Go: Building Web Applications

Build real-world, production-ready solutions by harnessing the powerful features of Go

Packt

LEARNING PATH

Go: Building Web Applications

Build real-world, production-ready solutions by harnessing the powerful features of Go

A course in three modules

Packt

BIRMINGHAM - MUMBAI

Go: Building Web Applications

Published on: August 2016

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street

Birmingham B3 2PB, UK.

ISBN 978-1-78712-349-6

www.packtpub.com

Credits

Authors
Nathan Kozyra
Mat Ryer

Reviewers
Karthik Nayak
Tyler Bunnell
Michael Hamrah
Nimish Parmar
Jeremy R. Budnack
János Fehér
Aleksandar S. Sokolovski
Michele Della Torre

Content Development Editor
Arun Nadar

Graphics
Abhinash Sahu

Production Coordinator
Melwyn D'sa

Preface

Since the late 1980s and early 1990s, there has been a slow flood of powerful new languages and paradigms—Perl, Python, Ruby, PHP, and JavaScript—have taken an expanding user base by storm and has become one of the most popular languages (up there with stalwarts such as C, C++, and Java). Multithreading, memory caching, and APIs have allowed multiple processes, dissonant languages, applications, and even separate operating systems to work in congress.

And while this is great, there's a niche that until very recently was largely unserved: powerful, compiled, cross-platform languages with concurrency support that are geared towards systems programmers.

So when Google announced Go in 2009, which included some of the best minds in language design (and programming in general)—Rob Pike and Ken Thompson of Bell Labs fame and Robert Griesemer, who worked on Google's JavaScript implementation V8—to design a modern, concurrent language with development ease at the forefront..

For Go programming bright future, the team focused on some sore spots in the alternatives, which are as follows:

- Dynamically typed languages have—in recent years—become incredibly popular. Go eschews the explicit, "cumbersome" type systems of Java or C++. Go uses type inference, which saves development time, but is still also strongly typed.
- Concurrency, parallelism, pointers/memory access, and garbage collection are unwieldy in the aforementioned languages. Go lets these concepts be as easy or as complicated as you want or need them to be.
- As a newer language, Go has a focus on multicore design that was a necessary afterthought in languages such as C++.

- Go's compiler is super-fast; it's so fast that there are implementations of it that treat Go code as interpreted.
- Although Google designed Go to be a systems language, it's versatile enough to be used in a myriad of ways. Certainly, the focus on advanced, cheap concurrency makes it ideal for network and systems programming.
- Go is loose with syntax, but strict with usage. By this we mean that Go will let you get a little lazy with some lexer tokens, but you still have to produce fundamentally tight code. As Go provides a formatting tool that attempts to clarify your code, you can also spend less time on readability concerns as you're coding

What this learning path covers

Module 1, Learning Go Web Development, starts off with introducing and setting up Go before you move on to produce responsive servers that react to certain web endpoint. You will then implement database connections to acquire data and then present it to our users using different template packages. Later on, you will learn about sessions and cookies to retain information before delving with the basics of microservices. By the end of this module, we will be covering the testing, debugging, and the security aspect.

Module 2, Go Programming Blueprints, has a project-based approach where you will be building chat application, adding authentication, and adding your own profile pictures in different ways. You will learn how Go makes it easy to build powerful command-line tools to find domain names before building a highly scalable Twitter polling and vote counting engine powered by NSQ and MongoDB. Later on it covers the functionalities of RESTful Data Web Service API and Google Places API before you move on to build a simple but powerful filesystem backup tool for our code projects.

Module 3, Mastering Concurrency in Go, introduces you to Concurrency in Go where you will be understanding the Concurrency model and developing a strategy for designing applications. You will learn to create basic and complex communication channels between our goroutines to manage data not only across single or multithreaded systems but also distributed systems. Later on you will be tackling a real-world problem, that is, being able to develop a high performance web server that can handle a very large volume of live, active traffic. You will then learn how to scale your application and make it capable of being expanded in scope, design, and/or capacity. It will then focus on when and where to implement concurrent patterns, utilize parallelism, and ensure data consistency. At the end of this module, we will be logging and testing concurrency before we finally look at the best practices on how to implement complicated and advanced techniques offered by Go.

What you need for this learning path

For this course, any modern computers running a standard Linux flavor, OS X or Windows should be enough to get started. You can find a full list of requirements at `https://golang.org/dl/`. Later on, you'll also need to have the following software installed:

- MySQL (http://dev.mysql.com/downloads/)
- Couchbase (http://www.couchbase.com/download)

Your choice of IDE is a matter of personal preference.

Who this learning path is for

This course is intended for developers who are new to Go but have previous experience of building web applications and APIs. It is also targeted towards systems or network programmer with some knowledge of Go and concurrency, but would like to know about the implementation of concurrent systems written in Go

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this course—what you liked or disliked. Reader feedback is important for us as it helps us develop titles that you will really get the most out of.

To send us general feedback, simply e-mail `feedback@packtpub.com`, and mention the course's title in the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide at `www.packtpub.com/authors`.

Customer support

Now that you are the proud owner of a Packt course, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files for this course from your account at `http://www.packtpub.com`. If you purchased this course elsewhere, you can visit `http://www.packtpub.com/support` and register to have the files e-mailed directly to you.

You can download the code files by following these steps:

1. Log in or register to our website using your e-mail address and password.
2. Hover the mouse pointer on the **SUPPORT** tab at the top.
3. Click on **Code Downloads & Errata**.
4. Enter the name of the course in the **Search** box.
5. Select the course for which you're looking to download the code files.
6. Choose from the drop-down menu where you purchased this course from.
7. Click on **Code Download**.

You can also download the code files by clicking on the **Code Files** button on the course's webpage at the Packt Publishing website. This page can be accessed by entering the course's name in the **Search** box. Please note that you need to be logged in to your Packt account.

Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of:

- WinRAR / 7-Zip for Windows
- Zipeg / iZip / UnRarX for Mac
- 7-Zip / PeaZip for Linux

The code bundle for the course is also hosted on GitHub at `https://github.com/PacktPublishing/repository-name`. We also have other code bundles from our rich catalog of books and videos available at `https://github.com/PacktPublishing/`. Check them out!

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our courses—maybe a mistake in the text or the code—we would be grateful if you could report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this course. If you find any errata, please report them by visiting `http://www.packtpub.com/submit-errata`, selecting your course, clicking on the **Errata Submission Form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded to our website or added to any list of existing errata under the Errata section of that title.

To view the previously submitted errata, go to `https://www.packtpub.com/books/content/support` and enter the name of the course in the search field. The required information will appear under the **Errata** section.

Piracy

Piracy of copyrighted material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works in any form on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at `copyright@packtpub.com` with a link to the suspected pirated material.

We appreciate your help in protecting our authors and our ability to bring you valuable content.

Questions

If you have a problem with any aspect of this course, you can contact us at `questions@packtpub.com`, and we will do our best to address the problem.

Module 1: Learning Go Web Development

Module 2: Go Programming Blueprints

Module 1

Learning Go Web Development

Build frontend-to-backend web applications using the best practices of a powerful, fast, and easy-to-deploy server language

1 Introducing and Setting Up Go

When starting with Go, one of the most common things you'll hear being said is that it's a systems language.

Indeed, one of the earlier descriptions of Go, by the Go team itself, was that the language was built to be a modern systems language. It was constructed to combine the speed and power of languages, such as C with the syntactical elegance and thrift of modern interpreted languages, such as Python. You can see that goal realized when you look at just a few snippets of Go code.

From the Go FAQ on why Go was created:

> "*Go was born out of frustration with existing languages and environments for systems programming.*"

Perhaps the largest part of present-day Systems programming is designing backend servers. Obviously, the Web comprises a huge, but not exclusive, percentage of that world.

Go hasn't been considered a web language until recently. Unsurprisingly, it took a few years of developers dabbling, experimenting, and finally embracing the language to start taking it to new avenues.

While Go is web-ready out of the box, it lacks a lot of the critical frameworks and tools people so often take for granted with web development now. As the community around Go grew, the scaffolding began to manifest in a lot of new and exciting ways. Combined with existing ancillary tools, Go is now a wholly viable option for end-to-end web development. But back to that primary question: Why Go? To be fair, it's not right for every web project, but any application that can benefit from high-performance, secure web-serving out of the box with the added benefits of a beautiful concurrency model would make for a good candidate.

In this book, we're going to explore those aspects and others to outline what can make Go the right language for your web architecture and applications.

We're not going to deal with a lot of the low-level aspects of the Go language. For example, we assume you're familiar with variable and constant declaration. We assume you understand control structures.

In this chapter we will cover the following topics:

- Installing Go
- Structuring a project
- Importing packages
- Introducing the net package
- Hello, Web

Installing Go

The most critical first step is, of course, making sure that Go is available and ready to start our first web server.

> While one of Go's biggest selling points is its cross-platform support (both building and using locally while targeting other operating systems), your life will be much easier on a Nix compatible platform.
>
> If you're on Windows, don't fear. Natively, you may run into incompatible packages, firewall issues when running using `go run` command and some other quirks, but 95% of the Go ecosystem will be available to you. You can also, very easily, run a virtual machine, and in fact that is a great way to simulate a potential production environment.

In-depth installation instructions are available at `https://golang.org/doc/install`, but we'll talk about a few quirky points here before moving on.

For OS X and Windows, Go is provided as a part of a binary installation package. For any Linux platform with a package manager, things can be pretty easy.

> **To install via common Linux package managers:**
>
> Ubuntu: `sudo apt-get golang`
>
> CentOS: `sudo yum install golang`

On both OS X and Linux, you'll need to add a couple of lines to your path—the `GOPATH` and `PATH`. First, you'll want to find the location of your Go binary's installation. This varies from distribution to distribution. Once you've found that, you can configure the `PATH` and `GOPATH`, as follows:

```
export PATH=$PATH:/usr/local/go/bin
export GOPATH="/usr/share/go"
```

While the path to be used is not defined rigidly, some convention has coalesced around starting at a subdirectory directly under your user's home directory, such as `$HOME/go` or `~Home/go`. As long as this location is set perpetually and doesn't change, you won't run into issues with conflicts or missing packages.

You can test the impact of these changes by running the `go env` command. If you see any issues with this, it means that your directories are not correct.

Note that this may not prevent Go from running—depending on whether the GOBIN directory is properly set—but will prevent you from installing packages globally across your system.

To test the installation, you can grab any Go package by a `go get` command and create a Go file somewhere. As a quick example, first get a package at random, we'll use a package from the Gorilla framework, as we'll use this quite a bit throughout this book.

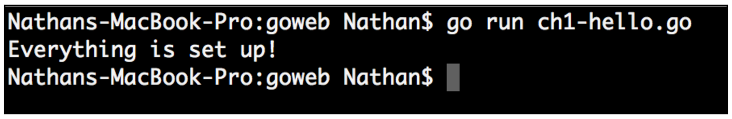
```
go get github.com/gorilla/mux
```

If this runs without any issue, Go is finding your `GOPATH` correctly. To make sure that Go is able to access your downloaded packages, draw up a very quick package that will attempt to utilize Gorilla's mux package and run it to verify whether the packages are found.

```
package main

import (
  "fmt"
  "github.com/gorilla/mux"
  "net/http"
)

func TestHandler(w http.ResponseWriter, r *http.Request) {

}
```

```
func main() {
  router := mux.NewRouter()
  router.HandleFunc("/test", TestHandler)
  http.Handle("/", router)
  fmt.Println("Everything is set up!")
}
```

Run `go run test.go` in the command line. It won't do much, but it will deliver the good news as shown in the following screenshot:



```
Nathans-MacBook-Pro:goweb Nathan$ go run ch1-hello.go
Everything is set up!
Nathans-MacBook-Pro:goweb Nathan$
```

Structuring a project

When you're first getting started and mostly playing around, there's no real problem with setting your application lazily.

For example, to get started as quickly as possible, you can create a simple `hello.go` file anywhere you like and compile without issue.

But when you get into environments that require multiple or distinct packages (more on that shortly) or have more explicit cross-platform requirements, it makes sense to design your project in a way that will facilitate the use of the go build tool.

The value of setting up your code in this manner lies in the way that the go build tool works. If you have local (to your project) packages, the build tool will look in the `src` directory first and then your `GOPATH`. When you're building for other platforms, go build will utilize the local bin folder to organize the binaries.

When building packages that are intended for mass use, you may also find that either starting your application under your `GOPATH` directory and then symbolically linking it to another directory, or doing the opposite, will allow you to develop without the need to subsequently go get your own code.

Code conventions

As with any language, being a part of the Go community means perpetual consideration of the way others create their code. Particularly if you're going to work in open source repositories, you'll want to generate your code the way that others do, in order to reduce the amount of friction when people get or include your code.

One incredibly helpful piece of tooling that the Go team has included is go `fmt`. `fmt` here, of course, means format and that's exactly what this tool does, it automatically formats your code according to the designed conventions.

By enforcing style conventions, the Go team has helped to mitigate one of the most common and pervasive debates that exist among a lot of other languages.

While the language communities tend to drive coding conventions, there are always little idiosyncrasies in the way individuals write programs. Let's use one of the most common examples around—where to put the opening bracket.

Some programmers like it on the same line as the statement:

```
for (int i = 0; i < 100; i++) {
  // do something
}
```

While others prefer it on the subsequent line:

```
for (int i = 0; i < 100; i++)
{
  // do something
}
```

These types of minor differences spark major, near-religious debates. The Gofmt tool helps alleviate this by allowing you to yield to Go's directive.

Now, Go bypasses this obvious source of contention at the compiler, by formatting your code similar to the latter example discussed earlier. The compiler will complain and all you'll get is a fatal error. But the other style choices have some flexibility, which are enforced when you use the tool to format.

Here, for example, is a piece of code in Go before `go fmt`:

```
func Double(n int) int {

  if (n == 0) {
    return 0
  } else {
    return n * 2
  }
}
```

Arbitrary whitespace can be the bane of a team's existence when it comes to sharing and reading code, particularly when every team member is not on the same IDE.

By running `go fmt`, we clean this up, thereby translating our whitespace according to Go's conventions:

```
func Double(n int) int {
  if n == 0 {
    return 0
  } else {
    return n * 2
  }
}
```

Long story short: always run `go fmt` before shipping or pushing your code.

Importing packages

Beyond the absolute and the most trivial application—one that cannot even produce a **Hello World** output—you must have some imported package in a Go application.

To say **Hello World**, for example, we'd need some sort of a way to generate an output. Unlike in many other languages, even the core language library is accessible by a namespaced package. In Go, namespaces are handled by a repository endpoint URL, which is `github.com/nkozyra/gotest`, which can be opened directly on Github (or any other public location) for the review.

Handling private repositories

The go get tool easily handles packages hosted at the repositories, such as Github, Bitbucket, and Google Code (as well as a few others). You can also host your own projects, ideally a git project, elsewhere, although it might introduce some dependencies and sources for errors, which you'd probably like to avoid.

But what about the private repos? While go get is a wonderful tool, you'll find yourself looking at an error without some additional configuration, SSH agent forwarding, and so on.

You can work around this in a couple of ways, but one very simple method is to clone the repository locally, using your version control software directly.

Dealing with versioning

You may have paused when you read about the way namespaces are defined and imported in a Go application. What happens if you're using version 1 of the application but would like to bring in version 2? In most cases, this has to be explicitly defined in the path of the `import`. For example:

```
import (
  "github.com/foo/foo-v1"
)
```

versus:

```
import (
  "github.com/foo/foo-v2"
)
```

As you might imagine, this can be a particularly sticky aspect of the way Go handles the remote packages.

Unlike a lot of other package managers, go get is decentralized—that is, nobody maintains a canonical reference library of packages and versions. This can sometimes be a sore spot for new developers.

For the most part, packages are always imported via the `go get` command, which reads the master branch of the remote repository. This means that maintaining multiple versions of a package at the same endpoint is, for the most part, impossible.

It's the utilization of the URL endpoints as namespaces that allows the decentralization, but it's also what provides a lack of internal support for versioning.

Your best bet as a developer is to treat every package as the most up-to-date version when you perform a `go get` command. If you need a newer version, you can always follow whatever pattern the author has decided on, such as the preceding example.

As a creator of your own packages, make sure that you also adhere to this philosophy. Keeping your master branch HEAD as the most up-to-date will make sure your that the code fits with the conventions of other Go authors.

Introducing the net package

At the heart of all network communications in Go is the aptly-named net package, which contains subpackages not only for the very relevant HTTP operations, but also for other TCP/UDP servers, DNS, and IP tools.

In short, everything you need to create a robust server environment.

Of course, what we care about for the purpose of this book lies primarily in the `net/http` package, but we'll look at a few other functions that utilize the rest of the package, such as a TCP connection, as well as WebSockets.

Let's quickly take a look at just performing that Hello World (or Web, in this case) example we have been talking about.

Hello, Web

The following application serves as a static file at the location `/static`, and a dynamic `response` at the location `/dynamic`:

```
package main

import (
  "fmt"
  "net/http"
  "time"
)

const (
  Port = ":8080"
)

func serveDynamic(w http.ResponseWriter, r *http.Request) {
  response := "The time is now " + time.Now().String()
  fmt.Fprintln(w,response)
}
```

Just as `fmt.Println` will produce desired content at the console level, `Fprintln` allows you to direct output to any writer. We'll talk a bit more about the writers in *Chapter 2*, *Serving and Routing*, but they represent a fundamental, flexible interface that is utilized in many Go applications, not just for the Web:

```
func serveStatic(w http.ResponseWriter, r *http.Request) {
  http.ServeFile(w, r, "static.html")
}
```

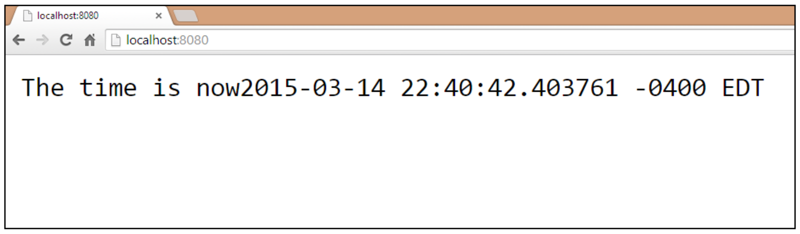
Our `serveStatic` method just serves one file, but it's trivial to allow it to serve any file directly and use Go as an old-school web server that serves only static content:

```
func main() {
  http.HandleFunc("/static",serveStatic)
  http.HandleFunc("/",serveDynamic)
  http.ListenAndServe(Port,nil)
}
```

Feel free to choose the available port of your choice—higher ports will make it easier to bypass the built-in security functionality, particularly in Nix systems.

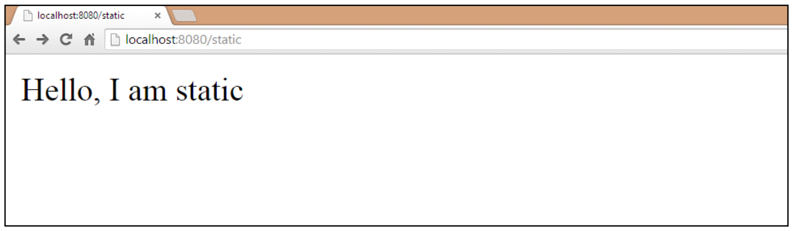
If we take the preceding example and visit the respective URLs—in this case the root at / and a static page at `/static`, we should see the intended output as shown:

At the root, / , the output is as follows:



```
The time is now2015-03-14 22:40:42.403761 -0400 EDT
```

At `/static`, the output is as follows:



Hello, I am static

As you can see, producing a very simple output for the Web is, well, very simple in Go. The built-in package allows us to create a basic, yet inordinately fast site in Go with just a few lines of code using native packages.

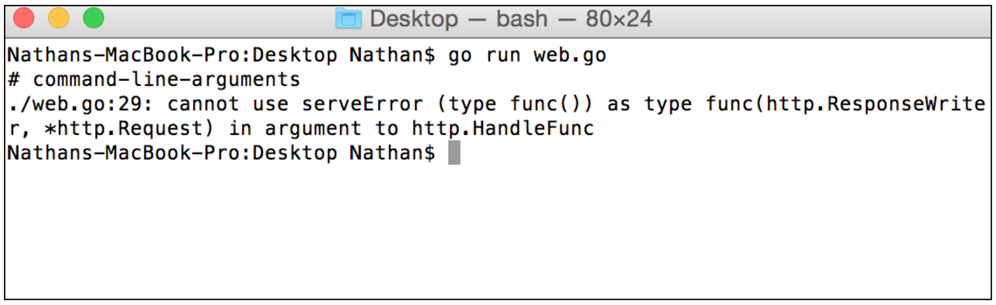
This may not be very exciting, but before we can run, we must walk. Producing the preceding output introduces a few key concepts.

First, we've seen how `net/http` directs requests using a URI or URL endpoint to helper functions, which must implement the `http.ResponseWriter` and `http.Request` methods. If they do not implement it, we get a very clear error on that end.

The following is an example that attempts to implement it in this manner:

```
func serveError() {
  fmt.Println("There's no way I'll work!")
}

func main() {
  http.HandleFunc("/static", serveStatic)
  http.HandleFunc("/", serveDynamic)
  http.HandleFunc("/error",serveError)
  http.ListenAndServe(Port, nil)
}
```

The following screenshot shows the resulting error you'll get from Go:



```
Nathans-MacBook-Pro:Desktop Nathan$ go run web.go
# command-line-arguments
./web.go:29: cannot use serveError (type func()) as type func(http.ResponseWrite
r, *http.Request) in argument to http.HandleFunc
Nathans-MacBook-Pro:Desktop Nathan$ 
```

You can see that `serveError` does not include the required parameters and thus results in a compilation error.

Summary

This chapter serves as an introduction to the most basic concepts of Go and producing for the Web in Go, but these points are critical foundational elements for being productive in the language and in the community.

We've looked at coding conventions and package design and organization, and we've produced our first program—the all-too-familiar Hello, World application—and accessed it via our localhost.

Obviously, we're a long way from a real, mature application for the Web, but the building blocks are essential to getting there.

In *Chapter 2*, *Serving and Routing*, we'll look at how to direct different requests to different application logic using the built-in routing functionality in Go's `net/http` package, as well as a couple of third party router packages.

2
Serving and Routing

The cornerstone of the Web as a commercial entity—the piece on which marketing and branding has relied on nearly exclusively—is the URL. While we're not yet looking at the top-level domain handling, we need to take up the reins of our URL and its paths (or endpoints).

