effect after multiple applications as a single application. As the designers of HTTP understood, it also happens to be an important property of any cloud native API that guarantees that any communication can be safely repeated (see “[The Origins of Idempotence on the Web](#)” for a bit on that history).

The actual means of achieving idempotence will vary from service to service, but there are some consistent patterns that we’ll review in the remainder of this section.

## THE ORIGINS OF IDEMPOTENCE ON THE WEB

The concepts of idempotence and safety, at least in the context of networked services, were first defined way back in 1997 in the HTTP/1.1 standard<sup>13</sup>.

An interesting aside: that ground-breaking proposal, as well as the HTTP/1.0 “informational draft” that preceded it the year before<sup>14</sup>, were authored by two greats.

The primary author of the original HTTP/1.0 draft (and the last author of the proposed HTTP/1.1 standard) was Sir Timothy John Berners-Lee, who is credited for inventing the World Wide Web, the first web browser, and the fundamental protocols and algorithms allowing the Web to scale — for which he was awarded with an ACM Turing Award, a knighthood, and various honorary degrees.

The primary author of the proposed HTTP/1.1 standard (and the second author of the original HTTP/1.0 draft) was Roy Fielding, then a graduate student at the University of California Irvine.

Despite being one of the original authors of the World Wide Web, Fielding is perhaps known for his doctoral dissertation, in which he invented REST<sup>15</sup>.

## THE MATHEMATICAL DEFINITION OF IDEMPOTENCE

The origin of idempotence actually lays 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  $x$ .

## How Do I Make My Service Idempotent?

Idempotence isn't baked into the logic of any particular framework. Even in HTTP — and by extension, REST — idempotence is a matter of convention and isn't explicitly enforced. There's nothing stopping you from — by oversight or on purpose — implementing a non-idempotent GET if you really wanted to<sup>16</sup>.

One of the reasons that idempotence is sometimes so tricky is because it relies on logic built into the core application, rather than at the REST or gRPC API layer. For example, if back in [Chapter 5](#) we had wanted to make our key-value store consistent with traditional CRUD (create, read, update, and delete) operations (and therefore *not* idempotent) we might have done something like this:

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

func Create(key, value string) error {
    if _, ok := store[key]; ok {
        return errors.New("duplicate key")
    }

    store[key] = value
    return nil
}
```

```

func Update(key, value string) error {
    if _, ok := store[key]; !ok {
        return errors.New("no such key")
    }

    store[key] = value
    return nil
}

func Delete(key string) error {
    if _, ok := store[key]; ok {
        return errors.New("no such key")
    }

    delete(store, key)
    return nil
}

```

This CRUD-like service implementation may be entirely well-meaning, but if any of these methods have to be repeated the result would be an error. What's more, there's also a fair amount of logic involved in checking against the current state which wouldn't be necessary in an equivalent idempotent implementation like the following:

```

var store = make(map[string]string)

func Set(key, value string) {
    store[key] = value
}

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

```

This version is a *lot* simpler, in more than one way. First, we no longer need separate “create” and “update” operations, so we can combine these into a single Set function. Also, not having to check the current state with each operation reduces the logic in each

method, a benefit that continues to pay dividends as the service increases in complexity.

Finally, if an operation has to be repeated, it's no big deal. For both the Set and Delete functions, multiple identical calls will have the same result. They are idempotent.

## What About Scalar Operations?

“So”, you might say, “that’s all good and well for operations that are either *done* or *not done*, but what about more complex operations? Operations on scalar values, for example?”

That’s a fair question. After all, it’s one thing to PUT a thing in a place: it’s either been PUT, or it hasn’t. All you have to do is not return an error for re-PUTs. Fine.

But what about an operation like “add \$500 to account 12345”? Such a request might carry a JSON payload that looks something like the following:

```
{  
  "credit":{  
    "accountID": 12345,  
    "amount": 500  
  }  
}
```

Repeated applications of this operation would lead to an extra \$500 going to account 12345, and while the owner of the account might not mind so much, the bank probably would.

But consider what happens we we add a `transactionID` value to our JSON payload:

```
{  
  "credit":{  
    "accountID": 12345,  
    "amount": 500,  
    "transactionID": 789
```

```
    }  
}
```

It may require some more bookkeeping, but this approach provides a workable solution to our dilemma. By tracking `transactionID` values, the recipient can safely identify and reject duplicate transactions. Idempotence achieved!

## Service Redundancy

Redundancy — the duplication of critical components or functions of a system with the intention of increasing reliability of the system — is often the first line of defense when it comes to increasing resilience in the face of failure.

We've already discussed one particular kind of redundancy — messaging redundancy, also known as "retries" — in "[Play It Again: Retrying Requests](#)". In this section, however, we'll consider the value of replicating critical system components so that if any one fails, one or more others are there to pick up the slack.

In a public cloud, this would mean deploying your component to multiple server instances, ideally across multiple zones or even across multiple regions. In a container orchestration platform like Kubernetes, this may even just be a matter of setting your replica count to a value greater than one.

As interesting as this subject is, however, we won't actually spend too much time on it. Service replication is an architectural subject that's been thoroughly covered in many other sources<sup>[17](#)</sup>. This is supposed to be a Go book, after all. But still, we'd be remiss to have an entire chapter about resilience and not even mention it.

## A WORD OF CAUTION: FAULT MASKING

*Fault masking* occurs when a system fault is invisibly compensated for without being explicitly detected.

For example, imagine a system with three service nodes, all performing a share of the tasks. If one node goes bad and the other nodes can compensate, you might never notice anything wrong. The fault has been masked.

Fault masking can conceal possibly progressive faults, and may eventually — and quietly — result in a loss of protective redundancy, often with a sudden and catastrophic outcome.

To prevent fault masking, it's important to include service health checks — which we'll discuss in "[Healthy Health Checks](#)" — that accurately report the health of a service instance.

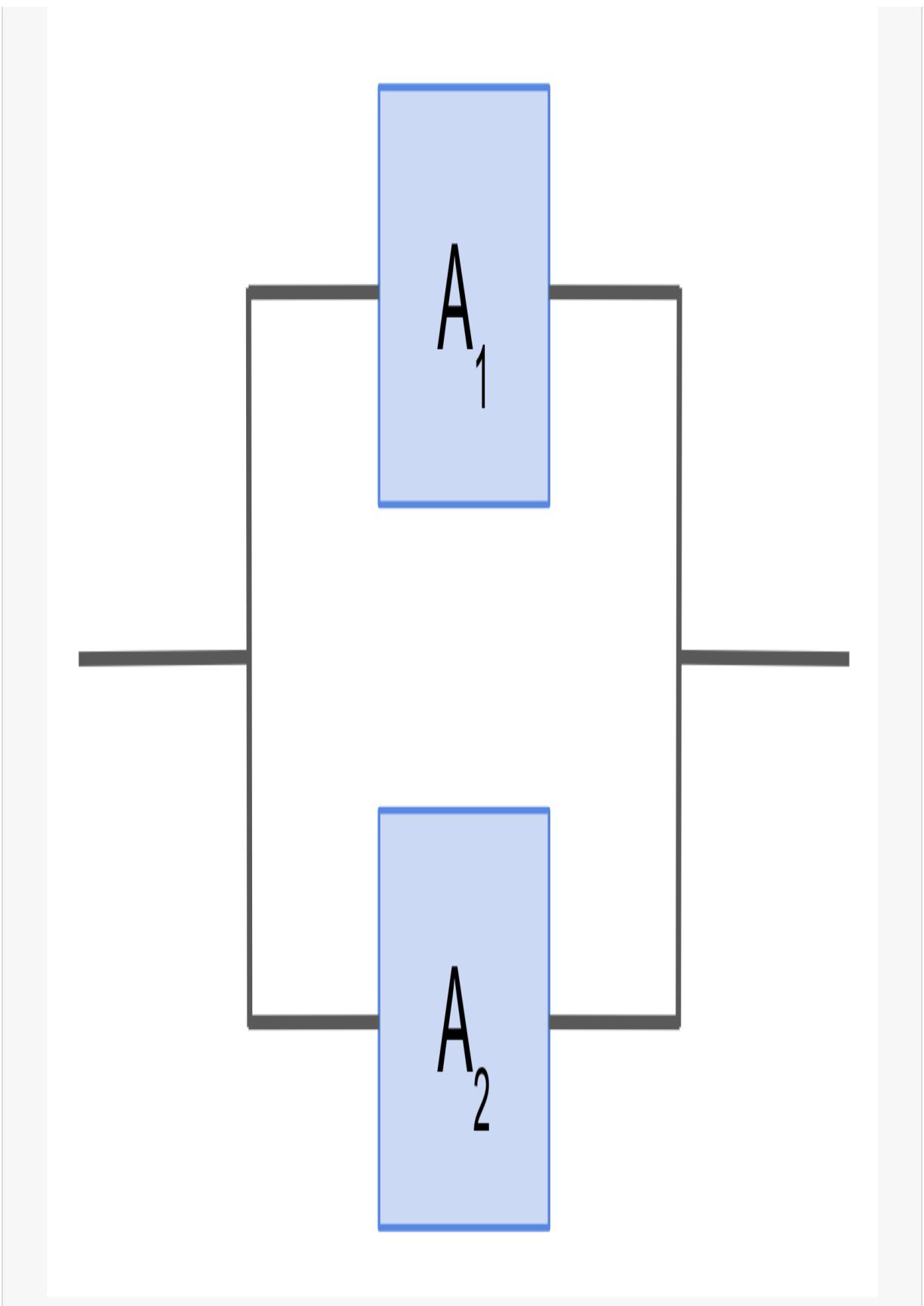
## Designing for Redundancy

The effort involved in designing a system so that its functions can be replicated across multiple instances can yield significant dividends. But exactly how much? Well... a lot. You can feel free to take a look at the box below if you're interested in the math, but if you don't you can just trust me on this one.

## RELIABILITY BY THE NUMBERS

Imagine, if you will, a service with “two-nines”—or 99%—of availability. Any given request to this system it has a theoretical probability of success of 0.99, which is denoted  $A_s$ . This actually isn’t a very good, but that’s the point.

So, what kind of availability can you get if two of these identical instances are arranged in parallel, so that both have to be down to interrupt service<sup>18</sup>? This kind of arrangement can be diagrammed as below:



So what's the resulting system availability? What we really want to know is: what's the probability that both instances will be *unavailable*? To answer this, we take the product of each components' probability of failure:

$$U_s = (1 - A_1) \times (1 - A_2)$$

This method generalizes to any number of components arranged in parallel, so that the availability of any  $N$  components is equal to 1 minus the product of their unavailabilities:

$$A_s = 1 - \prod_1^N (1 - A_i)$$

When all  $A_i$  are equal, then this can be simplified to:

$$A_s = 1 - (1 - A_i)^N$$

So what about our example? Well, with two components, each with a 99% availability<sup>19</sup>, we get the following:

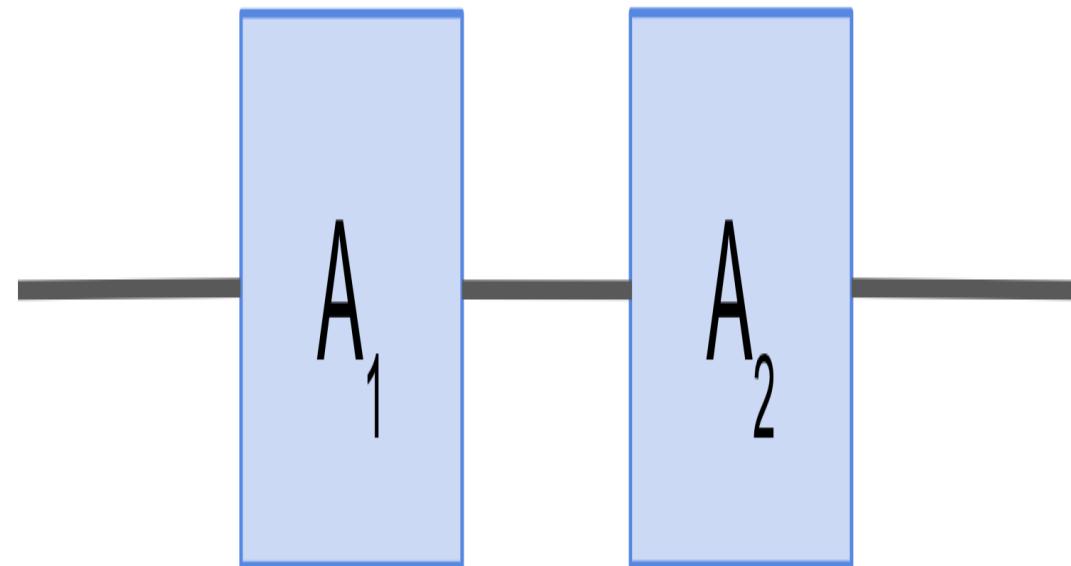
$$A_s = 1 - (1 - 0.99)^2 = 0.9999$$

99.99%. Four nines. That's an improvement of two orders of magnitude, which isn't half bad. But what if we added a third replica? Extending this out a little, we get some interesting results, summarized in the table below.

<b>Components</b>	<b>Availability</b>	<b>Downtime per year</b>	<b>Downtime per month</b>
One component	99% ("2-nines")	3.65 days	7.31 hours
Two parallel components	99.99% ("4-nines")	52.60 minutes	4.38 minutes
Three parallel components	99.9999% ("6-nines")	31.56 seconds	2.63 seconds