In this chapter, we'll do just this by introducing multiple routes and corresponding handlers. First, we'll do this with a simple flat file serving and then we'll introduce complex mixers to do the routing with more flexibility by implementing a library that utilizes regular expressions in its routes.

By the end of this chapter, you should be able to create a site on localhost that can be accessed by any number of paths and return content relative to the requested path.

In this chapter, we will cover the following topics:

- Serving files directly
- Basic routing
- Using more complex routing with Gorilla
- Redirecting requests
- Serving basic errors

Serving files directly

In the preceding chapter, we utilized the `fmt.Fprintln` function to output some generic Hello, World messaging in the browser.

This obviously has limited utility. In the earliest days of the Web and web servers, the entirety of the Web was served by directing requests to corresponding static files. In other words, if a user requested `home.html`, the web server would look for a file called `home.html` and return it to the user.

This might seem quaint today, as a vast majority of the Web is now served in some dynamic fashion, with content often being determined via database IDs, which allows for pages to be generated and regenerated without someone modifying the individual files.

Let's take a look at the simplest way in which we can serve files in a way similar to those olden days of the Web as shown:

```
package main

import (
  "net/http"
)

const (
  PORT = ":8080"
)

func main() {

  http.ListenAndServe(PORT,
  http.FileServer(http.Dir("/var/www")))
}
```

Pretty simple, huh? Any requests made to the site will attempt to find a corresponding file in our local `/var/www` directory. But while this has a more practical use compared to the example in *Chapter 1, Introducing and Setting Up Go*, it's still pretty limited. Let's take a look at expanding our options a bit.

Basic routing

In *Chapter 1, Introducing and Setting Up*, we produced a very basic URL endpoint that allowed static file serving.

The following are the simple routes we produced for that example:

```
func main() {
  http.HandleFunc("/static",serveStatic)
  http.HandleFunc("/",serveDynamic)
  http.ListenAndServe(Port,nil)
}
```

In review, you can see two endpoints, `/static` and `/`, which either serve a single static file or generate output to the `http.ResponseWriter`.

We can have any number of routers sitting side by side. However, consider a scenario where we have a basic website with about, contact, and staff pages, with each residing in `/var/www/about/index.html`, `/var/www/contact.html`, and `/var/www/staff/home.html`. While it's an intentionally obtuse example, it demonstrates the limitations of Go's built-in and unmodified routing system. We cannot route all requests to the same directory locally, we need something that provides more malleable URLs.

Using more complex routing with Gorilla

In the previous session, we looked at basic routing but that can only take us so far, we have to explicitly define our endpoints and then assign them to handlers. What happens if we have a wildcard or a variable in our URL? This is an absolutely essential part of the Web and any serious web server.

To invoke a very simple example, consider hosting a blog with unique identifiers for each blog entry. This could be a numeric ID representing a database ID entry or a text-based globally unique identifier, such as `my-first-block-entry`.

> In the preceding example, we want to route a URL like `/pages/1` to a filename called `1.html`. Alternately, in a database-based scenario, we'd want to use `/pages/1` or `/pages/hello-world` to map to a database entry with a GUID of `1` or `hello-world`, respectively. To do this we either need to include an exhaustive list of possible endpoints, which is extremely wasteful, or implement wildcards, ideally through regular expressions.

In either case, we'd like to be able to utilize the value from the URL directly within our application. This is simple with URL parameters from `GET` or `POST`. We can extract those simply, but they aren't particularly elegant in terms of clean, hierarchical or descriptive URLs that are often necessary for search engine optimization purposes.

The built-in `net/http` routing system is, perhaps by design, relatively simple. To get anything more complicated out of the values in any given request, we either need to extend the routing capabilities or use a package that has done this.

In the few years that Go has been publicly available and the community has been growing, a number of web frameworks have popped up. We'll talk about these in a little more depth as we continue the book, but one in particular is well-received and very useful: the Gorilla web toolkit.

As the name implies, Gorilla is less of a framework and more of a set of very useful tools that are generally bundled in frameworks. Specifically, Gorilla contains:

- `gorilla/context`: This is a package for creating a globally-accessible variable from the request. It's useful for sharing a value from the URL without repeating the code to access it across your application.
- `gorilla/rpc`: This implements RPC-JSON, which is a system for remote code services and communication without implementing specific protocols. This relies on the JSON format to define the intentions of any request.
- `gorilla/schema`: This is a package that allows simple packing of form variables into a `struct`, which is an otherwise cumbersome process.
- `gorilla/securecookie`: This, unsurprisingly, implements authenticated and encrypted cookies for your application.
- `gorilla/sessions`: Similar to cookies, this provides unique, long-term, and repeatable data stores by utilizing a file-based and/or cookie-based session system.
- `gorilla/mux`: This is intended to create flexible routes that allow regular expressions to dictate available variables for routers.
- The last package is the one we're most interested in here, and it comes with a related package called `gorilla/reverse`, which essentially allows you to reverse the process of creating regular expression-based muxes. We will cover that topic in detail in the later section.

You can grab individual Gorilla packages by their GitHub location with a `go get`. For example, to get the mux package, going to `github.com/gorilla/mux` will suffice and bring the package into your `GOPATH`. For the locations of the other packages (they're fairly self-explanatory), visit `http://www.gorillatoolkit.org/`

Let's dive-in and take a look at how to create a route that's flexible and uses a regular expression to pass a parameter to our handler:

```
package main

import (
  "github.com/gorilla/mux"
  "net/http"
)

const (
  PORT = ":8080"
)
```

This should look familiar to our last code with the exception of the Gorilla package import:

```
func pageHandler(w http.ResponseWriter, r *http.Request) {
  vars := mux.Vars(r)
  pageID := vars["id"]
  fileName := "files/" + pageID + ".html"
  http.ServeFile(w,r,fileName)
}
```

Here, we've created a route handler to accept the response. The thing to be noted here is the use of `mux.Vars`, which is a method that will look for query string variables from the `http.Request` and parse them into a map. The values will then be accessible by referencing the result by key, in this case `id`, which we'll cover in the next section.

```
func main() {
  rtr := mux.NewRouter()
  rtr.HandleFunc("/pages/{id:[0-9]+}",pageHandler)
  http.Handle("/",rtr)
  http.ListenAndServe(PORT,nil)
}
```

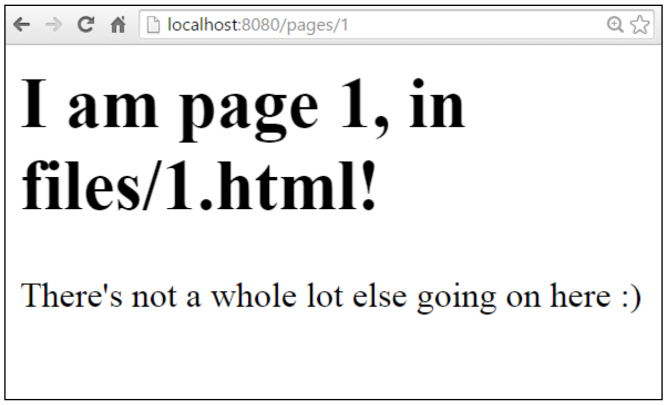
Here, we can see a (very basic) regular expression in the handler. We're assigning any number of digits after `/pages/` to a parameter named `id` in `{id:[0-9]+}`; this is the value we pluck out in `pageHandler`.

A simpler version that shows how this can be used to delineate separate pages can be seen by adding a couple of dummy endpoints:

```
func main() {
  rtr := mux.NewRouter()
  rtr.HandleFunc("/pages/{id:[0-9]+}", pageHandler)
  rtr.HandleFunc("/homepage", pageHandler)
  rtr.HandleFunc("/contact", pageHandler)
  http.Handle("/", rtr)
  http.ListenAndServe(PORT, nil)
}
```

When we visit a URL that matches this pattern, our `pageHandler` attempts to find the page in the `files/` subdirectory and returns that file directly.

A response to `/pages/1` would look like this:


localhost:8080/pages/1

I am page 1, in files/1.html!

There's not a whole lot else going on here :)

At this point, you might already be asking, but what if we don't have the requested page? Or, what happens if we've moved that location? This brings us to two important mechanisms in web serving—returning error responses and, as part of that, potentially redirecting requests that have moved or have other interesting properties that need to be reported back to the end users.

Redirecting requests

Before we look at simple and incredibly common errors like 404s, let's address the idea of redirecting requests, something that's very common. Although not always for reasons that are evident or tangible for the average user.

So we might we want to redirect requests to another request? Well there are quite a few reasons, as defined by the HTTP specification that could lead us to implement automatic redirects on any given request. Here are a few of them with their corresponding HTTP status codes:

- A non-canonical address may need to be redirected to the canonical one for SEO purposes or for changes in site architecture. This is handled by *301 Moved Permanently* or *302 Found*.
- Redirecting after a successful or unsuccessful `POST`. This helps us to prevent re-POSTing of the same form data accidentally. Typically, this is defined by *307 Temporary Redirect*.
- The page is not necessarily missing, but it now lives in another location. This is handled by the status code *301 Moved Permanently*.

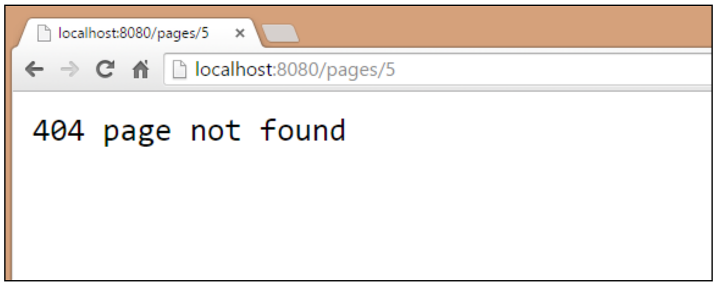
Executing any one of these is incredibly simple in basic Go with `net/http`, but as you might expect, it is facilitated and improved with more robust frameworks, such as Gorilla.

Serving basic errors

At this point, it makes some sense to talk a bit about errors. In all likelihood, you may have already encountered one as you played with our basic flat file serving server, particularly if you went beyond two or three pages.

Our example code includes four example HTML files for flat serving, numbered `1.html`, `2.html`, and so on. What happens when you hit the `/pages/5` endpoint, though? Luckily, the `http` package will automatically handle the file not found errors, just like most common web servers.

Also, similar to most common web servers, the error page itself is small, bland, and nondescript. In the following section, you can see the **404 page not found** status response we get from Go:



```
404 page not found
```

As mentioned, it's a very basic and nondescript page. Often, that's a good thing—error pages that contain more information or flair than necessary can have a negative impact.

Consider this error—the `404`—as an example. If we include references to images and stylesheets that exist on the same server, what happens if those assets are also missing?

In short, you can very quickly end up with recursive errors—each `404` page calls an image and stylesheet that triggers `404` responses and the cycle repeats. Even if the web server is smart enough to stop this, and many are, it will produce a nightmare scenario in the logs, rendering them so full of noise that they become useless.

Let's look at some code that we can use to implement a catch-all `404` page for any missing files in our `/files` directory:

```
package main

import (
  "github.com/gorilla/mux"
  "net/http"
  "os"
)

const (
  PORT = ":8080"
)

func pageHandler(w http.ResponseWriter, r *http.Request) {
  vars := mux.Vars(r)
  pageID := vars["id"]
  fileName := "files/" + pageID + ".html"_,
  err := os.Stat(fileName)
    if err != nil {
      fileName = "files/404.html"
    }

  http.ServeFile(w,r,fileName)
}
```

Here, you can see that we first attempt to check the file with `os.Stat` (and its potential error) and output our own `404` response:

```
func main() {
  rtr := mux.NewRouter()
  rtr.HandleFunc("/pages/{id:[0-9]+}",pageHandler)
  http.Handle("/",rtr)
  http.ListenAndServe(PORT,nil)
}
```

Now if we take a look at the `404.html` page, we will see that we've created a custom HTML file that produces something that is a little more user-friendly than the default **Go Page Not Found** message that we were invoking previously.

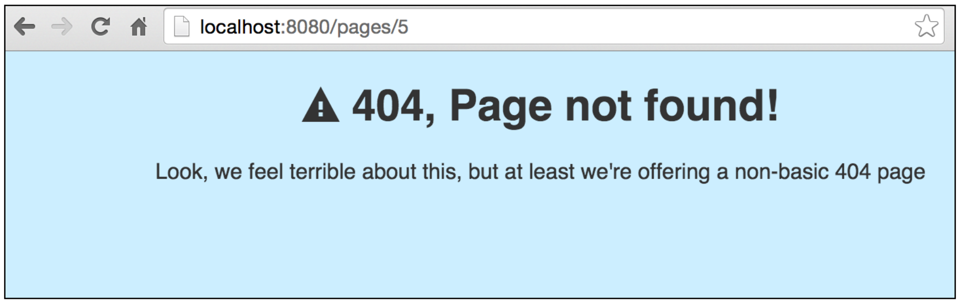
Let's take a look at what this looks like, but remember that it can look any way you'd like:

```
<!DOCTYPE html>
<html>
<head>
<title>Page not found!</title>
<style type="text/css">
body {
  font-family: Helvetica, Arial;
  background-color: #cceeff;
  color: #333;
  text-align: center;
}
</style>
<link rel="stylesheet" type="text/css" media="screen"
href="http://code.ionicframework.com/ionicons/2.0.1/css/ion
icons.min.css"></link>
</head>

<body>
<h1><i class="ion-android-warning"></i> 404, Page not found!</h1>
<div>Look, we feel terrible about this, but at least we're offer
ing a non-basic 404 page</div>
</body>

</html>
```

Also, note that while we keep the `404.html` file in the same directory as the rest of our files, this is solely for the purposes of simplicity.

In reality, and in most production environments with custom error pages, we'd much rather have it exist in its own directory, which is ideally outside the publicly available part of our web site. After all, you can now access the error page in a way that is not actually an error by visiting `http://localhost:8080/pages/404`. This returns the error message, but the reality is that in this case the file was found, and we're simply returning it.

Let's take a look at our new, prettier `404` page by accessing `http://localhost/pages/5`, which specifies a static file that does not exist in our filesystem:



404, Page not found!

Look, we feel terrible about this, but at least we're offering a non-basic 404 page

By showing a more user-friendly error message, we can provide more useful actions for users who encounter them. Consider some of the other common errors that might benefit from more expressive error pages.

Summary

We can now produce not only the basic routes from the `net/http` package but more complicated ones using the Gorilla toolkit. By utilizing Gorilla, we can now create regular expressions and implement pattern-based routing and allow much more flexibility to our routing patterns.

With this increased flexibility, we also have to be mindful of errors now, so we've looked at handling error-based redirects and messages, including a custom **404, Page not found** message to produce more customized error messages.

Now that we have the basics down for creating endpoints, routes, and handlers; we need to start doing some non-trivial data serving.

In *Chapter 3, Connecting to Data*, we'll start getting dynamic information from databases, so we can manage data in a smarter and more reliable fashion. By connecting to a couple of different, commonly-used databases, we'll be able to build robust, dynamic, and scalable web applications.

3
Connecting to Data

In the previous chapter, we explored how to take URLs and translate them to different pages in our web application. In doing so, we built URLs that were dynamic and resulted in dynamic responses from our (very simple) `net/http` handlers.

By implementing an extended mux router from the Gorilla toolkit, we expanded the capabilities of the built-in router by allowing regular expressions, which gives our application a lot more flexibility.

This is something that's endemic to some of the most popular web servers. For example, both Apache and Nginx provide methods to utilize regular expressions in routes and staying at par with common solutions should be our minimal baseline for functionality.

But this is just an admittedly important stepping stone to build a robust web application with a lot of varied functionality. To go any further, we need to look at bringing in data.

Our examples in the previous chapter relied on hardcoded content grabbed from static files—this is obviously archaic and doesn't scale. Anyone who has worked in the pre-CGI early days of the Web could regale you with tales of site updates requiring total retooling of static files or explain the anachronism that was Server-Side Includes.

But luckily, the Web became largely dynamic in the late 1990s and databases began to rule the world. While APIs, microservices and NoSQL have in some places replaced that architecture, it still remains the bread and butter of the way the Web works today.

So without further ado, let's get some dynamic data.

In this chapter, we will cover the following topics:

- Connecting to a database

- Using GUID for prettier URLs
- Handling 404s

Connecting to a database

When it comes to accessing databases, Go's SQL interface provides a very simple and reliable way to connect to various database servers that have drivers.

At this point, most of the big names are covered—MySQL, Postgres, SQLite, MSSQL, and quite a few more have well-maintained drivers that utilize the `database/sql` interface provided by Go.

The best thing about the way Go handles this through a standardized SQL interface is that you won't have to learn custom Go libraries to interact with your database. This doesn't preclude needing to know the nuances of the database's SQL implementation or other functionality, but it does eliminate one potential area of confusion.

Before you go too much farther, you'll want to make sure that you have a library and a driver for your database of choice installed via `go get` command.

The Go project maintains a Wiki of all of the current SQLDrivers and is a good starting reference point when looking for an adapter at `https://github.com/golang/go/wiki/SQLDrivers`

[Note: We're using MySQL and Postgres for various examples in this book, but use the solution that works best for you. Installing MySQL and Postgres is fairly basic on any Nix, Windows, or OS X machine.]

MySQL can be downloaded from `https://www.mysql.com/` and although there are a few drivers listed by Google, we recommend the Go-MySQL-Driver. Though you won't go wrong with the recommended alternatives from the Go project, the Go-MySQL-Driver is very clean and well-tested. You can get it at `https://github.com/go-sql-driver/mysql/`

For Postgres, grab a binary or package manager command from `http://www.postgresql.org/`. The Postgres driver of choice here is `pq`, which can be installed via `go get` at `github.com/lib/pq`

Creating a MySQL database

You can choose to design any application you wish, but for these examples we'll look at a very simple blog concept.

Our goal here is to have as few blog entries in our database as possible, to be able to call those directly from our database by GUID and display an error if the particular requested blog entry does not exist.

To do this, we'll create a MySQL database that contains our pages. These will have an internal, automatically incrementing numeric ID, a textual globally unique identifier, or GUID, and some metadata around the blog entry itself.

To start simply, we'll create a title `page_title`, body text `page_content` and a Unix timestamp `page_date`. You can feel free to use one of MySQL's built-in date fields; using an integer field to store a timestamp is just a matter of preference and can allow for some more elaborate comparisons in your queries.

The following is the SQL in your MySQL console (or GUI application) to create the database `cms` and the requisite table `pages`:

```
CREATE TABLE `pages` (
  `id` int(11) unsigned NOT NULL AUTO_INCREMENT,
  `page_guid` varchar(256) NOT NULL DEFAULT '',
  `page_title` varchar(256) DEFAULT NULL,
  `page_content` mediumtext,
  `page_date` timestamp NOT NULL DEFAULT CURRENT_TIMESTAMP ON
UPDATE CURRENT_TIMESTAMP,
  PRIMARY KEY (`id`),
  UNIQUE KEY `page_guid` (`page_guid`)
) ENGINE=InnoDB AUTO_INCREMENT=2 DEFAULT CHARSET=latin1;
```

As mentioned, you can execute this query through any number of interfaces. To connect to MySQL, select your database and try these queries, you can follow the command line documentation at `http://dev.mysql.com/doc/refman/5.7/en/connecting.html`.

Note the `UNIQUE KEY` on `page_guid`. This is pretty important, as if we happen to allow duplicate GUIDs, well, we have a problem. The idea of a globally unique key is that it cannot exist elsewhere, and since we'll rely on it for URL resolution, we want to make sure that there's only one entry per GUID.

As you can probably tell, this is a very basic content type of blog database. We have an auto-incrementing ID value, a title, a date and the page's content, and not a whole lot else going on.

While it's not a lot, it's enough to demonstrate dynamic pages in Go utilizing a database interface.

Just to make sure there's some data in the `pages` table, add the following query to fill this in a bit:

```
INSERT INTO `pages` (`id`, `page_guid`, `page_title`,
`page_content`, `page_date`) VALUES (NULL, 'hello-world', 'Hello,
World', 'I\'m so glad you found this page!  It\'s been sitting
patiently on the Internet for some time, just waiting for a
visitor.', CURRENT_TIMESTAMP);
```

This will give us something to start with.

Now that we have our structure and some dummy data, let's take a look at how we can connect to MySQL, retrieve the data, and serve it dynamically based on URL requests and Gorilla's mux patterns.

To get started, let's create a shell of what we'll need to connect:

```
package main

import (
  "database/sql"
  "fmt"
  _ "github.com/go-sql-driver/mysql"
  "log"
)
```

We're importing the MySQL driver package for what's known as *side effects*. By this, it's generally meant that the package is complementary to another and provides various interfaces that do not need to be referenced specifically.

You can note this through the underscore _ syntax that precedes the packages import. You're likely already familiar with this as a quick-and-dirty way to ignore the instantiation of a returned value from a method. For example `x, _ := something()` allows you to ignore the second returned value.

It's also often used when a developer plans to use a library, but hasn't yet. By prepending the package this way, it allows the import declaration to stay without causing a compiler error. While this is frowned upon, the use of the underscore—or blank identifier—in the preceding method, for side effects, is fairly common and often acceptable.

As always, though, this all depends on how and why you're using the identifier:

```
const (
  DBHost  = "127.0.0.1"
  DBPort  = ":3306"
  DBUser  = "root"
  DBPass   = "password!"
  DBDbase = "cms"
)
```

Make sure to replace these values with whatever happens to be relevant to your installation, of course:

```
var database *sql.DB
```

By keeping our database connection reference as a global variable, we can avoid a lot of duplicate code. For the sake of clarity, we'll define it fairly high up in the code. There's nothing preventing you from making this a constant instead, but we've left it mutable for any necessary future flexibility, such as adding multiple databases to a single application:

```
type Page struct {
  Title   string
  Content string
  Date    string
}
```

This `struct`, of course, matches our database schema rather closely, with `Title`, `Content` and `Date` representing the non-ID values in our table. As we'll see a bit later in this chapter (and more in the next), describing our data in a nicely-designed struct helps parlay the templating functions of Go. And on that note, make sure your struct fields are exportable or public by keeping them propercased. Any lowercased fields will not be exportable and therefore not available to templates. We will talk more on that later:

```
func main() {
  dbConn := fmt.Sprintf("%s:%s@tcp(%s)/%s", DBUser, DBPass,
DBHost, DBDbase)
  db, err := sql.Open("mysql", dbConn)
  if err != nil {
    log.Println("Couldn't connect!")
    log.Println(err.Error)
  }
  database = db
}
```

As we mentioned earlier, this is largely scaffolding. All we want to do here is ensure that we're able to connect to our database. If you get an error, check your connection and the log entry output after `Couldn't connect`.