Incredibly, three parallel instances, each of which isn't exactly awesome on its own, can provide a very impressive 6-nines of availability! This is why cloud providers advise customers to deploy their applications with three replicas.

But what if the components are arranged serially, like a load balancer in front of our components? This might look something like the following:



In this kind of arrangement, if either component is unavailable, the entire system is unavailable. Its availability is the product of the availabilities of its components:

$$A_s = \prod_1^N A_i$$

When all  $A_i$  are equal, then this can be simplified to:

$$A_s = A^N$$

So what if we slapped a dodgy load balancer instance in front of our fancy 99.9999% available set of service replicas? As it turns out, the result isn't so good:

$$0.99 \times 0.999999 = 0.98999901$$

That's even lower than the load balancer by itself! This is important, because as it turns out:

### WARNING

The total reliability of a sequential system cannot be higher than the reliability of any one of its sequences of subsystems.

## Autoscaling

Very often, the amount of load that a service is subjected to varies over time. The textbook example is the user-facing web service where load increases during the day and decreases at night. If such a service is built to handle the peak load, its wasting time and money at night. If it's built only to handle the nighttime load, it will be overburdened in the daytime.

Autoscaling is a technique that builds on the idea of load balancing by automatically adding or removing resources — be they cloud server instances or Kubernetes pods — to dynamically adjust capacity to meet current demand. This ensures that your service can meet a variety of traffic patterns, anticipated or otherwise.

As an added bonus, applying autoscaling to your cluster can save money by right-sizing resources according to service requirements.

All major cloud providers provide a mechanism for scaling server instances, and most of their managed services implicitly or explicitly support autoscaling. Container orchestration platforms like Kubernetes also include support for autoscaling, both for the number of pods (horizontal autoscaling) and their CPU and memory limits (vertical autoscaling).

Autoscaling mechanics vary considerably between cloud providers and orchestration platforms, so a detailed discussion of how to gather metrics and configure things like predictive autoscaling is beyond the scope of this book. However, some key points to remember:

- Set reasonable maximums, so that unusually large spikes in demand (or, heaven forbid, cascade failures) completely don't blow your budget. The throttling and load shedding techniques that we discussed in “[Preventing Overload](#)” are also useful here.
- Minimize startup times. If you're using server instances, bake machine images ahead of time to minimize configuration time at startup. This is less of an issue on Kubernetes, but container images should still be kept small and startup times reasonably short.
- No matter how fast your startup, scaling takes a non-zero amount of time. Your service should have *some* wiggle room without having to scale.

- As we discussed in “[Scaling Postponed: Efficiency](#)”, the best kind of scaling is the kind that never needs to happen.

## Healthy Health Checks

In “[Service Redundancy](#)”, we briefly discussed the value of redundancy — the duplication of critical components or functions of a system with the intention of increasing overall system reliability — and its value for improving the resilience of a system.

Multiple service instances means having a load balancing mechanism — a service mesh or dedicated load balancer — but what happens when a service instance goes bad? Certainly we don’t want the load balancer to continue sending traffic its way. So what do we do?

Enter the *health check*. In its simplest and most common form, a health check is implemented as an API endpoint that clients — load balancers, as well as monitoring services, service registries, etc. — can use to ask a service instance if it’s alive and healthy.