If, hopefully, you were able to connect with this script, we can move on to creating a generic route and outputting the relevant data from that particular request's GUID from our database.

To do this we need to reimplement Gorilla, create a single route, and then implement a handler that generates some very simple output that matches what we have in the database.

Let's take a look at the modifications and additions we'll need to make to allow this to happen:

```
package main

import (
  "database/sql"
  "fmt"
  _ "github.com/go-sql-driver/mysql"
  "github.com/gorilla/mux"
  "log"
  "net/http"
)
```

The big change here is that we're bringing Gorilla and `net/http` back into the project. We'll obviously need these to serve pages:

```
const (
  DBHost  = "127.0.0.1"
  DBPort  = ":3306"
  DBUser  = "root"
  DBPass  = "password!"
  DBDbase = "cms"
  PORT    = ":8080"
)
```

We've added a `PORT` constant, which refers to our HTTP server port.

Note that if your host is `localhost`/`127.0.0.1`, it's not necessary to specify a `DBPort`, but we've kept this line in the constants section. We don't use the host here in our MySQL connection:

```
var database *sql.DB

type Page struct {
  Title   string
  Content string
  Date    string
}

func ServePage(w http.ResponseWriter, r *http.Request) {
  vars := mux.Vars(r)
  pageID := vars["id"]
  thisPage := Page{}
  fmt.Println(pageID)
  err := database.QueryRow("SELECT page_title,page_content,page_date
FROM pages WHERE id=?",
pageID).Scan(&thisPage.Title, &thisPage.Content, &thisPage.Date)
  if err != nil {

    log.Println("Couldn't get page: +pageID")
    log.Println(err.Error)
  }
  html := `<html><head><title>` + thisPage.Title +
`</title></head><body><h1>` + thisPage.Title + `</h1><div>` +
thisPage.Content + `</div></body></html>`
  fmt.Fprintln(w, html)
}
```

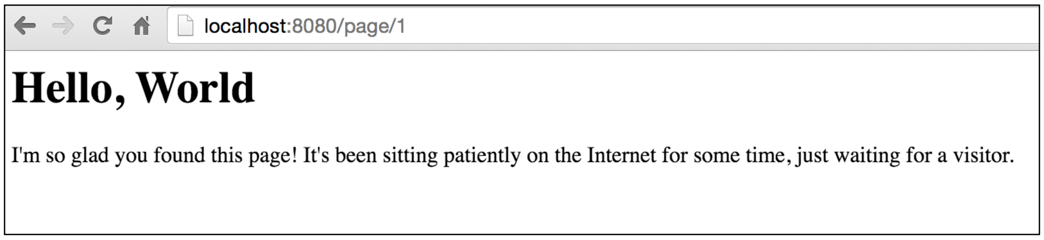
`ServePage` is the function that takes an `id` from `mux.Vars` and queries our database for the blog entry ID. There's some nuance in the way we make a query that is worth noting; the simplest way to eliminate SQL injection vulnerabilities is to use prepared statements, such as `Query`, `QueryRow`, or `Prepare`. Utilizing any of these and including a variadic of variables to be injected into the prepared statement removes the inherent risk of constructing a query by hand.

The `Scan` method then takes the results of a query and translates them to a struct; you'll want to make sure the struct matches the order and number of requested fields in the query. In this case, we're mapping `page_title`, `page_content` and `page_date` to a `Page` struct's `Title`, `Content` and `Date`:

```
func main() {
  dbConn := fmt.Sprintf("%s:%s@/%s", DBUser, DBPass, DBDbase)
  fmt.Println(dbConn)
  db, err := sql.Open("mysql", dbConn)
  if err != nil {
    log.Println("Couldn't connect to"+DBDbase)
    log.Println(err.Error)
  }
  database = db

  routes := mux.NewRouter()
  routes.HandleFunc("/page/{id:[0-9]+}", ServePage)
  http.Handle("/", routes)
  http.ListenAndServe(PORT, nil)

}
```

Note our regular expression here: it's just numeric, with one or more digits comprising what will be the `id` variable accessible from our handler.

Remember that we talked about using the built-in GUID? We'll get to that in a moment, but for now let's look at the output of `localhost:8080/page/1`:



Hello, World

I'm so glad you found this page! It's been sitting patiently on the Internet for some time, just waiting for a visitor.

In the preceding example, we can see the blog entry that we had in our database. This is good, but obviously lacking in quite a few ways.

Using GUID for prettier URLs

Earlier in this chapter we talked about using the GUID to act as the URL identifier for all requests. Instead, we started by yielding to the numeric, thus automatically incrementing column in the table. That was for the sake of simplicity, but switching this to the alphanumeric GUID is trivial.

All we'll need to do is to switch our regular expression and change our resulting SQL query in our `ServePage` handler.

If we only change our regular expression, our last URL's page will still work:

```
routes.HandleFunc("/page/{id:[0-9a-zA\\-]+}", ServePage)
```

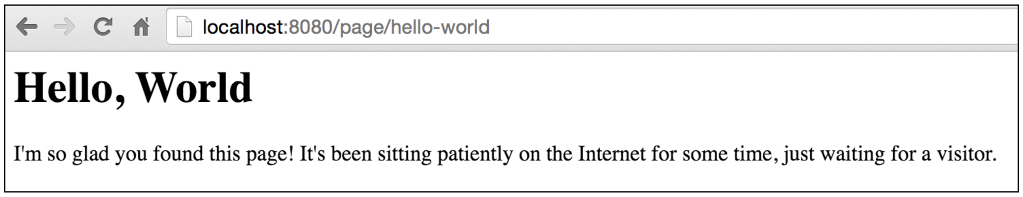
The page will of course still pass through to our handler. To remove any ambiguity, let's assign a `guid` variable to the route:

```
routes.HandleFunc("/page/{guid:[0-9a-zA\\-]+}", ServePage)
```

After that, we change our resulting call and SQL:

```
func ServePage(w http.ResponseWriter, r *http.Request) {
  vars := mux.Vars(r)
  pageGUID := vars["guid"]
  thisPage := Page{}
  fmt.Println(pageGUID)
  err := database.QueryRow("SELECT page_title,page_content,page_date
FROM pages WHERE page_guid=?",
pageGUID).Scan(&thisPage.Title, &thisPage.Content, &thisPage.Date)
```

After doing this, accessing our page by the `/pages/hello-world` URL will result in the same page content we got by accessing it through `/pages/1`. The only real advantage is cosmetic, it creates a prettier URL that is more human-readable and potentially more useful for search engines:


localhost:8080/page/hello-world

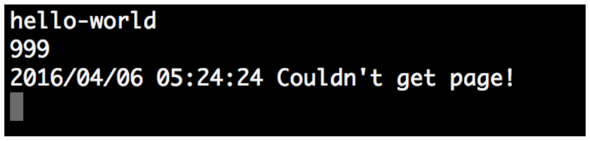
Hello, World

I'm so glad you found this page! It's been sitting patiently on the Internet for some time, just waiting for a visitor.

Handling 404s

A very obvious problem with our preceding code is that it does not handle a scenario wherein an invalid ID (or GUID) is requested.

As it is, a request to, say, `/page/999` will just result in a blank page for the user and in the background a **Couldn't get page!** message, as shown in the following screenshot:

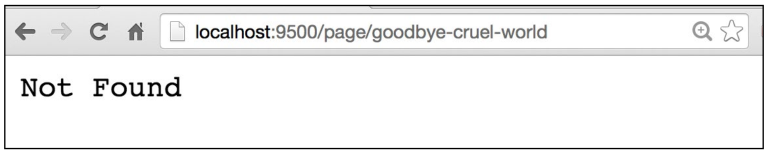

```
hello-world
999
2016/04/06 05:24:24 Couldn't get page!
```

Resolving this is pretty simple by passing proper errors. Now, in the previous chapter we explored custom `404` pages and you can certainly implement one of those here, but the easiest way is to just return an HTTP status code when a post cannot be found and allow the browser to handle the presentation.

In our preceding code, we have an error handler that doesn't do much except return the issue to our log file. Let's make that more specific:

```
  err := database.QueryRow("SELECT
page_title,page_content,page_date FROM pages WHERE page_guid=?",
pageGUID).Scan(&thisPage.Title, &thisPage.Content, &thisPage.Date)
  if err != nil {
    http.Error(w, http.StatusText(404), http.StatusNotFound)
    log.Println("Couldn't get page!")
  }
```

You will see the output in the following screenshot. Again, it would be trivial to replace this with a custom `404` page, but for now we want to make sure we're addressing the invalid requests by validating them against our database:



```
Not Found
```

Providing good error messages helps improve usability for both developers and other users. In addition, it can be beneficial for SEO, so it makes sense to use HTTP status codes as defined in HTTP standards.

Summary

In this chapter, we've taken the leap from simply showing content to showing content that's maintained in a sustainable and maintainable way using a database. While this allows us to display dynamic data easily, it's just a core step toward a fully-functional application.

We've looked at creating a database and then retrieving the data from it to inject into route while keeping our query parameters sanitized to prevent SQL injections.

We also accounted for potential bad requests with invalid GUIDs, by returning *404 Not Found* statuses for any requested GUID that does not exist in our database. We also looked at requesting data by ID as well as the alphanumeric GUID.

This is just the start of our application, though.

In *Chapter 4*, *Using Templates*, we'll take the data that we've grabbed from MySQL (and Postgres) and apply some of Go's template language to them to give us more frontend flexibility.

By the end of that chapter, we will have an application that allows for creation and deletion of pages directly from our application.

4
Using Templates

In *Chapter 2*, *Serving and Routing*, we explored how to take URLs and translate them to different pages in our web application. In doing so, we built URLs that were dynamic and resulted in dynamic responses from our (very simple) `net/http` handlers.

We've presented our data as real HTML, but we specifically hard-coded our HTML directly into our Go source. This is not ideal for production-level environments for a number of reasons.

Luckily, Go comes equipped with a robust but sometimes tricky template engine for both text templates, as well as HTML templates.

Unlike a lot of other template languages that eschew logic as a part of the presentation side, Go's template packages enable you to utilize some logic constructs, such as loops, variables, and function declarations in a template. This allows you to offset some of your logic to the template, which means that it's possible to write your application, but you need to allow the template side to provide some extensibility to your product without rewriting the source.

We say some logic constructs because Go templates are sold as logic-less. We will discuss more on this topic later.

In this chapter, we'll explore ways to not only present your data but also explore some of the more advanced possibilities in this chapter. By the end, we will be able to parlay our templates into advancing the separation of presentation and source code.

We will cover the following topics:

- Introducing templates, context, and visibility
- HTML templates and text templates
- Displaying variables and security
- Using logic and control structures

Introducing templates, context, and visibility

It's worth noting very early that while we're talking about taking our HTML part out of the source code, it's possible to use templates inside our Go application. Indeed, there's nothing wrong with declaring a template as shown:

```
tpl, err := template.New("mine").Parse(`<h1>{{.Title}}</h1>`)
```

If we do this, however, we'll need to restart our application every time the template needs to change. This doesn't have to be the case if we use file-based templates; instead we can make changes to the presentation (and some logic) without restarting.

The first thing we need to do to move from in-application HTML strings to file-based templates is create a template file. Let's briefly look at an example template that somewhat approximates to what we'll end up with later in this chapter:

```
<!DOCTYPE html>
<html>
<head>
<title>{{.Title}}</title>
</head>
<body>
  <h1>{{.Title}}</h1>

  <div>{{.Date}}</div>

  {{.Content}}
</body>
</html>
```

Very straightforward, right? Variables are clearly expressed by a name within double curly brackets. So what's with all of the periods/dots? Not unlike a few other similarly-styled templating systems (Mustache, Angular, and so on), the dot signifies scope or context.

The easiest way to demonstrate this is in areas where the variables might otherwise overlap. Imagine that we have a page with a title of **Blog Entries** and we then list all of our published blog articles. We have a page title but we also have individual entry titles. Our template might look something similar to this:

```
{{.Title}}
{{range .Blogs}}
  <li><a href="{{.Link}}">{{.Title}}</a></li>
{{end}}
```

The dot here specifies the specific scope of, in this case, a loop through the range template operator syntax. This allows the template parser to correctly utilize `{{.Title}}` as a blog's title versus the page's title.

This is all noteworthy because the very first templates we'll be creating will utilize general scope variables, which are prefixed with the dot notation.

HTML templates and text templates

In our first example of displaying the values from our blog from our database to the Web, we produced a hardcoded string of HTML and injected our values directly.

Following are the two lines that we used in *Chapter 3*, *Connecting to Data*:

```
  html := `<html><head><title>` + thisPage.Title +
`</title></head><body><h1>` + thisPage.Title + `</h1><div>` +
thisPage.Content + `</div></body></html>
  fmt.Fprintln(w, html)
```

It shouldn't be hard to realize why this isn't a sustainable system for outputting our content to the Web. The best way to do this is to translate this into a template, so we can separate our presentation from our application.

To do this as succinctly as possible, let's modify the method that called the preceding code, `ServePage`, to utilize a template instead of hardcoded HTML.

So we'll remove the HTML we placed earlier and instead reference a file that will encapsulate what we want to display. From your root directory, create a `templates` subdirectory and `blog.html` within it.

The following is the very basic HTML we included, feel free to add some flair:

```
<html>
<head>
<title>{{.Title}}</title>
</head>
<body>
  <h1>{{.Title}}</h1>
  <p>
    {{.Content}}
  </p>
  <div>{{.Date}}</div>
</body>
</html>
```

Back in our application, inside the `ServePage` handler, we'll change our output code slightly to leave an explicit string and instead parse and execute the HTML template we just created:

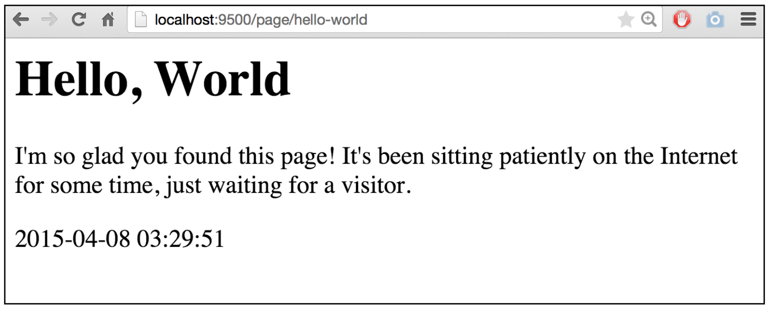
```
func ServePage(w http.ResponseWriter, r *http.Request) {
  vars := mux.Vars(r)
  pageGUID := vars["guid"]
  thisPage := Page{}
  fmt.Println(pageGUID)
  err := database.QueryRow("SELECT
page_title,page_content,page_date FROM pages WHERE page_guid=?",
pageGUID).Scan(&thisPage.Title, &thisPage.Content, &thisPage.Date)
  if err != nil {
    http.Error(w, http.StatusText(404), http.StatusNotFound)
    log.Println("Couldn't get page!")
    return
  }
  // html := <html>...</html>

  t, _ := template.ParseFiles("templates/blog.html")
  t.Execute(w, thisPage)
}
```

If, somehow, you failed to create the file or it is otherwise not accessible, the application will panic when it attempts to execute. You can also get panicked if you're referencing `struct` values that don't exist—we'll need to handle errors a bit better.

Note: Don't forget to include `html/template` in your imports.

The benefits of moving away from a static string should be evident, but we now have the foundation for a much more extensible presentation layer.

If we visit `http://localhost:9500/page/hello-world` we'll see something similar to this:



localhost:9500/page/hello-world

Hello, World

I'm so glad you found this page! It's been sitting patiently on the Internet for some time, just waiting for a visitor.

2015-04-08 03:29:51

Displaying variables and security

To demonstrate this, let's create a new blog entry by adding this SQL command to your MySQL command line:

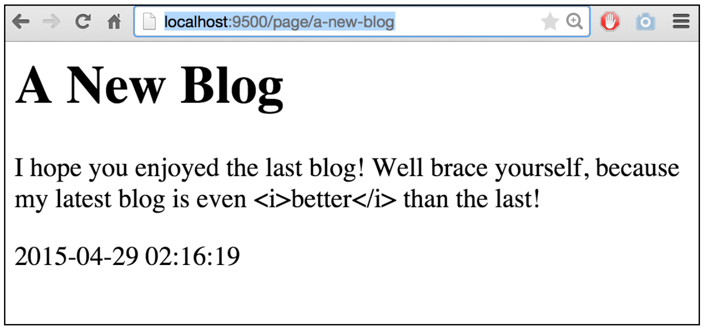
```
INSERT INTO `pages` (`id`, `page_guid`, `page_title`,
page_content`, `page_date`)
```

VALUES:

```
  (2, 'a-new-blog', 'A New Blog', 'I hope you enjoyed the last
blog!  Well brace yourself, because my latest blog is even
<i>better</i> than the last!', '2015-04-29 02:16:19');
```

Another thrilling piece of content, for sure. Note, however that we have some embedded HTML in this when we attempt to italicize the word better.

Debates about how formatting should be stored notwithstanding, this allows us to take a look at how Go's templates handle this by default. If we visit `http://localhost:9500/page/a-new-blog` we'll see something similar to this:



localhost:9500/page/a-new-blog

A New Blog

I hope you enjoyed the last blog! Well brace yourself, because my latest blog is even <i>better</i> than the last!

2015-04-29 02:16:19

As you can see, Go automatically sanitizes our data for output. There are a lot of very, very wise reasons to do this, which is why it's the default behavior. The biggest one, of course, is to avoid XSS and code-injection attack vectors from untrusted sources of input, such as the general users of the site and so on.

But ostensibly we are creating this content and should be considered trusted. So in order to validate this as trusted HTML, we need to change the type of `template.HTML`:

```
type Page struct {
  Title   string
  Content template.HTML
  Date   string
}
```

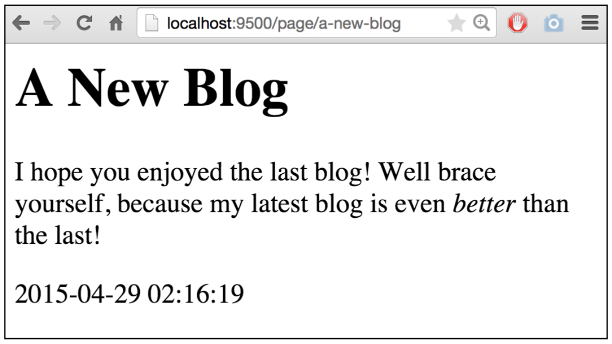
If you attempt to simply scan the resulting SQL string value into a `template.HTML` you'll find the following error:

```
sql: Scan error on column index 1: unsupported driver -> Scan
pair: []uint8 -> *template.HTML
```

The easiest way to work around this is to retain the string value in `RawContent` and assign it back to `Content`:

```
type Page struct {
  Title    string
  RawContent string
  Content    template.HTML
  Date    string
}
  err := database.QueryRow("SELECT
page_title,page_content,page_date FROM pages WHERE page_guid=?",
pageGUID).Scan(&thisPage.Title, &thisPage.RawContent,
&thisPage.Date)
  thisPage.Content = template.HTML(thisPage.RawContent)
```

If we go run this again, we'll see our HTML as trusted:



A New Blog

I hope you enjoyed the last blog! Well brace yourself, because my latest blog is even *better* than the last!

2015-04-29 02:16:19

Using logic and control structures

Earlier in this chapter we looked at how we can use a range in our templates just as we would directly in our code. Take a look at the following code:

```
{{range .Blogs}}
  <li><a href="{{.Link}}">{{.Title}}</a></li>
{{end}}
```

You may recall that we said that Go's templates are without any logic, but this depends on how you define logic and whether shared logic lies exclusively in the application, the template, or a little of both. It's a minor point, but because Go's templates offer a lot of flexibility; it's the one worth thinking about.

Having a range feature in the preceding template, by itself, opens up a lot of possibilities for a new presentation of our blog. We can now show a list of blogs or break our blog up into paragraphs and allow each to exist as a separate entity. This can be used to allow relationships between comments and paragraphs, which have started to pop up as a feature in some publication systems in recent years.

But for now, let's use this opportunity to create a list of blogs in a new index page. To do this, we'll need to add a route. Since we have /page we could go with /pages, but since this will be an index, let's go with / and /home:

```
routes := mux.NewRouter()
routes.HandleFunc("/page/{guid:[0-9a-zA\\-]+}", ServePage)
routes.HandleFunc("/", RedirIndex)
routes.HandleFunc("/home", ServeIndex)
http.Handle("/", routes)
```

We'll use `RedirIndex` to automatically redirect to our `/home` endpoint as a canonical home page.

Serving a simple `301` or `Permanently Moved` redirect requires very little code in our method, as shown:

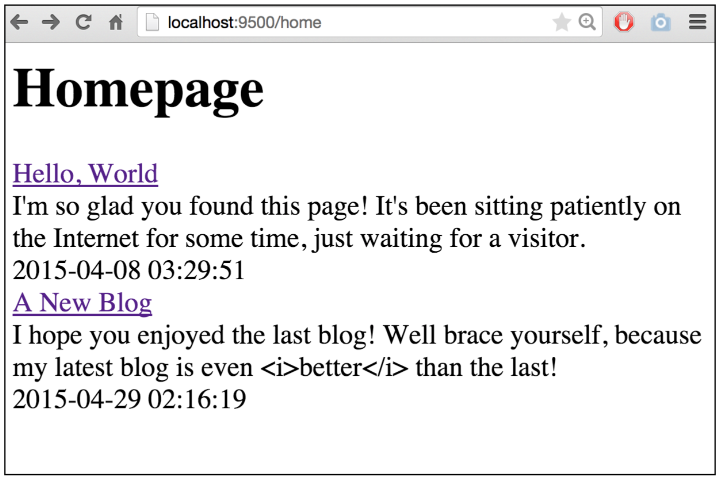
```
func RedirIndex(w http.ResponseWriter, r *http.Request) {
  http.Redirect(w, r, "/home", 301)
}
```

This is enough to take any requests from `/` and bring the user to `/home` automatically. Now, let's look at looping through our blogs on our index page in the `ServeIndex` HTTP handler:

```
func ServeIndex(w http.ResponseWriter, r *http.Request) {
  var Pages = []Page{}
  pages, err := database.Query("SELECT page_title,page_content,page_
date FROM pages ORDER BY ? DESC",
"page_date")
  if err != nil {
    fmt.Fprintln(w, err.Error)
  }
  defer pages.Close()
  for pages.Next() {
    thisPage := Page{}
    pages.Scan(&thisPage.Title, &thisPage.RawContent,
&thisPage.Date)
    thisPage.Content = template.HTML(thisPage.RawContent)
    Pages = append(Pages, thisPage)
  }
  t, _ := template.ParseFiles("templates/index.html")
  t.Execute(w, Pages)
}
```

And here's `templates/index.html`:

```
<h1>Homepage</h1>

{{range .}}
  <div><a href="!">{{.Title}}</a></div>
  <div>{{.Content}}</div>
  <div>{{.Date}}</div>
{{end}}
```



localhost:9500/home

Homepage

Hello, World
I'm so glad you found this page! It's been sitting patiently on the Internet for some time, just waiting for a visitor.
2015-04-08 03:29:51
A New Blog
I hope you enjoyed the last blog! Well brace yourself, because my latest blog is even <i>better</i> than the last!
2015-04-29 02:16:19

We've highlighted an issue with our `Page struct` here—we have no way to get the reference to the page's `GUID`. So, we need to modify our `struct` to include that as the exportable `Page.GUID` variable:

```
type Page struct {
  Title  string
  Content  template.HTML
  RawContent  string
  Date  string
  GUID   string
}
```

Now, we can link our listings on our index page to their respective blog entries as shown:

```
  var Pages = []Page{}
  pages, err := database.Query("SELECT page_title,page_content,page_
date,page_guid FROM pages ORDER BY ?
DESC", "page_date")
  if err != nil {
    fmt.Fprintln(w, err.Error)
  }
  defer pages.Close()
  for pages.Next() {
    thisPage := Page{}
    pages.Scan(&thisPage.Title, &thisPage.Content, &thisPage.Date,
&thisPage.GUID)
    Pages = append(Pages, thisPage)
  }
```

And we can update our HTML part with the following code:

```
<h1>Homepage</h1>

{{range .}}
  <div><a href="/page/{{.GUID}}">{{.Title}}</a></div>
  <div>{{.Content}}</div>
  <div>{{.Date}}</div>
{{end}}
```

But this is just the start of the power of the templates. What if we had a much longer piece of content and wanted to truncate its description?

We can create a new field within our `Page struct` and truncate that. But that's a little clunky; it requires the field to always exist within a `struct`, whether populated with data or not. It's much more efficient to expose methods to the template itself.

So let's do that.

First, create yet another blog entry, this time with a larger content value. Choose whatever you like or select the `INSERT` command as shown:

```
INSERT INTO `pages` (`id`, `page_guid`, `page_title`,
`page_content`, `page_date`)
```

VALUES:

```
  (3, 'lorem-ipsum', 'Lorem Ipsum', 'Lorem ipsum dolor sit amet,
consectetur adipiscing elit. Maecenas sem tortor, lobortis in
posuere sit amet, ornare non eros. Pellentesque vel lorem sed nisl
dapibus fringilla. In pretium...', '2015-05-06 04:09:45');
```

Note: For the sake of brevity, we've truncated the full length of our preceding Lorem Ipsum text.

Now, we need to represent our truncation as a method for the type `Page`. Let's create that method to return a string that represents the shortened text.

The cool thing here is that we can essentially share a method between the application and the template:

```
func (p Page) TruncatedText() string {
  chars := 0
  for i, _ := range p.Content {
    chars++
    if chars > 150 {
      return p.Content[:i] + ` ...`
```

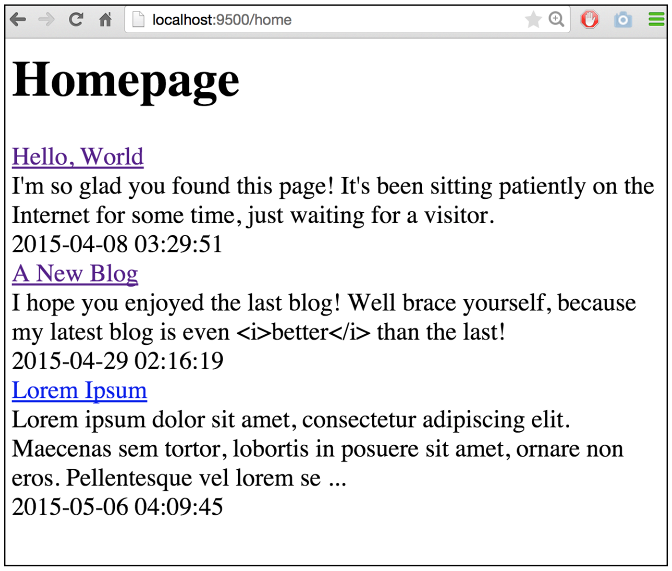
```
        }
      }
      return p.Content
    }
```

This code will loop through the length of content and if the number of characters exceeds `150`, it will return the slice up to that number in the index. If it doesn't ever exceed that number, `TruncatedText` will return the content as a whole.

Calling this in the template is simple, except that you might be expected to need a traditional function syntax call, such as `TruncatedText()`. Instead, it's referenced just as any variable within the scope:

```
<h1>Homepage</h1>

{{range .}}
  <div><a href="/page/{{.GUID}}">{{.Title}}</a></div>
  <div>{{.TruncatedText}}</div>
  <div>{{.Date}}</div>
{{end}}
```

By calling `.TruncatedText`, we essentially process the value inline through that method. The resulting page reflects our existing blogs and not the truncated ones and our new blog entry with truncated text and ellipsis appended:



Homepage

Hello, World
I'm so glad you found this page! It's been sitting patiently on the Internet for some time, just waiting for a visitor.
2015-04-08 03:29:51
A New Blog
I hope you enjoyed the last blog! Well brace yourself, because my latest blog is even <i>better</i> than the last!
2015-04-29 02:16:19
Lorem Ipsum
Lorem ipsum dolor sit amet, consectetur adipiscing elit. Maecenas sem tortor, lobortis in posuere sit amet, ornare non eros. Pellentesque vel lorem se ...
2015-05-06 04:09:45

I'm sure you can imagine how being able to reference embedded methods directly in your templates can open up a world of presentation possibilities.

Summary

We've just scratched the surface of what Go's templates can do and we'll explore further topics as we continue, but this chapter has hopefully introduced the core concepts necessary to start utilizing templates directly.

We've looked at simple variables, as well as implementing methods within the application, within the templates themselves. We've also explored how to bypass injection protection for trusted content.

In the next chapter, we'll integrate a backend API for accessing information in a RESTful way to read and manipulate our underlying data. This will allow us to do some more interesting and dynamic things on our templates with Ajax.

5
Frontend Integration with RESTful APIs

In *Chapter 2*, *Serving and Routing*, we explored how to route URLs to the different pages in our web application. In doing so, we built URLs that were dynamic and resulted in dynamic responses from our (very simple) `net/http` handlers.

We've just scratched the surface of what Go's templates can do, and we'll also explore further topics as we continue, but in this chapter we have tried to introduce the core concepts that are necessary to start utilizing the templates directly.

We've looked at simple variables as well as the implementing methods within the application using the templates themselves. We've also explored how to bypass injection protection for trusted content.

The presentation side of web development is important, but it's also the least engrained aspect. Almost any framework will present its own extension of built-in Go templating and routing syntaxes. What really takes our application to the next level is building and integrating an API for both general data access, as well as allowing our presentation layer to be more dynamically driven.

In this chapter, we'll develop a backend API for accessing information in a RESTful way and to read and manipulate our underlying data. This will allow us to do some more interesting and dynamic things in our templates with Ajax.

In this chapter, we will cover the following topics:

- Setting up the basic API endpoint
- RESTful architecture and best practices
- Creating our first API endpoint
- Implementing security

- Creating data with POST
- Modifying data with PUT

Setting up the basic API endpoint

First, we'll set up a basic API endpoint for both pages and individual blog entries.

We'll create a Gorilla endpoint route for a `GET` request that will return information about our pages and an additional one that accepts a GUID, which matches alphanumeric characters and hyphens:

```
routes := mux.NewRouter()
routes.HandleFunc("/api/pages", APIPage).
  Methods("GET").
  Schemes("https")
routes.HandleFunc("/api/pages/{guid:[0-9a-zA\\-]+}", APIPage).
  Methods("GET").
  Schemes("https")
routes.HandleFunc("/page/{guid:[0-9a-zA\\-]+}", ServePage)
http.Handle("/", routes)
http.ListenAndServe(PORT, nil)
```

Note here that we're capturing the GUID again, this time for our `/api/pages/*` endpoint, which will mirror the functionality of the web-side counterpart, returning all meta data associated with a single page.

```
func APIPage(w http.ResponseWriter, r *http.Request) {
vars := mux.Vars(r)
pageGUID := vars["guid"]
thisPage := Page{}
fmt.Println(pageGUID)
err := database.QueryRow("SELECT page_title,page_content,page_date
FROM pages WHERE page_guid=?", pageGUID).Scan(&thisPage.Title,
&thisPage.RawContent, &thisPage.Date)
thisPage.Content = template.HTML(thisPage.RawContent)
if err != nil {
  http.Error(w, http.StatusText(404), http.StatusNotFound)
  log.Println(err)
  return
}
APIOutput, err := json.Marshal(thisPage)
    fmt.Println(APIOutput)
if err != nil {
  http.Error(w, err.Error(), http.StatusInternalServerError)
```

```
    return
  }
  w.Header().Set("Content-Type", "application/json")
  fmt.Fprintln(w, thisPage)
  }
```

The preceding code represents the simplest GET-based request, which returns a single record from our `/pages` endpoint. Let's take a look at REST now, and see how we'll structure and implement other verbs and data manipulations from the API.

RESTful architecture and best practices

In the world of web API design, there has been an array of iterative, and sometimes competing, efforts to find a standard system and format to deliver information across multiple environments.

In recent years, the web development community at large seems to have—at least temporarily—settled on REST as the de facto approach. REST came after a few years of SOAP dominance and introduced a simpler method for sharing data.

REST APIs aren't bound to a format and are typically cacheable and delivered via HTTP or HTTPS.

The biggest takeaway to start with is an adherence to HTTP verbs; those initially specified for the Web are honored in their original intent. For example, HTTP verbs, such as `DELETE` and `PATCH` fell into years of disuse despite being very explicit about their purpose. REST has been the primary impetus for the use of the right method for the right purpose. Prior to REST, it was not uncommon to see `GET` and `POST` requests being used interchangeably to do myriad things that were otherwise built into the design of HTTP.

In REST, we follow a **Create-Read-Update-Delete** (**CRUD**)-like approach to retrieve or modify data. `POST` is used majorly to create, `PUT` is used as an update (though it can also be used to create), the familiar `GET` is used to read and `DELETE` is used to delete, is well, just that.

Perhaps even more important is the fact that a RESTful API should be stateless. By that we mean that each request should exist on its own, without the server necessarily having any knowledge about prior or potential future requests. This means that the idea of a session would technically violate this ethos, as we'd be storing some sense of state on the server itself. Some people disagree; we'll look at this in detail later on.

One final note is on API URL structure, because the method is baked into the request itself as part of the header, we don't need to explicitly express that in our request.

In other words, we don't need something, such as `/api/blogs/delete/1`. Instead, we can simply make our request with the `DELETE` method to `api/blogs/1`.

There is no rigid format of the URL structure and you may quickly discover that some actions lack HTTP-specific verbs that make sense, but in short there are a few things we should aim for:

- The resources are expressed cleanly in the URL
- We properly utilize HTTP verbs
- We return appropriate responses based on the type of request

Our goal in this chapter is to hit the preceding three points with our API.

If there is a fourth point, it would say that we maintain backwards compatibility with our APIs. As you examine the URL structure here, you might wonder how versions are handled. This tends to vary from organization to organization, but a good policy is to keep the most recent URL canonical and deprecate to explicit version URLs.

For example, even though our comments will be accessible at `/api/comments`, the older versions will be found at `/api/v2.0/comments`, where `2` obviously represents our API, as it existed in version `2.0`.

Despite being relatively simple and well-defined in nature, REST is an oft-argued subject with enough ambiguity to start, most often for the better, a lot of debate. Remember that REST is not a standard; for example, the W3C has not and likely will not ever weigh in on what REST is and isn't. If you haven't already, you'll begin to develop some very strong opinions on what you feel is properly RESTful.

Creating our first API endpoint

Given that we want to access data from the client-side as well as from server to server, we'll need to start making some of that accessible via an API.

The most reasonable thing for us to do is a simple read, since we don't yet have methods to create data outside of direct SQL queries. We did that at the beginning of the chapter with our `APIPage` method, routed through a `/api/pages/{UUID}` endpoint.

This is great for `GET` requests, where we're not manipulating data, but if we need to create or modify data, we'll need to utilize other HTTP verbs and REST methods. To do this effectively, it's time to investigate some authentication and security in our API.

Implementing security

When you think about creating data with an API like the one we've just designed, what's the first concern that comes to your mind? If it was security, then good for you. Accessing data is not always without a security risk, but it's when we allow for modification of data that we need to really start thinking about security.

In our case, read data is totally benign. If someone can access all of our blog entries via a `GET` request, who cares? Well, we may have a blog on embargo or accidentally exposed sensitive data on some resource.

Either way, security should always be a concern, even with a small personal project like a blogging platform, similar to the one we're building.

There are two big ways of separating these concerns:

- Are the requests to our APIs secure and private?
- Are we controlling access to data?

Lets tackle Step 2 first. If we want to allow users to create or delete information, we need to give them specific access to that.

There are a few ways to do this:

We can provide API tokens that will allow short-lived request windows, which can be validated by a shared secret. This is the essence of Oauth; it relies on a shared secret to validate cryptographically encoded requests. Without the shared secret, the request and its token will never match, and an API request can then be rejected.

The `cond` method is a simple API key, which leads us back to point number 1 in the preceding list.

If we allow cleartext API keys, then we might as well not have security at all. If our requests can be sniffed off the wire without much effort, there's little point in even requiring an API key.

So this means that no matter which method we choose, our servers should provide an API over HTTPS. Luckily, Go provides a very easy way to utilize either HTTP or HTTPS via **Transport Layer Security** (**TLS**); TLS is the successor of SSL. As a web developer, you must already be familiar with SSL and also be aware of its history of security issues, most recently its susceptibility to the POODLE vulnerability, which was exposed in 2014.

To allow either method, we need to have a user registration model so that we can have new users and they can have some sort of credentials to modify data. To invoke a TLS server, we'll need a secure certificate. Since this is a small project for experimentation, we won't worry too much about a real certificate with a high level of trust. Instead, we'll just generate our own.

Creating a self-signed certificate varies by OS and is beyond the scope of this book, so let's just look at the method for OS X.

A self-signed certificate doesn't have a lot of security value, obviously, but it allows us to test things without needing to spend money or time verifying server ownership. You'll obviously need to do those things for any certificate that you expect to be taken seriously.

To create a quick set of certificates in OS X, go to your terminal and enter the following three commands:

```
openssl genrsa -out key.pem

openssl req -new -key key.pem -out cert.pem

openssl req -x509 -days 365 -key key.pem -in cert.pem -out certifi
cate.pem
```

In this example, I generated the certificates using an OpenSSL on Ubuntu.

> Note: OpenSSL comes preinstalled on OS X and most Linux distributions. If you're on the latter, give the preceding commands a shot before hunting for Linux-specific instructions. If you're on Windows, particularly newer versions such as 8, you can do this in a number of ways, but the most accessible way might be through the MakeCert tool, provided by Microsoft through MSDN.
>
> Read more about MakeCert at `https://msdn.microsoft.com/en-us/library/bfsktky3%28v=vs.110%29.aspx`.

Once you have the certificate files, place them somewhere in your filesystem that is not within your accessible application directory/directories.

To switch from HTTP to TLS, we can use the references to these certificate files; beyond that it's mostly the same in our code. Lets first add the certificates to our code.

> Note: Once again, you can choose to maintain both HTTP and TLS/HTTPS requests within the same server application, but we'll be switching ours across the board.

Earlier, we started our server by listening through this line:

```
http.ListenAndServe(PORT, nil)
```

Now, we'll need to expand things a bit. First, let's load our certificate:

```
  certificates, err := tls.LoadX509KeyPair("cert.pem", "key.pem")
  tlsConf := tls.Config{Certificates:
[]tls.Certificate{certificates}}
  tls.Listen("tcp", PORT, &tlsConf)
```

> Note: If you find that your server apparently runs without error but does not stay running; there's probably a problem with your certificate. Try running the preceding generation code again and working with the new certificates.

Creating data with POST

Now that we have a security certificate in place, we can switch to TLS for our API calls for both `GET` and other requests. Let's do that now. Note that you can retain HTTP for the rest of our endpoints or switch them at this point as well.

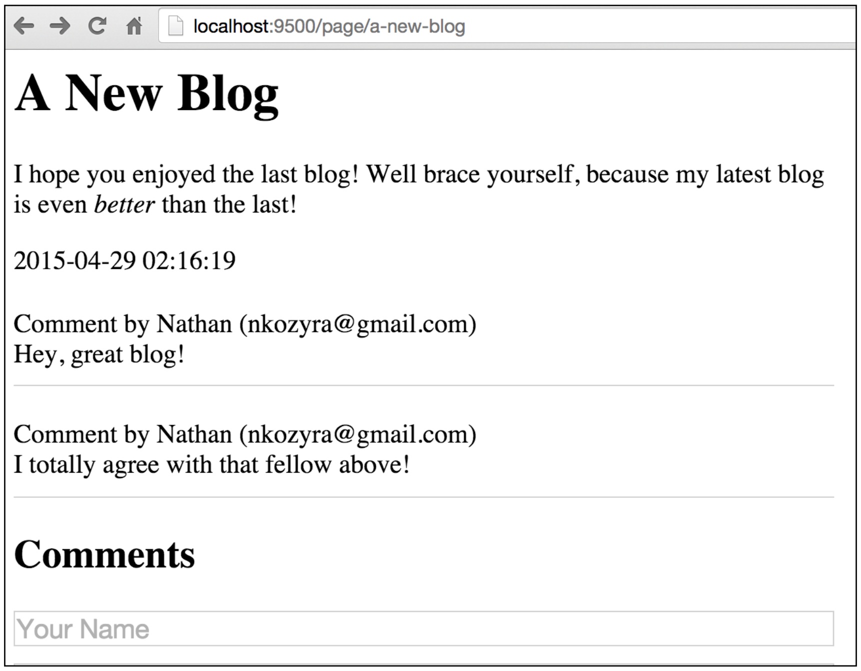
> Note: It's largely becoming a common practice to go the HTTPS-only way and it's probably the best way to future-proof your app. This doesn't solely apply to APIs or areas where explicit and sensitive information is otherwise sent in cleartext, with privacy on the forefront; major providers and services are stressing on the value of HTTPS everywhere.

Lets add a simple section for anonymous comments on our blog:

```
<div id="comments">
  <form action="/api/comments" method="POST">
    <input type="hidden" name="guid" value="{{Guid}}" />
    <div>
      <input type="text" name="name" placeholder="Your Name" />
    </div>
```

```
        <div>
          <input type="email" name="email" placeholder="Your Email" />
        </div>
        <div>
          <textarea name="comments" placeholder="Your Com-
    ments"></textarea>
        </div>
        <div>
          <input type="submit" value="Add Comments" />
        </div>
      </form>
    </div>
```

This will allow any user to add anonymous comments to our site on any of our blog items, as shown in the following screenshot:


localhost:9500/page/a-new-blog

A New Blog

I hope you enjoyed the last blog! Well brace yourself, because my latest blog is even *better* than the last!

2015-04-29 02:16:19

Comment by Nathan (nkozyra@gmail.com)
Hey, great blog!

Comment by Nathan (nkozyra@gmail.com)
I totally agree with that fellow above!

Comments

But what about all the security? For now, we just want to create an open comment section, one that anyone can post to with their valid, well-stated thoughts as well as their spammy prescription deals. We'll worry about locking that down shortly; for now we just want to demonstrate a side-by-side API and frontend integration.