For example, a service might provide an HTTP endpoint (`/health` and `/healthz` are common naming choices) that returns a `200 OK` if the replica is healthy, and a `503 Service Unavailable` when it’s not. More sophisticated implementations can even return different status codes for different states: HashiCorp’s Consul service registry interprets any `2XX` status as a success, a `429 Too Many Requests` as a warning, and anything else as a failure.

Having an endpoint that can tell a client when a service instance is healthy (or not) sounds great and all, but it invites the question of what, exactly, does it mean for an instance to be “healthy”?

## TIP

Health checks are like bloom filters. A failing health check means a service isn't up, but a health check passing means the service is *probably* "healthy". (Credit: Cindy Sridharan<sup>20</sup>)

# What Does It Mean for an Instance to Be "Healthy"?

We use the word "healthy" in the context of services and service instances, but what exactly do we mean when we say that? Well, as is so often the case, there's a simple answer and a complex answer. Probably a lot of answers in between, too.

We'll start with the simple answer. Reusing an existing definition, an instance is considered "healthy" when it's "available". That is, when it's able to provide correct service.

Unfortunately, it isn't always so clear cut. What if the instance itself is functioning as intended, but a downstream dependency is malfunctioning? Should a health check even make that distinction? If so, should the load balancer behave differently in each case? Should an instance be reaped and replaced if it's not the one at fault, particularly if all service replicas are affected?

Unfortunately, there aren't any easy answers to these questions, so instead of answers I'll offer the next best thing: a discussion of the three most common approaches to health checking and their associated advantages and disadvantages. Your own implementations will depend on the needs of your service and your load balancing behavior.

## The Three Types of Health Checks

When a service instance fails, it's usually because of one of the following:

- A local failure like an application error or resource — CPU, memory, database connections, etc. — depletion.
- A remote failure in some dependency — a database or other downstream service — that affects the functioning of the service.

These two broad categories of failures give rise to three (yes, three) health checking strategies, each with its own fun little pros and cons.

*Liveness checks* do little more than return a “success” signal. They make no additional attempt to determine the status of the service, and say nothing about the service except that it’s listening and reachable. But, then again, sometimes this is enough. We’ll talk more about liveness checks in “[Liveness Checks](#)”.

*Shallow health checks* go further than liveness checks by verifying that the service instance is likely to be able to function. These health checks only test local resources, so they’re unlikely to fail on many instances simultaneously, but they can’t say for certain whether a particular request service instance will be successful. We’ll wade into shallow health checks in “[Shallow Health Checks](#)”.

*Deep health checks* provide a much better understanding of instance health, since they actually inspect of the ability of a service instance to perform its function, which also exercises downstream resources like databases. While thorough, they can be expensive, and are susceptible to false positives. We’ll dig into deep health checks in “[Deep Health Checks](#)”.

## Liveness Checks

A liveness endpoint always returns a “success” value, no matter what. While this might seem trivial to the point of uselessness — after all, what is the value of a health check that doesn’t say anything about health — liveness probes actually can provide some useful information by confirming:

- that the service instance is listening and accepting new connections on the expected port,
- that the instance is reachable over the network, and
- that any firewall, security group, or other configurations are correctly defined.

This simplicity comes with a predictable cost, of course. The absence of any active health checking logic makes liveness checks of limited use when it comes to evaluating whether a service instance can actually perform its function.

Liveness probes are also dead easy to implement. Using the `net/http` package, we can do the following:

```
func healthLivenessHandler(w http.ResponseWriter, r *http.Request) {
    w.WriteHeader(http.StatusOK)
    w.Write([]byte("OK"))
}

func main() {
    r := mux.NewRouter()
    http.HandleFunc("/healthz", healthLivenessHandler)
    log.Fatal(http.ListenAndServe(":8080", r))
}
```

The above snippet shows how little work can go into a liveness check. In it, we create and register a `/healthz` endpoint that does nothing but return a 200 OK (and the text OK, just to be thorough).

### WARNING

If you're using the `gorilla/mux` package, any registered middleware (like the load shedding function from “[Load Shedding](#)”) can affect your health checks!

## Shallow Health Checks

Shallow health checks go further than liveness checks by verifying that the service instance is *likely* to be able to function, but stop short of investigating in any way that might exercise a database or other downstream dependency.

Shallow health checks can evaluate any number of conditions that could adversely affect the service, including (but certainly not limited to):

- The availability of key local resources (memory, CPU, database connections).
- The ability to read or write local data, which checks disk space, permissions, and for hardware malfunctions such as disk failure.
- The presence of support processes, like monitoring or updater processes.

Shallow health checks are more definitive than liveness checks, and their specificity means that any failures are unlikely to affect the entire fleet at once<sup>21</sup>. However, shallow checks are prone to false positives: if your service is down because of some issue involving an external resource, a shallow check will miss it. What you gain in specificity, you also sacrifice in sensitivity.

A shallow health check might look something like the example below, which tests the service's ability to read and write to and from local disk.