We'll obviously need a `comments` table in our database, so make sure you create that before implementing any of the API:

```
CREATE TABLE `comments` (
`id` int(11) unsigned NOT NULL AUTO_INCREMENT,
`page_id` int(11) NOT NULL,
`comment_guid` varchar(256) DEFAULT NULL,
`comment_name` varchar(64) DEFAULT NULL,
`comment_email` varchar(128) DEFAULT NULL,
`comment_text` mediumtext,
`comment_date` timestamp NULL DEFAULT NULL,
PRIMARY KEY (`id`),
KEY `page_id` (`page_id`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1;
```

With the table in place, let's take our form and `POST` it to our API endpoint. To create a general purpose and a flexible JSON response, you can add a `JSONResponse struct` that consists of essentially a hash-map, as shown:

```
type JSONResponse struct {
  Fields map[string]string
}
```

Then we'll need an API endpoint to create comments, so let's add that to our routes under `main()`:

```
func APICommentPost(w http.ResponseWriter, r *http.Request) {
  var commentAdded bool
  err := r.ParseForm()
  if err != nil {
    log.Println(err.Error)
  }
  name := r.FormValue("name")
  email := r.FormValue("email")
  comments := r.FormValue("comments")

  res, err := database.Exec("INSERT INTO comments SET com
ment_name=?, comment_email=?, comment_text=?", name, email, com
ments)

  if err != nil {
    log.Println(err.Error)
  }
```

```
    id, err := res.LastInsertId()
    if err != nil {
      commentAdded = false
    } else {
      commentAdded = true
    }
    commentAddedBool := strconv.FormatBool(commentAdded)
    var resp JSONResponse
    resp.Fields["id"] = string(id)
    resp.Fields["added"] =  commentAddedBool
    jsonResp, _ := json.Marshal(resp)
    w.Header().Set("Content-Type", "application/json")
    fmt.Fprintln(w, jsonResp)
  }
```

There are a couple of interesting things about the preceding code:

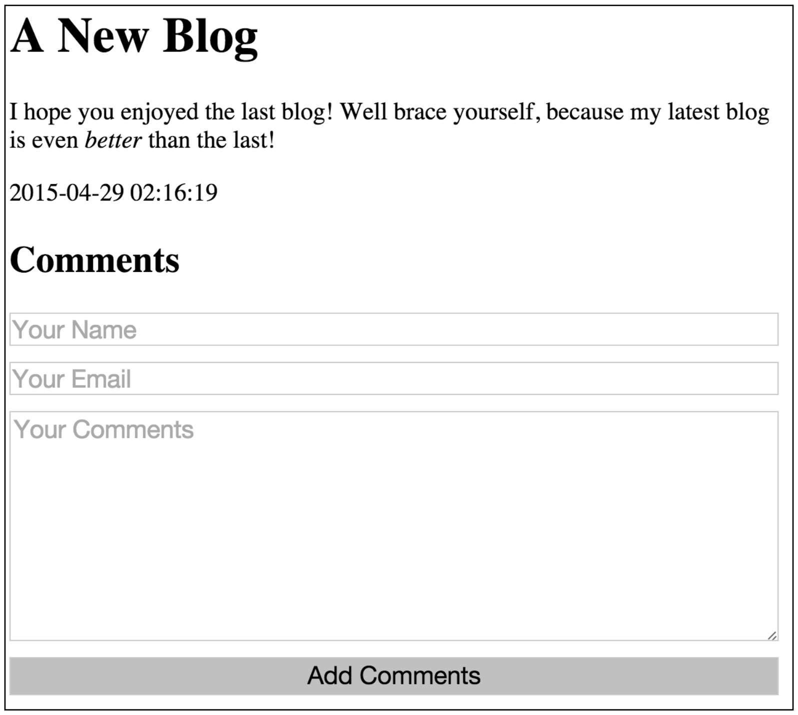
First, note that we're using `commentAdded` as a `string` and not a `bool`. We're doing this largely because the json marshaller does not handle booleans elegantly and also because casting directly to a string from a boolean is not possible. We also utilize `strconv` and its `FormatBool` to handle this translation.

You might also note that for this example, we're POSTing the form directly to the API endpoint. While an effective way to demonstrate that data makes it into the database, utilizing it in practice might force some RESTful antipatterns, such as enabling a redirect URL to return to the calling page.

A better way to approach this is through the client by utilizing an Ajax call through a common library or through `XMLHttpRequest` natively.

Note: While internal functions/method names are largely a matter of preference, we recommend keeping all methods distinct by resource type and request method. The actual convention used here is irrelevant, but as a matter of traversing the code, something such as `APICommentPost`, `APICommentGet`, `APICommentPut`, and `APICommentDelete` gives you a nice hierarchical way of organizing the methods for better readability.

Given the preceding client-side and server-side code, we can see how this will appear to a user hitting our second blog entry:



A New Blog

I hope you enjoyed the last blog! Well brace yourself, because my latest blog is even *better* than the last!

2015-04-29 02:16:19

Comments

As mentioned, actually adding your comments here will send the form directly to the API endpoint, where it will silently succeed (hopefully).

Modifying data with PUT

Depending on whom you ask, `PUT` and `POST` can be used interchangeably for the creation of records. Some people believe that both can be used for updating the records and most believe that both can be used for the creation of records given a set of variables. In lieu of getting into a somewhat confusing and often political debate, we've separated the two as follows:

- Creation of new records: `POST`
- Updating existing records, idempotently: `PUT`

Given these guidelines, we'll utilize the `PUT` verb when we wish to make updates to resources. We'll allow comments to be edited by anyone as nothing more than a proof of concept to use the REST `PUT` verb.

In *Chapter 6*, *Session and Cookies*, we'll lock this down a bit more, but we also want to be able to demonstrate the editing of content through a RESTful API; so this will represent an incomplete stub for what will eventually be more secure and complete.

As with the creation of new comments, there is no security restriction in place here. Anyone can create a comment and anyone can edit it. It's the wild west of blog software, at least at this point.

First, we'll want to be able to see our submitted comments. To do so, we need to make minor modifications to our `Page struct` and the creation of a `Comment struct` to match our database structure:

```
type Comment struct {
  Id    int
  Name   string
  Email  string
  CommentText string
}

type Page struct {
  Id         int
  Title      string
  RawContent string
  Content    template.HTML
  Date       string
  Comments   []Comment
  Session    Session
  GUID       string
}
```

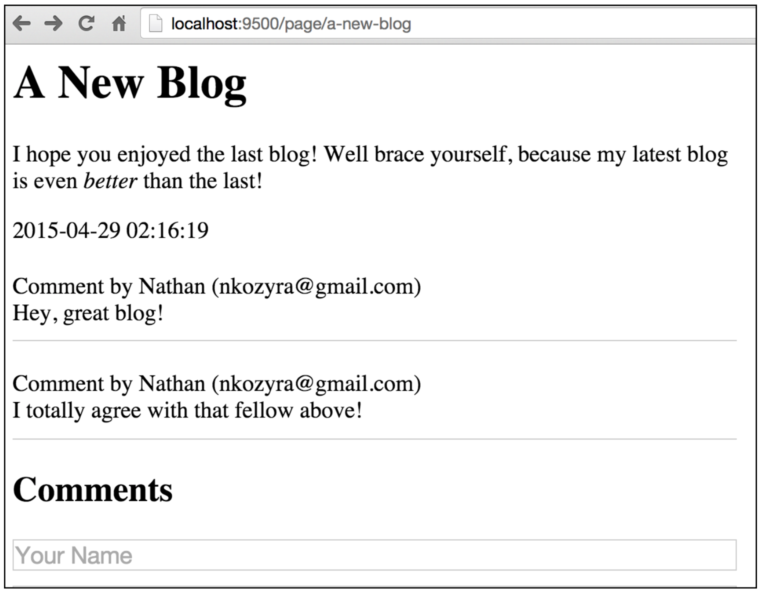
Since all the previously posted comments went into the database without any real fanfare, there was no record of the actual comments on the blog post page. To remedy that, we'll add a simple query of `Comments` and scan them using the `.Scan` method into an array of `Comment struct`.

First, we'll add the query to `ServePage`:

```
func ServePage(w http.ResponseWriter, r *http.Request) {
  vars := mux.Vars(r)
  pageGUID := vars["guid"]
  thisPage := Page{}
  fmt.Println(pageGUID)
  err := database.QueryRow("SELECT
id,page_title,page_content,page_date FROM pages WHERE
page_guid=?", pageGUID).Scan(&thisPage.Id, &thisPage.Title,
&thisPage.RawContent, &thisPage.Date)
```

```
  thisPage.Content = template.HTML(thisPage.RawContent)
  if err != nil {
    http.Error(w, http.StatusText(404), http.StatusNotFound)
    log.Println(err)
    return
  }

  comments, err := database.Query("SELECT id, comment_name as Name,
comment_email, comment_text FROM comments WHERE page_id=?", this
Page.Id)
  if err != nil {
    log.Println(err)
  }
  for comments.Next() {
    var comment Comment
    comments.Scan(&comment.Id, &comment.Name, &comment.Email,
&comment.CommentText)
    thisPage.Comments = append(thisPage.Comments, comment)
  }

  t, _ := template.ParseFiles("templates/blog.html")
  t.Execute(w, thisPage)
}
```

Now that we have `Comments` packed into our `Page struct`, we can display the **Comments** on the page itself:



A New Blog

I hope you enjoyed the last blog! Well brace yourself, because my latest blog is even *better* than the last!

2015-04-29 02:16:19

Comment by Nathan (nkozyra@gmail.com)
Hey, great blog!

Comment by Nathan (nkozyra@gmail.com)
I totally agree with that fellow above!

Comments

Since we're allowing anyone to edit, we'll have to create a form for each item, which will allow modifications. In general, HTML forms allow either `GET` or `POST` requests, so our hand is forced to utilize `XMLHttpRequest` to send this request. For the sake of brevity, we'll utilize jQuery and its `ajax()` method.

First, for our template's range for comments:

```
{{range .Comments}}
  <div class="comment">
    <div>Comment by {{.Name}} ({{.Email}})</div>
    {{.CommentText}}

    <div class="comment_edit">
    <h2>Edit</h2>
    <form onsubmit="return putComment(this);">
      <input type="hidden" class="edit_id" value="{{.Id}}" />
      <input type="text" name="name" class="edit_name" placehold
er="Your Name" value="{{.Name}}" />
     <input type="text" name="email" class="edit_email" placehold
er="Your Email" value="{{.Email}}" />
      <textarea class="edit_comments" name="comments">{{.
CommentText}}</textarea>
      <input type="submit" value="Edit" />
    </form>
    </div>
  </div>
{{end}}
```

And then for our JavaScript to process the form using `PUT`:

```
<script>
    function putComment(el) {
        var id = $(el).find('.edit_id');
        var name = $(el).find('.edit_name').val();
        var email = $(el).find('.edit_email').val();
        var text = $(el).find('.edit_comments').val();
        $.ajax({
            url: '/api/comments/' + id,
            type: 'PUT',
            succes: function(res) {
                alert('Comment Updated!');
            }
        });
        return false;
    }
</script>
```

To handle this call with the `PUT` verb, we'll need an update route and function. Lets add them now:

```
  routes.HandleFunc("/api/comments", APICommentPost).
    Methods("POST")
  routes.HandleFunc("/api/comments/{id:[\\w\\d\\-]+}", APICom
mentPut).
    Methods("PUT")
```

This enables a route, so now we just need to add the corresponding function, which will look fairly similar to our `POST`/`Create` method:

```
func APICommentPut(w http.ResponseWriter, r *http.Request) {
  err := r.ParseForm()
  if err != nil {
  log.Println(err.Error)
  }
  vars := mux.Vars(r)
  id := vars["id"]
  fmt.Println(id)
  name := r.FormValue("name")
  email := r.FormValue("email")
  comments := r.FormValue("comments")
  res, err := database.Exec("UPDATE comments SET comment_name=?,
comment_email=?, comment_text=? WHERE comment_id=?", name, email,
comments, id)
  fmt.Println(res)
  if err != nil {
    log.Println(err.Error)
  }

  var resp JSONResponse

  jsonResp, _ := json.Marshal(resp)
  w.Header().Set("Content-Type", "application/json")
  fmt.Fprintln(w, jsonResp)
}
```

In short, this takes our form and transforms it into an update to the data based on the comment's internal ID. As mentioned, it's not entirely different from our `POST` route method, and just like that method it doesn't return any data.

Summary

In this chapter, we've gone from exclusively server-generated HTML presentations to dynamic presentations that utilize an API. We've examined the basics of REST and implemented a RESTful interface for our blogging application.

While this can use a lot more client-side polish, we have `GET`/`POST`/`PUT` requests that are functional and allow us to create, retrieve, and update comments for our blog posts.

In *Chapter 6*, *Session and Cookies*, we'll examine user authentication, sessions, and cookies, and how we can take the building blocks we've laid in this chapter and apply some very important security parameters to it. We had an open-ended creation and updates of comments in this chapter; we'll restrict that to unique users in the next.

In doing all of this, we'll turn our proof-of-concept comment management into something that can be used in production practically.

6
Sessions and Cookies

Our application is beginning to get a little more real now; in the previous chapter, we added some APIs and client-side interfaces to them.

In our application's current state, we've added /api/comments, /api/comments/[id], /api/pages, and /api/pages/[id], thus making it possible for us to get and update our data in JSON format and making the application better suited for Ajax and client-side access.

Though we can now add comments and edit them directly through our API, there is absolutely no restriction on who can perform these actions. In this chapter, we'll look at the ways to limit access to certain assets, establishing identities, and securely authenticating when we have them.

By the end, we should be able to enable users to register and log in and utilize sessions, cookies, and flash messages to keep user state in our application in a secure way.

Setting cookies

The most common, fundamental, and simplest way to create persistent memory across a user's session is by utilizing cookies.

Cookies provide a way to share state information across requests, URL endpoints, and even domains, and they have been used (and abused) in every possible way.

Most often, they're used to keep a track of identity. When a user logs into a service, successive requests can access some aspects of the previous request (without duplicating a lookup or the login module) by utilizing the session information stored in a cookie.

If you're familiar with cookies in any other language's implementation, the basic `struct` will look familiar. Even so, the following relevant attributes are fairly lockstep with the way a cookie is presented to the client:

```
type Cookie struct {
  Name       string
  Value      string
  Path       string
  Domain     string
  Expires    time.Time
  RawExpires string
  MaxAge   int
  Secure   bool
  HttpOnly bool
  Raw      string
  Unparsed []string
}
```

That's a lot of attributes for a very basic `struct`, so let's focus on the important ones.

The `Name` attribute is simply a key for the cookie. The `Value` attribute represents its contents and `Expires` is a `Time` value for the moment when the cookie should be flushed by a browser or another headless recipient. This is all you need in order to set a valid cookie that lasts in Go.

Beyond the basics, you may find setting a `Path`, `Domain`, and `HttpOnly` useful, if you want to lock down the accessibility of the cookie.

Capturing user information

When a user with a valid session and/or cookie attempts to access restricted data, we need to get that from the user's browser.

A session itself is just that—a single session on the site. It doesn't naturally persist indefinitely, so we need to leave a breadcrumb, but we also want to leave one that's relatively secure.

For example, we would never want to leave critical user information in the cookie, such as name, address, email, and so on.

However, any time we have some identifying information, we leave some vector for misdeed—in this case we'll likely leave a session identifier that represents our session ID. The vector in this case allows someone, who obtains this cookie, to log in as one of our users and change information, find billing details, and so on.

These types of physical attack vectors are well outside the scope of this (and most) application and to a large degree, it's a concession that if someone loses access to their physical machine, they can also have their account compromised.

What we want to do here is ensure that we're not transmitting personal or sensitive information over clear text or without a secure connection. We'll cover setting up TLS in *Chapter 9, Security*, so here we want to focus on limiting the amount of information we store in our cookies.

Creating users

In the previous chapter, we allowed non-authorized requests to create new comments by hitting our REST API via a `POST`. Anyone who's been on the Internet for a while knows a few truisms, such as:

1. The comments section is often the most toxic part of any blog or news post
2. Step 1 is true, even when users have to authenticate in non-anonymous ways

Now, let's lock down the comments section to ensure that users have registered themselves and are logged in.

We won't go deep into the authentication's security aspects now, as we'll be going deeper with that in *Chapter 9, Security*.

First, let's add a `users` table in our database:

```
CREATE TABLE `users` (
  `id` int(11) unsigned NOT NULL AUTO_INCREMENT,
  `user_name` varchar(32) NOT NULL DEFAULT '',
  `user_guid` varchar(256) NOT NULL DEFAULT '',
  `user_email` varchar(128) NOT NULL DEFAULT '',
  `user_password` varchar(128) NOT NULL DEFAULT '',
  `user_salt` varchar(128) NOT NULL DEFAULT '',
  `user_joined_timestamp` timestamp NULL DEFAULT NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1;
```

We could surely go a lot deeper with user information, but this is enough to get us started. As mentioned, we won't go too deep into security, so we'll just generate a hash for the password now and not worry about the salt.

Finally, to enable sessions and users in the app, we'll make some changes to our structs:

```
type Page struct {
  Id         int
  Title      string
  RawContent string
  Content    template.HTML
  Date       string
  Comments   []Comment
  Session    Session
}

type User struct {
  Id   int
  Name string
}

type Session struct {
  Id              string
  Authenticated   bool
  Unauthenticated bool
  User            User
}
```

And here are the two stub handlers for registration and logging in. Again, we're not putting our full effort into fleshing these out into something robust, we just want to open the door a bit.

Enabling sessions

In addition to storing the users themselves, we'll also want some way of persistent memory for accessing our cookie data. In other words, when a user's browser session ends and they come back, we'll validate and reconcile their cookie value against values in our database.

Use this SQL to create the `sessions` table:

```
CREATE TABLE `sessions` (
  `id` int(11) unsigned NOT NULL AUTO_INCREMENT,
  `session_id` varchar(256) NOT NULL DEFAULT '',
  `user_id` int(11) DEFAULT NULL,
  `session_start` timestamp NOT NULL DEFAULT CURRENT_TIMESTAMP ON
UPDATE CURRENT_TIMESTAMP,
  `session_update` timestamp NOT NULL DEFAULT '0000-00-00
00:00:00',
  `session_active` tinyint(1) NOT NULL,
  PRIMARY KEY (`id`),
  UNIQUE KEY `session_id` (`session_id`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1;
```

The most important values are the `user_id`, `session_id`, and the timestamps for updating and starting. We can use the latter two to decide if a session is actually valid after a certain period. This is a good security practice, just because a user has a valid cookie doesn't necessarily mean that they should remain authenticated, particularly if you're not using a secure connection.

Letting users register

To be able to allow users to create accounts themselves, we'll need a form for both registering and logging in. Now, most systems similar to this do some multi-factor authentication to allow a user backup system for retrieval as well as validation that the user is real and unique. We'll get there, but for now let's keep it as simple as possible.

We'll set up the following endpoints to allow a user to `POST` both the register and login forms:

```
routes.HandleFunc("/register", RegisterPOST).
  Methods("POST").
  Schemes("https")
routes.HandleFunc("/login", LoginPOST).
  Methods("POST").
  Schemes("https")
```

Keep in mind that these are presently set to the HTTPS scheme. If you're not using that, remove that part of the `HandleFunc` register.

Since we're only showing these following views to unauthenticated users, we can put them on our `blog.html` template and wrap them in `{{if .Session.Unauthenticated}} … {{end}}` template snippets. We defined `.Unauthenticated` and `.Authenticated` in the application under the `Session struct`, as shown in the following example:

```
{{if .Session.Unauthenticated}}<form action="/register"
method="POST">
  <div><input type="text" name="user_name" placeholder="User name"
/></div>
  <div><input type="email" name="user_email" placeholder="Your
email" /></div>
  <div><input type="password" name="user_password"
placeholder="Password" /></div>
  <div><input type="password" name="user_password2"
placeholder="Password (repeat)" /></div>
  <div><input type="submit" value="Register" /></div>
</form>{{end}}
```

And our `/register` endpoint:

```
func RegisterPOST(w http.ResponseWriter, r *http.Request) {
  err := r.ParseForm()
  if err != nil {
    log.Fatal(err.Error)
  }
  name := r.FormValue("user_name")
  email := r.FormValue("user_email")
  pass := r.FormValue("user_password")
  pageGUID := r.FormValue("referrer")
  // pass2 := r.FormValue("user_password2")
  gure := regexp.MustCompile("[^A-Za-z0-9]+")
  guid := gure.ReplaceAllString(name, "")
  password := weakPasswordHash(pass)

  res, err := database.Exec("INSERT INTO users SET user_name=?,
user_guid=?, user_email=?, user_password=?", name, guid, email,
password)
  fmt.Println(res)
  if err != nil {
    fmt.Fprintln(w, err.Error)
  } else {
    http.Redirect(w, r, "/page/"+pageGUID, 301)
  }
}
```

Note that this fails inelegantly for a number of reasons. If the passwords do not match, we don't check and report to the user. If the user already exists, we don't tell them the reason for a registration failure. We'll get to that, but now our main intent is producing a session.

For reference, here's our `weakPasswordHash` function, which is only intended to generate a hash for testing:

```
func weakPasswordHash(password string) []byte {
  hash := sha1.New()
  io.WriteString(hash, password)
  return hash.Sum(nil)
}
```

Letting users log in

A user may be already registered; in which case, we'll also want to provide a login mechanism on the same page. This can obviously be subject to better design considerations, but we just want to make them both available:

```
<form action="/login" method="POST">
  <div><input type="text" name="user_name" placeholder="User name"
/></div>
  <div><input type="password" name="user_password"
placeholder="Password" /></div>
  <div><input type="submit" value="Log in" /></div>
</form>
```

And then we'll need receiving endpoints for each POSTed form. We're not going to do a lot of validation here either, but we're not in a position to validate a session.

Initiating a server-side session

One of the most common ways of authenticating a user and saving their state on the Web is through sessions. You may recall that we mentioned in the last chapter that REST is stateless, the primary reason for that is because HTTP itself is stateless.

If you think about it, to establish a consistent state with HTTP, you need to include a cookie or a URL parameter or something that is not built into the protocol itself.

Sessions are created with unique identifiers that are usually not entirely random but unique enough to avoid conflicts for most logical and plausible scenarios. This is not absolute, of course, and there are plenty of (historical) examples of session token hijacking that are not related to sniffing.

Session support as a standalone process does not exist in Go core. Given that we have a storage system on the server side, this is somewhat irrelevant. If we create a safe process for generation of server keys, we can store them in secure cookies.

But generating session tokens is not completely trivial. We can do this using a set of available cryptographic methods, but with session hijacking as a very prevalent way of getting into systems without authorization, that may be a point of insecurity in our application.

Since we're already using the Gorilla toolkit, the good news is that we don't have to reinvent the wheel, there's a robust session system in place.

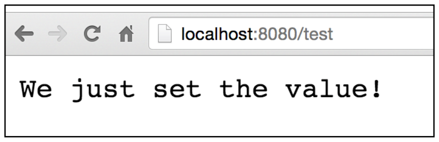
Not only do we have access to a server-side session, but we get a very convenient tool for one-time messages within a session. These work somewhat similar to a message queue in the manner that once data goes into them, the flash message is no longer valid when that data is retrieved.

Creating a store

To utilize the Gorilla sessions, we first need to invoke a cookie store, which will hold all the variables that we want to keep associated with a user. You can test this out pretty easily by the following code:

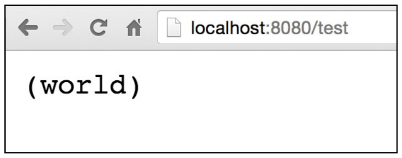
```
package main

import (
  "fmt"
  "github.com/gorilla/sessions"
  "log"
  "net/http"
)

func cookieHandler(w http.ResponseWriter, r *http.Request) {
  var cookieStore = sessions.NewCookieStore([]byte("ideally, some
random piece of entropy"))
  session, _ := cookieStore.Get(r, "mystore")
  if value, exists := session.Values["hello"]; exists {
    fmt.Fprintln(w, value)
  } else {
    session.Values["hello"] = "(world)"
    session.Save(r, w)
    fmt.Fprintln(w, "We just set the value!")
  }
}

func main() {
  http.HandleFunc("/test", cookieHandler)
  log.Fatal(http.ListenAndServe(":8080", nil))
}
```

The first time you hit your URL and endpoint, you'll see **We just set the value!**, as shown in the following screenshot:



```
We just set the value!
```

In the second request, you should see **(world)**, as shown in the following screenshot:



```
(world)
```

A couple of notes here. First, you must set cookies before sending anything else through your `io.Writer` (in this case the `ResponseWriter w`). If you flip these lines:

```
        session.Save(r, w)
        fmt.Fprintln(w, "We just set the value!")
```

You can see this in action. You'll never get the value set to your cookie store.