```
func healthShallowHandler(w http.ResponseWriter, r *http.Request) {
    // Create our test file.
    // This will create a filename like /tmp/shallow-123456
    tmpFile, err := ioutil.TempFile(os.TempDir(), "shallow-")
    if err != nil {
        http.Error(w, err.Error(), http.StatusServiceUnavailable)
        return
    }
    defer os.Remove(tmpFile.Name())
```

```

// Make sure that we can write to the file.
text := []byte("Check.")
if _, err = tmpFile.Write(text); err != nil {
    http.Error(w, err.Error(), http.StatusServiceUnavailable)
    return
}

// Make sure that we can close the file.
if err := tmpFile.Close(); err != nil {
    http.Error(w, err.Error(), http.StatusServiceUnavailable)
    return
}

w.WriteHeader(http.StatusOK)
}

func main() {
    r := mux.NewRouter()
    http.HandleFunc("/healthz", healthShallowHandler)
    log.Fatal(http.ListenAndServe(":8080", r))
}

```

This simultaneously checks for available disk space, write permissions, and malfunctioning hardware, which can be a very useful thing to test, particularly if the service needs to write to an on-disk cache or other transient files.

An observant reader might notice that it writes to the default directory to use for temporary files. On Linux, this is /tmp, which is actually a RAM drive. This might be a useful thing to test as well, but if you want to test for the ability to write to disk on Linux you'll need to specify a different directory, or this becomes a very different test.

## Deep Health Checks

Deep health checks directly inspect the ability of a service to interact with its adjacent systems. This provides much better understanding of instance health by potentially identifying issues with dependencies, like invalid credentials, the loss of connectivity to data stores, or other unexpected networking issues.

However, while thorough, deep health checks can be quite expensive. They can take a long time and place a burden on dependencies, particularly if you’re running too many of them, or running them too often.

### TIP

Don’t try to test *every* dependency in your health checks: focus on the ones that are required for the service to operate.

### TIP

When testing multiple downstream dependencies, evaluate them concurrently if possible.

What’s more, because the failure of a dependency will be reported as a failure of the instance, deep checks are especially susceptible to false positives. Combined with the lower specificity compared to a shallow check — issues with dependencies will be felt by the entire fleet — and you have the potential for a cascading failure.

If you’re using deep health checks, you should take advantage of strategies like circuit breaking (which we covered in “[Circuit Breaking](#)”) where you can, and your load balancer should “fail open” (which we’ll discuss in “[Failing Open](#)”) whenever possible.

Below, we have a trivial example of a possible deep health check that evaluates a database by calling a hypothetical service’s `GetUser` function.

```
func healthDeepHandler(w http.ResponseWriter, r *http.Request) {
    // Retrieve the context from the request and add a 5-second
    // timeout
    ctx, cancel := context.WithTimeout(r.Context(), 5*time.Second)
    defer cancel()
```

```

// service.GetUser is a hypothetical method on a service interface
// that executes a database query
if err := service.GetUser(ctx, 0); err != nil {
    http.Error(w, err.Error(), http.StatusServiceUnavailable)
    return
}