So now, let's apply it to our application. We will want to initiate a session store before any requests to `/login` or `/register`.

We'll initialize a global `sessionStore`:

```
var database *sql.DB
var sessionStore = sessions.NewCookieStore([]byte("our-social-
network-application"))
```

Feel free to group these, as well, in a `var ()`. Next, we'll want to create four simple functions that will get an active session, update a current one, generate a session ID, and evaluate an existing cookie. These will allow us to check if a user is logged in by a cookie's session ID and enable persistent logins.

First, the `getSessionUID` function, which will return a user's ID if a session already exists:

```
func getSessionUID(sid string) int {
  user := User{}
  err := database.QueryRow("SELECT user_id FROM sessions WHERE
session_id=?", sid).Scan(user.Id)
  if err != nil {
    fmt.Println(err.Error)
    return 0
  }
  return user.Id
}
```

Next, the update function, which will be called with every front-facing request, thus enabling a timestamp update or inclusion of a user ID if a new log in is attempted:

```
func updateSession(sid string, uid int) {
  const timeFmt = "2006-01-02T15:04:05.999999999"
  tstamp := time.Now().Format(timeFmt)
  _, err := database.Exec("INSERT INTO sessions SET session_id=?,
user_id=?, session_update=? ON DUPLICATE KEY UPDATE user_id=?,
session_update=?", sid, uid, tstamp, uid, tstamp)
  if err != nil {
    fmt.Println(err.Error)
  }
}
```

An important part is the ability to generate a strongly-random byte array (cast to string) that will allow unique identifiers. We do that with the following `generateSessionId()` function:

```
func generateSessionId() string {
  sid := make([]byte, 24)
  _, err := io.ReadFull(rand.Reader, sid)
  if err != nil {
    log.Fatal("Could not generate session id")
  }
  return base64.URLEncoding.EncodeToString(sid)
}
```

And finally, we have the function that will be called with every request to check for a cookie's session or create one if it doesn't exist.

```
func validateSession(w http.ResponseWriter, r *http.Request) {
  session, _ := sessionStore.Get(r, "app-session")
  if sid, valid := session.Values["sid"]; valid {
    currentUID := getSessionUID(sid.(string))
    updateSession(sid.(string), currentUID)
    UserSession.Id = string(currentUID)
  } else {
    newSID := generateSessionId()
    session.Values["sid"] = newSID
    session.Save(r, w)
    UserSession.Id = newSID
    updateSession(newSID, 0)
  }
  fmt.Println(session.ID)
}
```

This is predicated on having a global `Session struct`, in this case defined as:

```
var UserSession Session
```

This leaves us with just one piece—to call `validateSession()` on our `ServePage()` method and `LoginPost()` method and then validate the passwords on the latter and update our session on a successful login attempt:

```
func LoginPOST(w http.ResponseWriter, r *http.Request) {
  validateSession(w, r)
```

In our previously defined check against the form values, if a valid user is found, we'll update the session directly:

```
  u := User{}
  name := r.FormValue("user_name")
  pass := r.FormValue("user_password")
  password := weakPasswordHash(pass)
  err := database.QueryRow("SELECT user_id, user_name FROM users
WHERE user_name=? and user_password=?", name,
password).Scan(&u.Id, &u.Name)
  if err != nil {
    fmt.Fprintln(w, err.Error)
    u.Id = 0
    u.Name = ""
  } else {
    updateSession(UserSession.Id, u.Id)
    fmt.Fprintln(w, u.Name)
  }
```

Utilizing flash messages

As mentioned earlier in this chapter, Gorilla sessions offer a simple system to utilize a single-use and cookie-based data transfer between requests.

The idea behind a flash message is not all that different than an in-browser/server message queue. It's most frequently utilized in a process such as this:

- A form is POSTed
- The data is processed
- A header redirect is initiated
- The resulting page needs some access to information about the `POST` process (success, error)

At the end of this process, the message should be removed so that the message is not duplicated erroneously at some other point. Gorilla makes this incredibly easy, and we'll look at that shortly, but it makes sense to show a quick example of how this can be accomplished in native Go.

To start, we'll create a simple HTTP server that includes a starting point handler called `startHandler`:

```
package main

import (
  "fmt"
  "html/template"
  "log"
  "net/http"
  "time"
)

var (
  templates = template.Must(template.ParseGlob("templates/*"))
  port      = ":8080"
)

func startHandler(w http.ResponseWriter, r *http.Request) {
  err := templates.ExecuteTemplate(w, "ch6-flash.html", nil)
  if err != nil {
    log.Fatal("Template ch6-flash missing")
  }
}
```

We're not doing anything special here, just rendering our form:

```
func middleHandler(w http.ResponseWriter, r *http.Request) {
  cookieValue := r.PostFormValue("message")
  cookie := http.Cookie{Name: "message", Value: "message:" +
cookieValue, Expires: time.Now().Add(60 * time.Second), HttpOnly:
true}
  http.SetCookie(w, &cookie)
  http.Redirect(w, r, "/finish", 301)
}
```

Our `middleHandler` demonstrates creating cookies through a `Cookie struct`, as described earlier in this chapter. There's nothing important to note here except the fact that you may want to extend the expiration out a bit, just to ensure that there's no way a cookie could expire (naturally) between requests:

```
func finishHandler(w http.ResponseWriter, r *http.Request) {
  cookieVal, _ := r.Cookie("message")

  if cookieVal != nil {
    fmt.Fprintln(w, "We found: "+string(cookieVal.Value)+", but
try to refresh!")
    cookie := http.Cookie{Name: "message", Value: "", Expires:
time.Now(), HttpOnly: true}
    http.SetCookie(w, &cookie)
  } else {
    fmt.Fprintln(w, "That cookie was gone in a flash")
  }

}
```

The `finishHandler` function does the magic of a flash message—removes the cookie if and only if a value has been found. This ensures that the cookie is a one-time retrievable value:

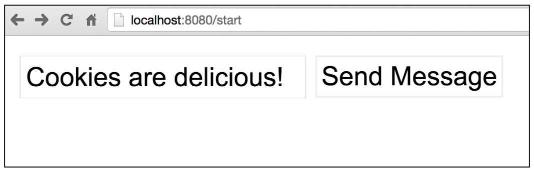
```
func main() {

  http.HandleFunc("/start", startHandler)
  http.HandleFunc("/middle", middleHandler)
  http.HandleFunc("/finish", finishHandler)
  log.Fatal(http.ListenAndServe(port, nil))

}
```

The following example is our HTML for POSTing our cookie value to the `/middle` handler:

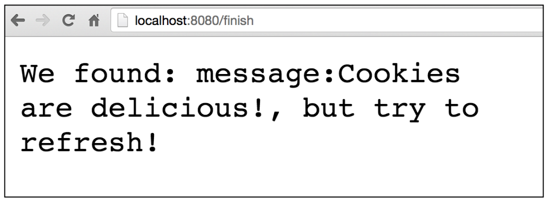
```
<html>
<head><title>Flash Message</title></head>
<body>
<form action="/middle" method="POST">
  <input type="text" name="message" />
  <input type="submit" value="Send Message" />
</form>
</body>
</html>
```

If you do as the page suggests and refresh again, the cookie value would have been removed and the page will not render, as you've previously seen.

To begin the flash message, we hit our `/start` endpoint and enter an intended value and then click on the **Send Message** button:

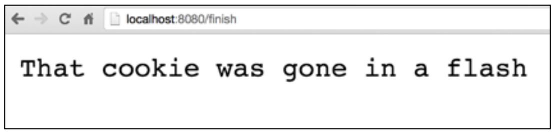


At this point, we'll be sent to the `/middle` endpoint, which will set the cookie value and HTTP redirect to `/finish`:



```
We found: message:Cookies
are delicious!, but try to
refresh!
```

And now we can see our value. Since the `/finish` endpoint handler also unsets the cookie, we'll be unable to retrieve that value again. Here's what happens if we do what `/finish` tells us on its first appearance:



```
That cookie was gone in a flash
```

That's all for now.

Summary

Hopefully by this point you have a grasp of how to utilize basic cookies and sessions in Go, either through native Go or through the use of a framework, such as Gorilla. We've tried to demonstrate the inner workings of the latter so you're able to build without additional libraries obfuscating the functionality.

We've implemented sessions into our application to enable persistent state between requests. This is the very basis of authentication for the Web. By enabling `users` and `sessions` table in our database, we're able to log users in, register a session, and associate that session with the proper user on subsequent requests.

By utilizing flash messages, we made use of a very specific feature that allows transfer of information between two endpoints without enabling an additional request that may look like an error to the user or generate erroneous output. Our flash messages works just once and then expire.

In *Chapter 7, Microservices and Communication*, we'll look at connecting disparate systems and applications across our existing and new APIs to allow event-based actions to be coordinated between those systems. This will facilitate connecting to other services within the same environment as well as those outside of our application.

7
Microservices and Communication

Our application is beginning to get a little more real now. In the previous chapter, we added some APIs and client-side interfaces to them.

Microservices have become very hot in the last few years, primarily because they reduce the developmental and support weight of a very large or monolithic application. By breaking apart these monoliths, microservices enable a more agile and concurrent development. They can allow separate teams to work on separate parts of the application without worrying too much about conflicts, backwards compatibility issues, or stepping on the toes of other parts of the application.

In this chapter, we'll introduce microservices and explore how Go can work within them, to enable them and even drive their central mechanisms.

To sum this all up, we will be covering the following aspects:

- Introducing the microservice approach
- Pros and cons of utilizing microservices
- Understanding the heart of microservices
- Communicating between microservices
- Putting a message on the wire
- Reading from another service

Introducing the microservice approach

If you've not yet encountered the term microservice or explored its meaning in depth, we can very quickly demystify it. Microservices are, in essence, independent functions of an overall application being broken apart and made accessible via some universally known protocol.

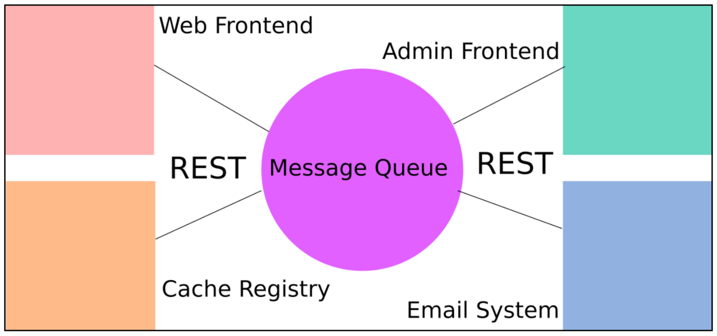
The microservice approach is, usually, utilized to break apart a very large monolithic application.

Imagine your standard web application in the mid-2000s. When new functionality is needed, let's say a function that emails new users, it's added directly to the codebase and integrated with the rest of the application.

As the application grows, so does the necessary test coverage. So, it increases the potential for critical errors too. In this scenario, a critical error doesn't just bring down that component, in this case the e-mailing system; it takes down the entire application.

This can be a nightmare to track down, patch, and re-deploy, and it's exactly the type of nightmare that microservices were designed to address.

If the e-mailing part of the application is separated into its own app, it has a level of isolation and insulation that makes finding problems much easier. It also means that the entire stack doesn't fall down just because someone introduced a critical error into one small part of the whole app, as shown in the following figure:



Consider the following basic example architecture, where an application is split into four separate concepts, which represent their own applications in the microservices framework.

Once, every single piece existed in its own application; now they are broken apart into smaller and more manageable systems. Communication between the applications happens via a message queue utilizing REST API endpoints.

Pros and cons of utilizing microservices

If microservices seem like a panacea at this point, we should also note that this approach does not come without its own set of issues. Whether the tradeoff is worth it or not depends heavily on an overall organizational approach.

As mentioned earlier, stability and error detection comprise a big production-level win for microservices. But if you think of the flip side of applications not breaking, it could also mean that issues go hidden for longer than they otherwise would. It's hard to ignore the entire site being down, but it could be hours before anyone realizes that e-mails have not been sent, unless some very robust logging is in place.

But there are other big pros to microservices. For one, utilizing an external standard communication protocol (REST, for example) means that you're not locked into a single language.

If, for example, some part of your application can be written better in Node than in Go, you can do that without having to rewrite an entire application. This is a frequent temptation for developers: rewriting the whole thing because the new and shiny language app or feature is introduced. Well, microservices safely enable this behavior—it allows a developer or a group of developers to try something without needing to go deeper than the specific function they wish to write.

This, too, comes with a potentially negative scenario—since the application components are decoupled, so can the institutional knowledge around them be decoupled. Few developers may know enough to keep the service operating ideally. Other members of the group may lack the language knowledge to jump in and fix critical errors.

One final, but important, consideration is that microservice architecture generally means a distributed environment by default. This leads us to the biggest immediate caveat, which is the fact that this situation almost always means that eventual consistency is the name of the game.

Since every message must depend on multiple external services, you're subject to several layers of latency to get a change enacted.

Understanding the heart of microservices

You might be wondering about one thing as you consider this system to design dissonant services that work in congress: what's the communication platform? To answer this, we'll say there is an easy answer and a more intricate one.

The easy answer is REST. This is great news, as you're likely to be well versed in REST or you at least understand some portion of it from *Chapter 5, Frontend Integration with RESTful APIs*. There we described the basics of API communication utilizing RESTful, stateless protocols and implementing HTTP verbs as actions.

Which leads us to the more complex answer: not everything in a large or involved application can operate on REST alone. Some things require state or at least some level of long-lasting consistency.

For the latter problem, most microservice architectures are centered on a message queue as a platform for information sharing and dissemination. A message queue serves as a conduit to receive REST requests from one service and holds it until another service retrieves the request for further processing.

Communicating between microservices

There are a number of approaches to communicate between microservices, as mentioned; REST endpoints provide a nice landing pad for messages. You may recall the preceding graphic, which shows a message queue as the central conduit between services. This is one of the most common ways to handle message passing and we'll use RabbitMQ to demonstrate this.

In this case, we'll show when new users register to an e-mail queue for the delivery of a message in our RabbitMQ installation, which will then be picked up by an emailing microservice.

> You can read more about RabbitMQ, which utilizes **Advanced Message Queuing Protocol** (**AMQP**) here: `https://www.rabbitmq.com/`.
>
> To install an AMQP client for Go, we'll recommend Sean Treadway's AMQP package. You can install it with a `go get` command. You can get it at `github.com/streadway/amqp`

Putting a message on the wire

There are a lot of approaches to use RabbitMQ. For example, one allows multiple workers to accomplish the same thing, as a method for distributing works among available resources.

Assuredly, as a system grows, it is likely to find use for that method. But in our tiny example, we want to segregate tasks based on a specific channel. Of course, this is not analogous to Go's concurrency channels, so keep that in mind when you read about this approach.

But to explain this method, we may have separate exchanges to route our messages. In our example, we might have a log queue where messages are aggregated from all services into a single log location, or a cache expiration method that removes cached items from memory when they're deleted from the database.

In this example, though, we'll implement an e-mail queue that can take a message from any other service and use its contents to send an e-mail. This keeps all e-mail functionality outside of core and supportive services.

Recall that in *Chapter 5*, *Frontend Integration with RESTful APIs*, we added register and login methods. The one we're most interested in here is `RegisterPOST()`, where we allowed users to register to our site and then comment on our posts.

It's not uncommon for newly registered users to get an e-mail, either for confirmation of identity or for a simple welcome message. We'll do the latter here, but adding confirmation is trivial; it's just a matter of producing a key, delivering via e-mail and then enabling the user once the link is hit.

Since we're using an external package, the first thing we need to do is import it.

Here's how we do it:

```
import (
  "bufio"
  "crypto/rand"
  "crypto/sha1"
  "database/sql"
  "encoding/base64"
  "encoding/json"
  "fmt"
  _ "github.com/go-sql-driver/mysql"
  "github.com/gorilla/mux"
  "github.com/gorilla/sessions"
  "github.com/streadway/amqp"
  "html/template"
  "io"
  "log"
  "net/http"
  "regexp"
  "text/template"
  "time"
)
```

Note that here we've included `text/template`, which is not strictly necessary since we have `html/template`, but we've noted here in case you wish to use it in a separate process. We have also included `bufio`, which we'll use as part of the same templating process.

For the sake of sending an e-mail, it will be helpful to have a message and a title for the e-mail, so let's declare these. In a production environment, we'd probably have a separate language file, but we don't have much else to show at this point:

```
var WelcomeTitle = "You've successfully registered!"
var WelcomeEmail = "Welcome to our CMS, {{Email}}!  We're glad you
could join us."
```

These simply represent the e-mail variables we need to utilize when we have a successful registration.

Since we're putting a message on the wire and yielding some responsibility for the application's logic to another service, for now we'll only need to ensure that our message has been received by RabbitMQ.

Next, we'll need to connect to the queue, which we can pass either by reference or reconnect with each message. Generally, you'll want to keep the connection in the queue for a long time, but you may choose to reconnect and close your connection each time while testing.

In order to do so, we'll add our MQ host information to our constants:

```
const (
  DBHost  = "127.0.0.1"
  DBPort  = ":3306"
  DBUser  = "root"
  DBPass  = ""
  DBDbase = "cms"
  PORT    = ":8080"
  MQHost  = "127.0.0.1"
  MQPort  = ":5672"
)
```

When we create a connection, we'll use the somewhat familiar `TCP Dial()` method, which returns an MQ connection. Here is our function for connecting:

```
func MQConnect() (*amqp.Connection, *amqp.Channel, error) {
  url := "amqp://" + MQHost + MQPort
  conn, err := amqp.Dial(url)
  if err != nil {
    return nil, nil, err
  }
  channel, err := conn.Channel()
  if err != nil {
    return nil, nil, err
  }
  if _, err := channel.QueueDeclare("", false, true, false, false,
nil); err != nil {
    return nil, nil, err
  }
  return conn, channel, nil
}
```

We can choose to pass the connection by reference or sustain it as a global with all applicable caveats considered here.

> You can read a bit more about RabbitMQ connections and detecting disrupted connections at `https://www.rabbitmq.com/heartbeats.html`

Technically, any producer (in this case our application) doesn't push messages to the queue; rather, it pushes them to the exchange. RabbitMQ allows you to find exchanges with the `rabbitmqctl list_exchanges` command (rather than `list_queues`). Here, we're using an empty exchange, which is totally valid. The distinction between a queue and an exchange is non-trivial; the latter is responsible for having defined the rules surrounding a message, en route to a queue or queues.

Inside our `RegisterPOST()`, let's send a JSON-encoded message when a successful registration takes place. We'll want a very simple `struct` to maintain the data we'll need:

```
type RegistrationData struct {
  Email   string `json:"email"`
  Message string `json:"message"`
}
```

Now we'll create a new `RegistrationData struct` if, and only if, the registration process succeeds:

```
  res, err := database.Exec("INSERT INTO users SET user_name=?,
user_guid=?, user_email=?, user_password=?", name, guid, email,
password)

  if err != nil {
    fmt.Fprintln(w, err.Error)
  } else {
    Email := RegistrationData{Email: email, Message: ""}
    message, err := template.New("email").Parse(WelcomeEmail)
    var mbuf bytes.Buffer
    message.Execute(&mbuf, Email)
    MQPublish(json.Marshal(mbuf.String()))
    http.Redirect(w, r, "/page/"+pageGUID, 301)
  }
```

And finally, we'll need the function that actually sends our data, `MQPublish()`:

```
func MQPublish(message []byte) {
  err = channel.Publish(
    "email", // exchange
    "",      // routing key
    false,   // mandatory
    false,   // immediate
    amqp.Publishing{
      ContentType: "text/plain",
      Body:        []byte(message),
    })
}
```

Reading from another service

Now that we've sent a message to our message queue in our app, let's use another microservice to pluck that from the queue on the other end.

To demonstrate the flexibility of a microservice design, our secondary service will be a Python script that connects to the MQ and listens for messages on the e-mail queue, when it finds one. It will parse the message and send an e-mail. Optionally, it could publish a status message back on the queue or log it, but we won't go down that road for now:

```
import pika
import json
import smtplib
from email.mime.text import MIMEText

connection = pika.BlockingConnection(pika.ConnectionParameters(
host='localhost'))
channel = connection.channel()
channel.queue_declare(queue='email')

print ' [*] Waiting for messages. To exit press CTRL+C'

def callback(ch, method, properties, body):
    print " [x] Received %r" % (body,)
    parsed = json.loads(body)
    msg = MIMEText()
    msg['From'] = 'Me'
```

```
        msg['To'] = parsed['email']
        msg['Subject'] = parsed['message']
        s = smtplib.SMTP('localhost')
        s.sendmail('Me', parsed['email'], msg.as_string())
        s.quit()

    channel.basic_consume(callback,
                          queue='email',
                          no_ack=True)

    channel.start_consuming()
```

Summary

In this chapter, we looked at experimenting with utilizing microservices as a way to dissect your app into separate domains of responsibility. In this example, we delegated the e-mail aspect of our application to another service written in Python.

We did this to utilize the concept of microservices or interconnected smaller applications as callable networked functions. This ethos is driving a large part of the Web of late and has myriad benefits and drawbacks.

In doing this, we implemented a message queue, which operates as the backbone of our communications system, allowing each component to speak to the other in a reliable and repeatable way. In this case, we used a Python application to read messages sent from our Go application across RabbitMQ and take that e-mail data and process it.

In *Chapter 8, Logging and Testing*, we'll focus on logging and testing, which we can use to extend the microservices concept so that we can recover from errors and understand where things might go awry in the process.

8
Logging and Testing

In the previous chapter, we discussed delegating application responsibility to networked services accessible by API and intra-process communication and handled by a message queue.

This approach mimics an emerging trend of breaking large monolithic applications into smaller chunks; thus, allowing developers to leverage dissonant languages, frameworks, and designs.

We listed a few upsides and downsides of this approach; while most of the upsides dealt with keeping the development agile and lean while preventing catastrophic and cascading errors that might bring down an entire application, a large downside is the fragility of each individual component. For example, if our e-mail microservice had bad code as a part of a large application, the error would make itself known quickly because it would almost assuredly have a direct and detectable impact on another component. But by isolating processes as part of microservices, we also isolate their state and status.

This is where the contents of this chapter come into play—the ability to test and log within a Go application is the strength of the language's design. By utilizing these in our application, it grows to include more microservices; due to which we can be in a better position to keep track of any issues with a cog in the system without imposing too much additional complexity to the overall application.

In this chapter we will cover the following topics:

- Introducing logging in Go
- Logging to IO
- Formatting your output
- Using panics and fatal errors
- Introducing testing in Go

Introducing logging in Go

Go comes with innumerable ways to display output to `stdout`, most commonly the `fmt` package's `Print` and `Println`. In fact, you can eschew the `fmt` package entirely and just use `print()` or `println()`.

In mature applications, you're unlikely to see too many of these, because simply displaying an output without having the capability to store that somewhere for debugging or later analysis is rare and lacks much utility. Even if you're just outputting minor feedback to a user, it often makes sense to do so and keep the ability to save that to a file or elsewhere, this is where the `log` package comes into play. Most of the examples in this book have used `log.Println` in lieu of `fmt.Println` for this very reason. It's trivial to make that change if, at some point, you choose to supplant `stdout` with some other (or additional) `io.Writer`.

Logging to IO

So far we've been logging in to `stdout`, but you can utilize any `io.Writer` to ingest the log data. In fact, you can use multiple `io.Writers` if you want the output to be routed to more than one place.

Multiple loggers

Most mature applications will write to more than one log file to delineate between the various types of messages that need to be retained.

The most common use case for this is found in web server. They typically keep an `access.log` and an `error.log` file to allow the analysis of all successful requests; however, they also maintain separate logging of different types of messages.

In the following example, we modify our logging concept to include errors as well as warnings:

```
package main

import (
  "log"
  "os"
)
var (
  Warn   *log.Logger
  Error  *log.Logger
  Notice *log.Logger
)
```

```
func main() {
  warnFile, err := os.OpenFile("warnings.log",
os.O_RDWR|os.O_APPEND, 0660)
  defer warnFile.Close()
  if err != nil {
    log.Fatal(err)
  }
  errorFile, err := os.OpenFile("error.log",
os.O_RDWR|os.O_APPEND, 0660)
  defer errorFile.Close()
  if err != nil {
    log.Fatal(err)
  }

  Warn = log.New(warnFile, "WARNING: ", Log.LstdFlags
)

  Warn.Println("Messages written to a file called 'warnings.log'
are likely to be ignored :(")

  Error = log.New(errorFile, "ERROR: ", log.Ldate|log.Ltime)
  Error.SetOutput(errorFile)
  Error.Println("Error messages, on the other hand, tend to catch
attention!")
}
```

We can take this approach to store all sorts of information. For example, if we wanted to store registration errors, we can create a specific registration error logger and allow a similar approach if we encounter an error in that process as shown:

```
  res, err := database.Exec("INSERT INTO users SET user_name=?,
user_guid=?, user_email=?, user_password=?", name, guid, email,
passwordEnc)

  if err != nil {
    fmt.Fprintln(w, err.Error)
    RegError.Println("Could not complete registration:",
err.Error)
  } else {
    http.Redirect(w, r, "/page/"+pageGUID, 301)
  }
```

Formatting your output

When instantiating a new `Logger`, you can pass a few useful parameters and/or helper strings to help define and clarify the output. Each log entry can be prepended with a string, which can be helpful while reviewing multiple types of log entries. You can also define the type of date and time formatting that you would like on each entry.

To create a custom formatted log, just invoke the `New()` function with an `io.Writer` as shown:

```
package main

import (
  "log"
  "os"
)

var (
  Warn   *log.Logger
  Error  *log.Logger
  Notice *log.Logger
)

func main() {
  warnFile, err := os.OpenFile("warnings.log",
os.O_RDWR|os.O_APPEND, 0660)
  defer warnFile.Close()
  if err != nil {
    log.Fatal(err)
  }
  Warn = log.New(warnFile, "WARNING: ", log.Ldate|log.Ltime)

  Warn.Println("Messages written to a file called 'warnings.log'
are likely to be ignored :(")
  log.Println("Done!")
}
```

This not only allows us to utilize `stdout` with our `log.Println` function but also store more significant messages in a log file called `warnings.log`. Using the `os.O_RDWR|os.O_APPEND` constants allow us to write to the file and use an append file mode, which is useful for logging.

Using panics and fatal errors

In addition to simply storing messages from your applications, you can create application panics and fatal errors that will prevent the application from continuing. This is critical for any use case where errors that do not halt execution lead to potential security issues, data loss, or any other unintended consequence. These types of mechanisms are generally relegated to the most critical of errors.

When to use a `panic()` method is not always clear, but in practice this should be relegated to errors that are unrecoverable. An unrecoverable error typically means the one where state becomes ambiguous or cannot otherwise be guaranteed.

For example, operations on acquired database records that fail to return expected results from the database may be considered unrecoverable because future operations might occur on outdated or missing data.

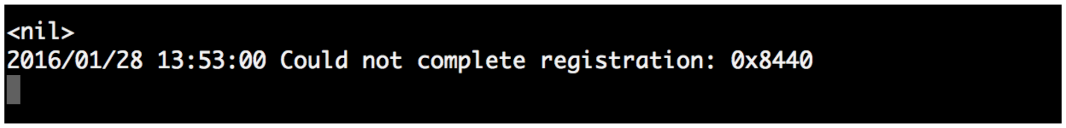
In the following example, we can implement a panic where we can't create a new user; this is important so that we don't attempt to redirect or move forward with any further creation steps:

```
  if err != nil {
    fmt.Fprintln(w, err.Error)
    RegError.Println("Could not complete registration:",
err.Error)
    panic("Error with registration,")
  } else {
    http.Redirect(w, r, "/page/"+pageGUID, 301)
  }
```

Note that if you want to force this error, you can just make an intentional MySQL error in your query:

```
  res, err := database.Exec("INSERT INTENTIONAL_ERROR INTO users
SET user_name=?, user_guid=?, user_email=?, user_password=?",
name, guid, email, passwordEnc)
```

When this error is triggered you will find this in your respective log file or `stdout`:



```
<nil>
2016/01/28 13:53:00 Could not complete registration: 0x8440
```

In the preceding example, we utilize the panic as a hard stop, one that will prevent further execution that could lead to further errors and/or data inconsistency. If it need not be a hard stop, utilizing the `recover()` function allows you to re-enter application flow once the problem has been addressed or mitigated.

Introducing testing in Go

Go comes packaged with a great deal of wonderful tools for making sure your code is clean, well-formatted, free of race conditions, and so on. From `go vet` to `go fmt`, many of the helper applications that you need to install separately in other languages come as a package with Go.

Testing is a critical step for software-development. Unit testing and test-driven development helps find bugs that aren't immediately apparent, especially to the developer. Often we're too close and too familiar with the application to make the types of usability mistakes that can invoke the otherwise undiscovered errors.

Go's testing package allows unit testing of actual functionality as well as making certain that all of the dependencies (network, file system locations) are available; testing in disparate environments allows you to discover these errors before users do.

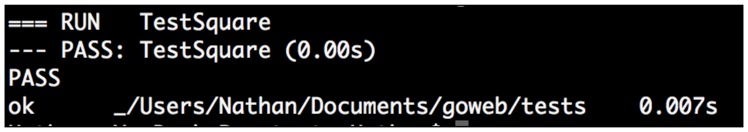
If you're already utilizing unit tests, Go's implementation will be both familiar and pleasant to get started in:

```
package example

func Square(x int) int {
  y := x * x
  return y
}
```

This is saved as `example.go`. Next, create another Go file that tests this square root functionality, with the following code:

```
package example

import (
  "testing"
)

func TestSquare(t *testing.T) {
  if v := Square(4); v != 16 {
    t.Error("expected", 16, "got", v)
  }
}
```

You can run this by entering the directory and simply typing `go test -v`. As expected, this passes given our test input:

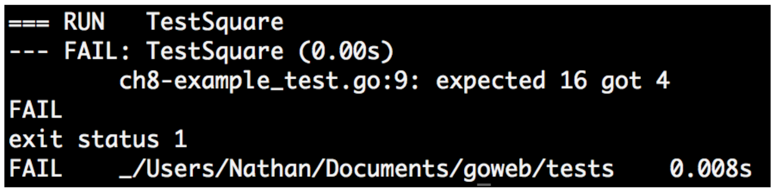


```
=== RUN   TestSquare
--- PASS: TestSquare (0.00s)
PASS
ok      _/Users/Nathan/Documents/goweb/tests    0.007s
```

This example is obviously trivial, but to demonstrate what you will see if your tests fail, let's change our `Square()` function as shown:

```
func Square(x int) int {
  y := x
  return y
}
```

And again after running the test, we get:



```
=== RUN   TestSquare
--- FAIL: TestSquare (0.00s)
        ch8-example_test.go:9: expected 16 got 4
FAIL
exit status 1
FAIL    _/Users/Nathan/Documents/goweb/tests    0.008s
```

Running command-line tests against command-line applications is different than interacting with the Web. Our application being the one that includes standard HTML endpoints as well as API endpoints; testing it requires more nuance than the approach we used earlier.

Luckily, Go also includes a package for specifically testing the results of an HTTP application, `net/http/httptest`.

Unlike the preceding example, `httptest` lets us evaluate a number of pieces of metadata returned from our individual functions, which act as handlers in the HTTP version of unit tests.

So let's look at a simple way of evaluating what our HTTP server might be producing, by generating a quick endpoint that simply returns the day of the year.

To begin, we'll add another endpoint to our API. Lets separate this handler example into its own application to isolate its impact:

```
package main

import (
  "fmt"
  "net/http"
  "time"
)

func testHandler(w http.ResponseWriter, r *http.Request) {
  t := time.Now()
  fmt.Fprintln(w, t.YearDay())
}

func main() {
  http.HandleFunc("/test", testHandler)
  http.ListenAndServe(":8080", nil)
}
```

This will simply return the day (1-366) of the year through the HTTP endpoint `/test`. So how do we test this?

First, we need a new file specifically for testing. When it comes to how much test coverage you'll need to hit, which is often helpful to the developer or organization—ideally we'd want to hit every endpoint and method to get a fairly comprehensive coverage. For this example, we'll make sure that one of our API endpoints returns a proper status code and that a `GET` request returns what we expect to see in the development:

```
package main

import (
  "io/ioutil"
  "net/http"
  "net/http/httptest"
  "testing"
)

func TestHandler(t *testing.T) {
  res := httptest.NewRecorder()
  path := "http://localhost:4000/test"
  o, err := http.NewRequest("GET", path, nil)
  http.DefaultServeMux.ServeHTTP(res, req)
```

```
    response, err := ioutil.ReadAll(res.Body)
    if string(response) != "115" || err != nil {
      t.Errorf("Expected [], got %s", string(response))
    }
  }
```

Now, we can implement this in our actual application by making certain that our endpoints pass (200) or fail (404) and return the text we expect them to return. We could also automate adding new content and validating it, and you should be equipped to take that on after these examples.

Given the fact that we have a hello-world endpoint, let's write a quick test that validates our response from the endpoint and have a look at how we can get a proper response in a `test.go` file:

```
package main

import (
  "net/http"
  "net/http/httptest"
  "testing"
)

func TestHelloWorld(t *testing.T) {


  req, err := http.NewRequest("GET", "/page/hello-world", nil)
  if err != nil {
    t.Fatal("Creating 'GET /page/hello-world' request failed!")
  }
  rec := httptest.NewRecorder()
  Router().ServeHTTP(rec, req)
}
```

Here we can test that we're getting the status code we expect, which is not necessarily a trivial test despite its simplicity. In practice, we'd probably also create one that should fail and another test that checks to make sure that we get the HTTP response we expect. But this sets the stage for more complex test suites, such as sanity tests or deployment tests. For example, we might generate development-only pages that generate HTML content from templates and check the output to ensure our page access and our template parsing work as we expect.

Read more about the testing with http and the httptest package at `https://golang.org/pkg/net/http/httptest/`

Summary

Simply building an application is not even half the battle and user-testing as a developer introduces a huge gap in testing strategy. Test coverage is a critical weapon when it comes to finding bugs, before they ever manifest to an end user.

Luckily, Go provides all the tools necessary to implement automated unit tests and the logging architecture necessary to support it.

In this chapter, we looked at both loggers and testing options. By producing multiple loggers for different messages, we were able separate warnings from errors brought about by internal application failures.

We then examined unit testing using the test and the `httptest` packages to automatically check our application and keep it current by testing for potential breaking changes.

In *Chapter 9*, *Security*, we'll look at implementing security more thoroughly; from better TLS/SSL, to preventing injection and man-in-the-middle and cross-site request forgery attacks in our application.

9
Security

In the previous chapter we looked at how to store information generated by our application as it works as well as adding unit tests to our suite to ensure that the application behaves as we expect it to and diagnose errors when it does not.

In that chapter, we did not add a lot of functionality to our blog app; so let's get back to that now. We'll also extend some of the logging and testing functionality from this chapter into our new features.

Till now, we have been working on the skeleton of a web application that implements some basic inputs and outputs of blog data and user-submitted comments. Just like any public networked server, ours is subject to a variety of attack vectors.

None of these are unique to Go, but we have an arsenal of tools at our disposal to implement the best practices and extend our server and application to mitigate common issues.

When building a publicly accessible networked application, one quick and easy reference guide for common attack vectors is the **Open Web Application Security Project** (**OWASP**), which provides a periodically updated list of the most critical areas where security issues manifest. OWASP can be found at `https://www.owasp.org/`. Its Top Ten Project compiles the 10 most common and/or critical network security issues. While it's not a comprehensive list and has a habit of becoming dated between updates, but it still remains a good first start when compiling potential vectors.

A few of the most pervasive vectors of the years have unfortunately stuck around; despite the fact that security experts have been shouting from the rooftops of their severity. Some have seen a rapid decrease in exposure across the Web (like injection), but they still tend to stick around longer, for years and years, even as legacy applications phase out.

Here is a glimpse of four of the most recent top 10 vulnerabilities, from late 2013, some of which we'll look at in this chapter:

- **Injections**: Any case where untrusted data has an opportunity to be processed without escaping, thus allowing data manipulation or access to data or systems, normally its not exposed publicly. Most commonly this is an SQL injection.
- **Broken authentication**: This is caused due to poor encryption algorithms, weak password requirements, session hijacking is feasible.
- **XSS**: Cross-site scripting allows an attacker to access sensitive information by injecting and executing scripts on another site.
- **Cross-site request forgery**: Unlike XSS, this allows the attack vector to originate from another site, but it fools a user into completing some action on another site.

While the other attack vectors range from being relevant to irrelevant for our use case, it is worth evaluating the ones that we aren't covering, to see what other places might be rife for exploitation.

To get going, we'll look at the best ways to implement and force HTTPS in your applications using Go.

HTTPS everywhere – implementing TLS

In *Chapter 5, Frontend Integration with RESTful APIs*, we looked at creating self-signed certificates and utilizing HTTPS/TLS in our app. But let's review quickly why this matters so much in terms of overall security for not just our application but the Web in general.

First, simple HTTP generally produces no encryption for traffic, particularly for vital request header values, such as cookies and query parameters. We say generally here because RFC 2817 does specify a system use TLS over the HTTP protocol, but it's all but unused. Most importantly, it would not give users the type of palpable feedback necessary to register that a site is secure.

Second and similarly, HTTP traffic is subsequently vulnerable to man-in-the-middle attacks.

One other side effect: Google (and perhaps other search engines) begun to favor HTTPS traffic over less secure counterparts.

Until relatively recently, HTTPS was relegated primarily to e-commerce applications, but the rise in available and prevalent attacks utilizing the deficiencies of HTTP—like sidejacking and man-in-the-middle attacks—began to push much of the Web toward HTTPS.

You may have heard of the resulting movement and motto **HTTPS Everywhere**, which also bled into the browser plugins that force site usage to implement the most secure available protocol for any given site.

One of the easiest things we can do to extend the work in *Chapter 6*, *Session and Cookies* is to require that all traffic goes through HTTPS by rerouting the HTTP traffic. There are other ways of doing this, as we'll see at the end of the chapter, but it can be accomplished fairly simply.

First, we'll implement a `goroutine` to concurrently serve our HTTPS and HTTP traffic using the `tls.ListenAndServe` and `http.ListenAndServe` respectively:

```
    var wg sync.WaitGroup
    wg.Add(1)
    go func() {
      http.ListenAndServe(PORT, http.HandlerFunc(redirectNonSecure))
      wg.Done()
    }()
    wg.Add(1)
    go func() {
      http.ListenAndServeTLS(SECUREPORT, "cert.pem", "key.pem",
  routes)
      wg.Done()
    }()

    wg.Wait()
```

This assumes that we set a `SECUREPORT` constant to, likely, `":443"` just as we set `PORT` to `":8080"`, or whatever you chose. There's nothing preventing you from using another port for HTTPS; the benefit here is that the browser directs `https://` requests to port `443` by default, just as it directs HTTP requests to ports `80` and sometimes fallback to port `8080`. Remember that you'll need to run as sudo or administrator in many cases to launch with ports below `1000`.

You'll note in the preceding example that we're utilizing a separate handler for HTTP traffic called `redirectNonSecure`. This fulfills a very basic purpose, as you'll see here:

```
func redirectNonSecure(w http.ResponseWriter, r *http.Request) {
  log.Println("Non-secure request initiated, redirecting.")
  redirectURL := "https://" + serverName + r.RequestURI
  http.Redirect(w, r, redirectURL, http.StatusMovedPermanently)
}
```

Here, `serverName` is set explicitly.

There are some potential issues with gleaning the domain or server name from the request, so it's best to set this explicitly if you can.

Another very useful piece to add here is **HTTP Strict Transport Security** (**HSTS**), an approach that, when combined with compliant consumers, aspires to mitigate protocol downgrade attacks (such as forcing/redirecting to HTTP).

This is nothing more than an HTTPS header that, when consumed, will automatically handle and force the `https://` requests for requests that would otherwise utilize less secure protocols.

OWASP recommends the following setting for this header:

```
Strict-Transport-Security: max-age=31536000; includeSubDomains;
preload
```

Note that this header is ignored over HTTP.

Preventing SQL injection

While injection remains one of the biggest attack vectors across the Web today, most languages have simple and elegant ways of preventing or largely mitigating the odds of leaving vulnerable SQL injections in place with prepared statements and sanitized inputs.

But even with languages that provide these services, there is still an opportunity to leave areas open for exploits.

One of the core tenets of any software development whether on the Web or a server or a standalone executable is to never trust input data acquired from an external (and sometimes internal) source.

This tenet stands true for any language, though some make interfacing with a database safer and/or easier either through prepared queries or abstractions, such as **Object-relational mapping** (**ORM**).

Natively, Go doesn't have any ORM and since there technically isn't even an O (Object) (Go not being purely object-oriented), it's hard to replicate a lot of what object-oriented languages have in this area.

There are, however, a number of third-party libraries that attempt to coerce ORM through interfaces and structs, but a lot of this could be very easily written by hand since you probably know your schemas and data structures better than any library, even in the abstract sense.

For SQL, however, Go has a robust and consistent interface for almost any database that supports SQL.

To show how an SQL injection exploit can simply surface in a Go application, we'll compare a raw SQL query to a prepared statement.

When we select pages from our database, we use the following query:

```
err := database.QueryRow("SELECT page_title,page_content,page_date
FROM pages WHERE page_guid="+requestGUID, pageGUID).Scan(&this
Page.Title, &thisPage.Content, &thisPage.Date)
```

This shows us how to open up your application to injection vulnerabilities by accepting unsanitized user input. In this case, anyone requesting a page like

`/page/foo;delete from pages` could, in theory, empty your `pages` table in a hurry.

We have some preliminary sanitization at the router level that does help in this regard. As our mux routes only include alphanumeric characters, we can avoid some of the characters that would otherwise need to be escaped being routed to our `ServePage` or `APIPage` handlers:

```
  routes.HandleFunc("/page/{guid:[0-9a-zA\\-]+}", ServePage)
  routes.HandleFunc("/api/page/{id:[\\w\\d\\-]+}", APIPage).
    Methods("GET").
    Schemes("https")
```

This is not a foolproof way of addressing this, though. The preceding query took raw input and appended it to the SQL query, but we can handle this much better with parameterized, prepared queries in Go. The following is what we ended up using:

```
  err := database.QueryRow("SELECT page_title,page_con
tent,page_date FROM pages WHERE page_guid=?",
pageGUID).Scan(&thisPage.Title, &thisPage.Content, &thisPage.Date)
```

```
    if err != nil {
      http.Error(w, http.StatusText(404), http.StatusNotFound)
      log.Println("Couldn't get page!")
      return
    }
```

This approach is available in any of Go's query interfaces, which take a query using `?` in place of values as a variadic:

```
res, err := db.Exec("INSERT INTO table SET field=?, field2=?",
value1, value2)
rows, err := db.Query("SELECT * FROM table WHERE field2=?",value2)
statement, err := db.Prepare("SELECT * FROM table WHERE
field2=?",value2)
row, err := db.QueryRow("SELECT * FROM table WHERE
field=?",value1)
```

While all of these fulfill a slightly different purpose within the world of SQL, they all implement the prepared query in the same way.

Protecting against XSS

We've touched briefly upon cross-site scripting and limiting this as a vector makes your application safer for all users, against the actions of a few bad apples. The crux of the issue is the ability for one user to add dangerous content that will be shown to users without scrubbing out the aspects that make it dangerous.

Ultimately you have a choice here—sanitize the data as it comes in or sanitize the data as you present it to other users.

In other words, if someone produces a block of comment text that includes a `script` tag, you must take care to stop that from ever being rendered by another user's browser. You can choose to save the raw HTML and then strip all, or only the sensitive tags on the output rendering. Or, you can encode it as it's entered.

There's no right answer; however, you may discover value in following the former approach, where you accept anything and sanitize the output.