w.WriteHeader(http.StatusOK)
}

func main() {
    r := mux.NewRouter()
    http.HandleFunc("/healthz", healthDeepHandler)
    log.Fatal(http.ListenAndServe(":8080", r))
}

```

Ideally, a dependency test should execute an actual system function, but also be lightweight to the greatest reasonable degree. In this example, the  `GetUser` function triggers a database query that satisfies both of these criteria<sup>22</sup>.

“Real” queries are generally preferable to just pinging the database for two reasons. First, they’re a more representative test of what the service is doing. Second, they allow the leveraging of end-to-end query time as a measure of database health. The above example actually does this — albeit in a very binary fashion — by using Context to set a hard timeout value, but you could choose to include more sophisticated logic instead.

## Failing Open

What if all of your instances simultaneously decide that they’re unhealthy? If you’re using deep health checks, this can actually happen quite easily (and, perhaps, regularly). Depending on how your load balancer is configured, you might find yourself with zero instances serving traffic, possibly causing failures rippling across your system.

Fortunately, some load balancers handle this quite cleverly by “failing open”. If a load balancer that fails open has *no* healthy targets — that is, if *all* of its targets’ health checks are failing — it will route traffic to all of its targets.

This is a slightly counter-intuitive behavior, but it makes deep health checks somewhat safer to use by allowing traffic to continue to flow even when a downstream dependency may be having a bad day.

## Summary

This was an interesting chapter to write. There’s quite a lot to say about resilience — and so much crucial supporting operational background — that I had to make some tough calls about what would make it in and what wouldn’t. At about 37 pages this chapter still turned out a fair bit longer than I intended, but I’m quite satisfied with the outcome. It’s a reasonable compromise between too little information and too much, and between operational background and actual Go implementations.

We reviewed what it means for a system to fail, and how complex systems fail (that is, one component at a time). This led naturally to discussing a particularly nefarious, yet common failure mode: cascading failures. In a cascade failure, a system’s own attempts to recover hasten its collapse, so we covered common measures of preventing them on the server side: throttling and load shedding.

Retries in the face of errors can contribute a lot to a service’s resilience, but as we saw in the DynamoDB case study, can also contribute to cascade failures when applied naively. So we dug deep into measures that can be taken on the client side as well, including circuit breakers, timeouts, and especially exponential backoff algorithms. There were several pretty graphs involved. I spent a lot of time on the graphs.

All of this led to conversations about service redundancy, how it affects reliability (with a little math thrown in, for fun), and when and how to best leverage autoscaling.

Of course, you can't talk about autoscaling without talking about resource "health". We asked (and did our best to answer) what it means for an instance to be "healthy", and how that translated into health checks. We covered the three kinds of health checks and weighed their pros and cons, paying particular attention to their relative sensitivity/specificity tradeoffs.

In [Link to Come] we'll take a break from the operational topics for a bit and wade into the subject of manageability: the art and science of changing the tires on a moving car.

- 
- 1 Lamport, Leslie. *DEC SRC Bulletin Board*, 28 May 1987
  - 2 Summary of the Amazon DynamoDB Service Disruption and Related Impacts in the US-East Region. Amazon AWS, September 2015.
  - 3 Cook, Richard I. "How Complex Systems Fail". 1998.
  - 4 If you're interested in a complete academic treatment, I highly recommend *Reliability and Availability Engineering* by Kishor S. Trivedi and Andrea Bobbio.
  - 5 Importantly, many faults are only evident in retrospect.
  - 6 See? We eventually got there.
  - 7 Go on, ask me how I know this.
  - 8 Especially if the service is available on the open sewer that is the public internet.
  - 9 Wikipedia contributors. "Token bucket". *Wikipedia, The Free Encyclopedia*, 5 Jun. 2019.
  - 10 The code used to simulate all data in this section is available in [the associated GitHub repository](#).
  - 11 Doing that here felt redundant, but I'll admit that I may have gotten a bit lazy.
  - 12 And, technically, request-scoped values, but the correctness of this functionality is debatable.

- 13 Fielding, R., et al, “Hypertext Transfer Protocol — HTTP/1.1”, Proposed Standard, RFC 2068, June 1997.
- 14 Berners-Lee, T., et al, “Hypertext Transfer Protocol — HTTP/1.0”, Informational, RFC 1945, May 1996.
- 15 Fielding, Roy Thomas. “Architectural Styles and the Design of Network-Based Software Architectures”. *UC Irvine*, 2000, pp. 76–106.
- 16 You monster.
- 17 *Building Secure and Reliable Systems: Best Practices for Designing, Implementing, and Maintaining Systems* by Heather Adkins — and a host of other authors — is one excellent example
- 18 Brace yourself. We’re going in.
- 19 This assumes that the failure rates of the components are absolutely independent, which is very unlikely in the real world. Treat as you would spherical cows in a vacuum.
- 20 Sridharan, Cindy (@copyconstruct). “Health checks are like bloom filters...” 5 Aug 2018, 3:21 AM. Tweet.
- 21 Though I’ve seen it happen.
- 22 It’s an imaginary function, so let’s just agree that that’s true.

1. I. Going Cloud Native
2. 1. What Is a “Cloud Native” Application?
  - a. The Story So Far
  - b. What Is Cloud Native?
    - i. Scalability
    - ii. Loose coupling
    - iii. Resilience
    - iv. Manageability
    - v. Observability
  - c. Why Is Cloud Native a Thing?
  - d. Summary
3. 2. Why Go Rules the Cloud Native World
  - a. The Motivation Behind Go
  - b. Features For a Cloud Native World
    - i. Composition and Structural Typing
    - ii. Comprehensibility
    - iii. CSP-Style Concurrency
    - iv. Fast Builds
    - v. Linguistic Stability
    - vi. Memory Safety
    - vii. Performance
    - viii. Static Linking

- ix. Static Typing
- c. Summary
- 4. II. Cloud Native Go Constructs
- 5. 3. Go Language Foundations
  - a. Types
    - i. Simple Numbers
    - ii. Complex Numbers
    - iii. Booleans
  - b. Variables
    - i. Short Variable Declarations
    - ii. Zero Values
  - c. Constants
  - d. Pointers
  - e. Collections: Arrays, Slices, and Maps
    - i. Arrays
    - ii. Slices
    - iii. Maps
  - f. Control Structures
    - i. Fun With for
    - ii. The if Statement
    - iii. The switch Statement
  - g. Error Handling

- i. Creating an Error
  - h. Putting the Fun in Functions: Variadics and Closures
    - i. Functions
    - ii. Variadic Functions
    - iii. Anonymous Functions and Closures
  - i. Structs, Methods, and Interfaces
    - i. Structs
    - ii. Methods
    - iii. Interfaces
  - j. The Good Stuff: Concurrency
    - i. Goroutines
    - ii. Channels
    - iii. Select
6. 4. Cloud Native Patterns
- a. Layout of this Chapter
    - i. The Context Package
  - b. Stability Patterns
    - i. Circuit Breaker
    - ii. Debounce
    - iii. Retry
    - iv. Throttle
    - v. Timeout

### c. Concurrency Patterns

- i. Fan-In
- ii. Fan-Out
- iii. Future
- iv. Sharding

## 7. 5. Building a Cloud Native Service

### a. Let's Build A Service!

- i. What's a Key-Value Store?

### b. Requirements

- i. What Is Idempotence and Why Does It Matter?

- ii. The Eventual Goal

### c. Generation 0: The Core Functionality

- i. Our Super Simple API

### d. Generation 1: The Monolith

- i. Building an HTTP Server With net/http

- ii. Building an HTTP Server With gorilla/mux

- iii. Building our RESTful Service

- iv. Making Our Data Structure Concurrency-Safe

### e. Generation 2: Persisting Resource State

- i. What's a Transaction Log?

- ii. Storing State in a Transaction Log File

iii. Storing State in an External Database

f. Generation 3: Implementing Transport Layer Security

i. Transport Layer Security

ii. Private Key and Certificate Files

iii. Securing Our Web Service With HTTPS

iv. Transport Layer Summary

g. Containerizing Our Key-Value Store

i. Docker (Absolute) Basics

ii. Building Our Key-Value Store Container

iii. Externalizing Container Data

h. Summary

8. 6. It's All About Dependability

a. What's the Point of Cloud Native?

b. It's All About Dependability

c. What Is Dependability and Why Is It So Important?

d. Achieving Dependability

i. Fault Prevention

ii. Fault Tolerance

iii. Fault Removal

iv. Fault Forecasting

e. The Continuing Relevance of the Twelve-Factor App

i. I. Codebase

- ii. II. Dependencies
- iii. III. Configuration
- iv. IV. Backing Services
- v. V. Build, Release, Run
- vi. VI. Processes
- vii. VII. Data Isolation
- viii. VIII. Scalability
- ix. IX. Disposability
- x. X. Development/Production Parity
- xi. XI. Logs
- xii. XII. Administrative Processes

f. Summary

9. 7. Scalability

- a. What Is Scalability?
- b. The Focus of This Chapter
- c. Bottlenecks and Scaling Targets
- d. Horizontal vs. Vertical Scaling
- e. State and Statelessness
  - i. Application State vs. Resource State
  - ii. Advantages of Statelessness
- f. Scaling Postponed: Efficiency
  - i. Using an LRU Cache

- ii. Efficient Synchronization
- iii. Memory Leaks Can... fatal error: runtime: out of memory

- g. Service Architectures

- i. The Monolith System Architecture
- ii. The Microservices System Architecture
- iii. Serverless Architectures

- h. Summary

## 10. 8. Loose Coupling

- a. Loose and Tight Coupling
  - i. Tight Coupling Takes Many Forms
  - ii. Distributed Monoliths
- b. Communications Between Services
- c. Request-Response Messaging
  - i. Common Request-Response Implementations
  - ii. Issuing HTTP Requests With net/http
  - iii. Remote Procedure Calls With gRPC
- d. Loose Coupling Local Resources With Plugins
  - i. In-Process Plugins With the plugin Package
  - ii. HashiCorp's Go Plugin System Over RPC
  - iii. Hexagonal Architecture
- e. Summary

## 11. 9. Resilience

- a. Keeping On Ticking: Why Resilience Matters
- b. What Does It Mean for a System to Fail?
  - i. Building For Resilience
- c. Cascading Failures
  - i. Preventing Overload
- d. Play It Again: Retrying Requests
  - i. Backoff Algorithms
  - ii. Circuit Breaking
  - iii. Timeouts
  - iv. Idempotence
- e. Service Redundancy
  - i. Designing for Redundancy
  - ii. Autoscaling
- f. Healthy Health Checks
  - i. What Does It Mean for an Instance to Be “Healthy”?
  - ii. The Three Types of Health Checks
  - iii. Failing Open
- g. Summary