There is risk with either, but this approach allows you to keep the original intent of the message should you choose to change your approach down the road. The downside is that of course you can accidentally allow some of this raw data to slip through unsanitized:

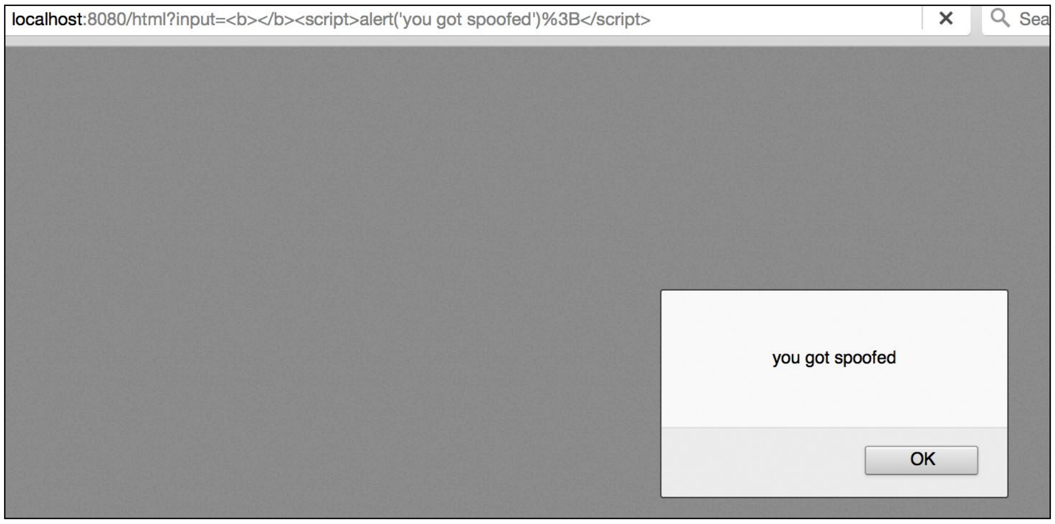
```
template.HTMLEscapeString(string)
template.JSEscapeString(inputData)
```

The first function will take the data and remove the formatting of the HTML to produce a plaintext version of the message input by a user.

The second function will do something similar but for JavaScript-specific values. You can test these very easily with a quick script similar to the following example:

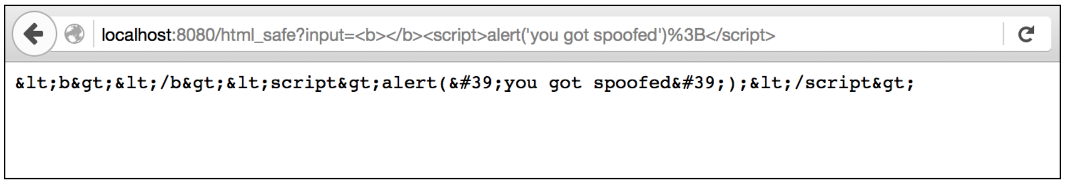
```
package main

import (
  "fmt"
  "github.com/gorilla/mux"
  "html/template"
  "net/http"
)

func HTMLHandler(w http.ResponseWriter, r *http.Request) {
  input := r.URL.Query().Get("input")
  fmt.Fprintln(w, input)
}

func HTMLHandlerSafe(w http.ResponseWriter, r *http.Request) {
  input := r.URL.Query().Get("input")
  input = template.HTMLEscapeString(input)
  fmt.Fprintln(w, input)
}

func JSHandler(w http.ResponseWriter, r *http.Request) {
  input := r.URL.Query().Get("input")
  fmt.Fprintln(w, input)
}

func JSHandlerSafe(w http.ResponseWriter, r *http.Request) {
  input := r.URL.Query().Get("input")
  input = template.JSEscapeString(input)
  fmt.Fprintln(w, input)
}

func main() {
  router := mux.NewRouter()
  router.HandleFunc("/html", HTMLHandler)
  router.HandleFunc("/js", JSHandler)
  router.HandleFunc("/html_safe", HTMLHandlerSafe)
  router.HandleFunc("/js_safe", JSHandlerSafe)
  http.ListenAndServe(":8080", router)
}
```

If we request from the unsafe endpoint, we'll get our data back:



```
localhost:8080/html?input=<b></b><script>alert('you got spoofed')%3B</script>
```

Compare this with `/html_safe`, which automatically escapes the input, where you can see the content in its sanitized form:



```
localhost:8080/html_safe?input=<b></b><script>alert('you got spoofed')%3B</script>
```

```
&lt;b&gt;&lt;/b&gt;&lt;script&gt;alert(&#39;you got spoofed&#39;);&lt;/script&gt;
```

None of this is foolproof, but if you choose to take input data as the user submits it, you'll want to look at ways to relay that information on resulting display without opening up other users to XSS.

Preventing cross-site request forgery (CSRF)

While we won't go very deeply into CSRF in this book, the general gist is that it is a slew of methods that malicious actors can use to fool a user into performing an undesired action on another site.

As it's at least tangentially related to XSS in approach, it's worth talking about now.

The biggest place where this manifests is in forms; think of it as a Twitter form that allows you to send tweets. If a third party forced a request on a user's behalf without their consent, think of something similar to this:

```
<h1>Post to our guestbook (and not twitter, we swear!)</h1>
  <form action="https://www.twitter.com/tweet" method="POST">
  <input type="text" placeholder="Your Name" />
  <textarea placeholder="Your Message"></textarea>
  <input type="hidden" name="tweet_message" value="Make sure to
check out this awesome, malicious site and post on their
guestbook" />
  <input type="submit" value="Post ONLY to our guestbook" />
</form>
```

Without any protection, anyone who posts to this guestbook would inadvertently help spread spam to this attack.

Obviously, Twitter is a mature application that has long ago handled this, but you get the general idea. You might think that restricting referrers will fix this problem, but that can also be spoofed.

The shortest solution is to generate secure tokens for form submissions, which prevents other sites from being able to construct a valid request.

Of course, our old friend Gorilla also provides a few helpful tools in this regard. Most relevant is the `csrf` package, which includes tools to produce tokens for requests as well as prebaked form fields that will produce `403` if violated or ignored.

The simplest way to produce a token is to include it as part of the interface that your handler will be using to produce a template, as so from our `ApplicationAuthenticate()` handler:

```
    Authorize.TemplateTag = csrf.TemplateField(r)
    t.ExecuteTemplate(w, "signup_form.tmpl", Authorize)
```

At this point we'll need to expose `{{.csrfField}}` in our template. To validate, we'll need to chain it to our `ListenAndServe` call:

```
    http.ListenAndServe(PORT, csrf.Protect([]byte("32-byte-long-
auth-key"))(r))
```

Securing cookies

One of the attack vectors we looked at earlier was session hijacking, which we discussed in the context of HTTP versus HTTPS and the way others can see the types of information that are critical to identity on a website.

Finding this data is incredibly simple on public networks for a lot of non-HTTPS applications that utilize sessions as definitive IDs. In fact, some large applications have allowed session IDs to be passed in URLs

In our application, we've used Gorilla's `securecookie` package, which does not rely on HTTPS because the cookie values themselves are encoded and validated using HMAC hashing.

Producing the key itself can be very simple, as demonstrated in our application and the `securecookie` documentation:

```
var hashKey = []byte("secret hash key")
var blockKey = []byte("secret-er block key")
var secureKey = securecookie.New(hashKey, blockKey)
```

For more info on Gorilla's `securecookie` package see: `http://www.gorillatoolkit.org/pkg/securecookie`

Presently, our app's server has HTTPS first and secure cookies, which means that we likely feel a little more confident about storing and identifying data in the cookie itself. Most of our create/update/delete operations are happening at the API level, which still implements session checking to ensure our users are authenticated.

Using the secure middleware

One of the more helpful packages for quickly implementing some of the security fixes (and others) mentioned in this chapter is a package from Cory Jacobsen called, helpfully, `secure`.

Secure offers a host of useful utilities, such as SSLRedirects (as we implemented in this chapter), allowed Hosts, HSTS options, and X-Frame-Options shorthand for preventing your site from being loaded into frames.

A good amount of this covers some of the topics that we looked at in this chapter and is largely the best practice. As a piece of middleware, secure can be an easy way to quickly cover some of those best practices in one swoop.

To grab `secure`, simply go get it at `github.com/unrolled/secure`.

Summary

While this chapter is not a comprehensive review of web security issues and solutions, we hoped to address some of the biggest and most common vectors as surfaced by OWASP and others.

Within this chapter we covered or reviewed the best practices to prevent some of these issues from creeping into your applications.

In *Chapter 10*, *Caching, Proxies, and Improved Performance*, we'll look at how to make your application scale with increased traffic while remaining performant and speedy.

10
Caching, Proxies and Improved Performance

We have covered a great deal about the web application that you'll need to connect to data sources, render templates, utilize SSL/TLS, build APIs for single-page applications, and so on.

While the fundamentals are clear, you may find that putting an application built on these guidelines into production would lead to some quick problems, particularly under heavy load.

We've implemented some of the best security practices in the last chapter by addressing some of the most common security issues in web applications. Let's do the same here in this chapter, by applying the best practices against some of the biggest issues of performance and speed.

To do this, we'll look at some of the most common bottlenecks in the pipeline and see how we can reduce these to make our application as performant as possible in production.

Specifically, we'll be identifying those bottlenecks and then looking to reverse proxies and load balancing, implementing caching into our application, utilizing **SPDY**, and look at how to use managed cloud services to augment our speed initiatives by reducing the number of requests that get to our application.

By this chapter's end, we hope to produce tools that can help any Go application squeeze every bit of performance out of our environment.

In this chapter, we will cover the following topics:

- Identifying bottlenecks
- Implementing reverse proxies

- Implementing caching strategies
- Implementing HTTP/2

Identifying bottlenecks

To simplify things a little, there are two types of bottlenecks for your application, those caused by development and programming deficiencies and those inherent to an underlying software or infrastructure limitation.

The answer to the former is simple, identify the poor design and fix it. Putting patches around bad code can hide the security vulnerabilities or delay even bigger performance issues from being discovered in a timely manner.

Sometimes these issues are born from a lack of stress testing; a code that is performant locally is not guaranteed to scale without applying artificial load. A lack of this testing sometimes leads to surprise downtime in production.

However, ignoring bad code as a source of issues, lets take a look at some of the other frequent offenders:

- Disk I/O
- Database access
- High memory/CPU usage
- Lack of concurrency support

There are of course hundreds of offenders, such as network issues, garbage collection overhead in some applications, not compressing payloads/headers, non-database deadlocks, and so on.

High memory and CPU usage is most often the result rather than the cause, but a lot of the other causes are specific to certain languages or environments.

For our application, we could have a weak point at the database layer. Since we're doing no caching, every request will hit the database multiple times. ACID-compliant databases (such as MySQL/PostgreSQL) are notorious for failing under loads, which would not be a problem on the same hardware for less strict key/value stores and NoSQL solutions. The cost of database consistency contributes heavily to this and it's one of the trade-offs of choosing a traditional relational database.

Implementing reverse proxies

As we know by now, unlike a lot of languages, Go comes with a complete and mature web server platform with `net/http`.

Of late, some other languages have been shipped with small toy servers intended for local development, but they are not intended for production. In fact, many specifically warn against it. Some common ones are WEBrick for Ruby, Python's SimpleHTTPServer, and PHP's -S. Most of these suffer from concurrency issues that prevent them from being viable choices in production.

Go's `net/http` is different; by default, it handles these issues with aplomb out of the box. Obviously, much of this depends on the underlying hardware, but in a pinch you could use it natively with success. Many sites are using `net/http` to serve non-trivial amounts of traffic.

But even strong underlying web servers have some inherent limitations:

- They lack failover or distributed options
- They have limited caching options upstream
- They cannot easily load balance the incoming traffic
- They cannot easily concentrate on centralized logging

This is where a reverse proxy comes into play. A reverse proxy accepts all the incoming traffic on behalf of one or more servers and distributes it by applying the preceding (and other) options and benefits. Another example is URL rewriting, which is more applicable for underlying services that may not have built-in routing and URL rewriting.

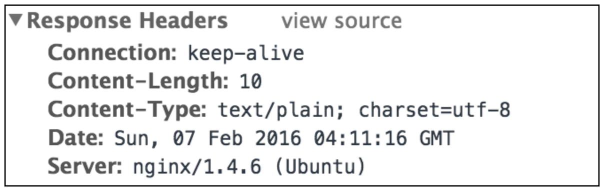
There are two big advantages of throwing a simple reverse proxy in front of your web server, such as Go; they are caching options and the ability to serve static content without hitting the underlying application.

One of the most popular options for reverse proxying sites is Nginx (pronounced Engine-X). While Nginx is a web server itself, it gained acclaim early on for being lightweight with a focus on concurrency. It quickly became the frontend du jour for front line defense of a web application in front of an otherwise slower or heavier web server, such as Apache. The situation has changed a bit in recent years, as Apache has caught up in terms of concurrency options and utilization of alternative approaches to events and threading. The following is an example of a reverse proxy Nginx configuration:

```
server {
  listen 80;
```

```
    root /var/;
    index index.html index.htm;

    large_client_header_buffers 4 16k;

    # Make site accessible from http://localhost/
    server_name localhost

    location / {
      proxy_pass http://localhost:8080;
      proxy_redirect off;
      proxy_set_header Host $host;
      proxy_set_header X-Real-IP $remote_addr;
      proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
    }

  }
```

With this in place, make sure that your Go app is running on port `8080` and restart Nginx. Requests to `http//:port 80` will be served through Nginx as a reverse proxy to your application. You can check this through viewing headers or in the **Developer tools** in your browser:



```
▼Response Headers        view source
   Connection: keep-alive
   Content-Length: 10
   Content-Type: text/plain; charset=utf-8
   Date: Sun, 07 Feb 2016 04:11:16 GMT
   Server: nginx/1.4.6 (Ubuntu)
```

Remember that we wish to support TLS/SSL whenever possible, but providing a reverse proxy here is just a matter of changing the ports. Our application should run on another port, likely a nearby port for clarity and then our reverse proxy would run on port `443`.

As a reminder, any port is legal for HTTP or HTTPS. However, when a port is not specified, the browsers automatically direct to `443` for secure connections. It's as simple as modifying the `nginx.conf` and our app's constant:

```
  server {
    listen 443;
    location / {
       proxy_pass http://localhost:444;
```

Lets see how to modify our application as shown in the following code:

```
const (
  DBHost  = "127.0.0.1"
  DBPort  = ":3306"
  DBUser  = "root"
  DBPass  = ""
  DBDbase = "cms"
  PORT    = ":444"
)
```

This allows us to pass through SSL requests with a frontend proxy.

On many Linux distributions, you'll need SUDO or root privileges to use ports below 1000.

Implementing caching strategies

There are a number of ways to decide when to create and when to expire the cache items, so we'll look at one of the easier and faster methods for doing so. But if you are interested in developing this further, you might consider other caching strategies; some of which can provide efficiencies for resource usage and performance.

Using Least Recently Used

One common tactic to maintain cache stability within allocated resources (disk space, memory) is the **Least Recently Used** (**LRU**) system for cache expiration. In this model, utilizing information about the last cache access time (creation or update) and the cache management system can remove the oldest entry in the list.

This has a number of benefits for performance. First, if we assume that the most recently created/updated cache entries are for entries that are presently the most popular, we can remove entries that are not being accessed much sooner; in order to free up the resources for the existing and new resources that might be accessed much more frequently.

This is a fair assumption, assuming the allocated resources for caching is not inconsequential. If you have a large volume for file cache or a lot of memory for memcache, the oldest entries, in terms of last access, are quite likely not being utilized with great frequency.

There is a related and more granular strategy called Least Frequently Used that maintains strict statistics on the usage of the cache entries themselves. This not only removes the need for assumptions about cache data but also adds overhead for the statistics maintenance.

For our demonstrations here, we will be using LRU.

Caching by file

Our first approach is probably best described as a classical one for caching, but a method not without issues. We'll utilize the disk to create file-based caches for individual endpoints, both API and Web.

So what are the issues associated with caching in the filesystem? Well, previously in the chapter, we mentioned that disk can introduce its own bottleneck. Here, we're doing a trade-off to protect the access to our database in lieu of potentially running into other issues with disk I/O.

This gets particularly complicated if our cache directory gets very big. At this point we end up introducing more file access issues.

Another downside is that we have to manage our cache; because the filesystem is not ephemeral and our available space is. We'll need to be able to expire cache files by hand. This introduces another round of maintenance and another point of failure.

All that said, it's still a useful exercise and can still be utilized if you're willing to take on some of the potential pitfalls:

```
package cache

const (
  Location "/var/cache/"
)

type CacheItem struct {
  TTL int
  Key string
}

func newCache(endpoint string, params ...[]string) {

}

func (c CacheItem) Get() (bool, string) {
  return true, ""
}

func (c CacheItem) Set() bool {

}

func (c CacheItem) Clear() bool {

}
```

This sets the stage to do a few things, such as create unique keys based on an endpoint and query parameters, check for the existence of a cache file, and if it does not exist, get the requested data as per normal.

In our application, we can implement this simply. Let's put a file caching layer in front of our `/page` endpoint as shown:

```
func ServePage(w http.ResponseWriter, r *http.Request) {
  vars := mux.Vars(r)
  pageGUID := vars["guid"]
  thisPage := Page{}
  cached := cache.newCache("page",pageGUID)
```

The preceding code creates a new `CacheItem`. We utilize the variadic `params` to generate a reference filename:

```
func newCache(endpoint string, params ...[]string) CacheItem {
cacheName := endponit + "_" + strings.Join(params, "_")
c := CacheItem{}
return c
}
```

When we have a `CacheItem` object, we can check using the `Get()` method, which will return `true` if the cache is still valid, otherwise the method will return `false`. We utilize filesystem information to determine if a cache item is within its valid time-to-live:

```
  valid, cachedData := cached.Get()
  if valid {
    thisPage.Content = cachedData
    fmt.Fprintln(w, thisPage)
    return
  }
```

If we find an existing item via the `Get()` method, we'll check to make sure that it has been updated within the set `TTL`:

```
func (c CacheItem) Get() (bool, string) {

  stats, err := os.Stat(c.Key)
  if err != nil {
    return false, ""
  }

  age := time.Nanoseconds() - stats.ModTime()
  if age <= c.TTL {
    cache, _ := ioutil.ReadFile(c.Key)
    return true, cache
  } else {
    return false, ""
  }
}
```

If the code is valid and within the TTL, we'll return `true` and the file's body will be updated. Otherwise, we will allow a passthrough to the page retrieval and generation. At the tail of this we can set the cache data:

```
  t, _ := template.ParseFiles("templates/blog.html")
  cached.Set(t, thisPage)
  t.Execute(w, thisPage)
```

We then save this as:

```
func (c CacheItem) Set(data []byte) bool {
  err := ioutil.WriteFile(c.Key, data, 0644)
}
```

This function effectively writes the value of our cache file.

We now have a working system that will take individual endpoints and innumerable query parameters and create a file-based cache library, ultimately preventing unnecessary queries to our database, if data has not been changed.

In practice we'd want to limit this to mostly read-based pages and avoid putting blind caching on any write or update endpoints, particularly on our API.

Caching in memory

Just as file system caching became a lot more palatable because storage prices plummeted, we've seen a similar move in RAM, trailing just behind hard storage. The big advantage here is speed, caching in memory can be insanely fast for obvious reasons.

Memcache, and its distributed sibling Memcached, evolved out of a need to create a light and super-fast caching for LiveJournal and a proto-social network from *Brad Fitzpatrick*. If that name feels familiar, it's because Brad now works at Google and is a serious contributor to the Go language itself.

As a drop-in replacement for our file caching system, Memcached will work similarly. The only major change is our key lookups, which will be going against working memory instead of doing file checks.

> To use memcache with Go language, go to `godoc.org/github.com/bradfitz/gomemcache/memcache` from *Brad Fitz*, and install it using `go get` command.

Implementing HTTP/2

One of the more interesting, perhaps noble, initiatives that Google has invested in within the last five years has been a focus on making the Web faster. Through tools, such as PageSpeed, Google has sought to push the Web as a whole to be faster, leaner, and more user-friendly.

No doubt this initiative is not entirely altruistic. Google has built their business on extensive web search and crawlers are always at the mercy of the speed of the pages they crawl. The faster the web pages, the faster and more comprehensive is the crawling; therefore, less time and less infrastructure resulting in less money required. The bottom line here is that a faster web benefits Google, as much as it does people creating and viewing web sites.

But this is mutually beneficial. If web sites are faster to comply with Google's preferences, everyone benefits with a faster Web.

This brings us to HTTP/2, a version of HTTP that replaces 1.1, introduced in 1999 and largely the defacto method for most of the Web. HTTP/2 also envelops and implements a lot of SPDY, a makeshift protocol that Google developed and supported through Chrome.

HTTP/2 and SPDY introduce a host of optimizations including header compression and non-blocking and multiplexed request handling.

If you're using version 1.6, `net/http` supports HTTP/2 out of the box. If you're using version 1.5 or earlier, you can use the experimental package.

> To use HTTP/2 prior to Go version 1.6, go get it from `godoc.org/golang.org/x/net/http2`

Summary

In this chapter, we focused on quick wins for increasing the overall performance for our application, by reducing impact on our underlying application's bottlenecks, namely our database.

We've implemented caching at the file level and described how to translate that into a memory-based caching system. We looked at SPDY and HTTP/2, which has now become a part of the underlying Go `net/http` package by default.

This in no way represents all the optimizations that we may need to produce highly performant code, but hits on some of the most common bottlenecks that can keep applications that work well in development from behaving similarly in production under heavy load.

This is where we end the book; hope you all enjoyed the ride!

Module 2

Go Programming Blueprints

Build real-world, production-ready solutions in Go using cutting-edge technology and techniques

1 Chat Application with Web Sockets

Go is great for writing high-performance, concurrent server applications and tools, and the Web is the perfect medium over which to deliver them. It would be difficult these days to find a gadget that is not web-enabled and allows us to build a single application that targets almost all platforms and devices.

Our first project will be a web-based chat application that allows multiple users to have a real-time conversation right in their web browser. Idiomatic Go applications are often composed of many packages, which are organized by having code in different folders, and this is also true of the Go standard library. We will start by building a simple web server using the `net/http` package, which will serve the HTML files. We will then go on to add support for web sockets through which our messages will flow.

In languages such as C#, Java, or Node.js, complex threading code and clever use of locks need to be employed in order to keep all clients in sync. As we will see, Go helps us enormously with its built-in channels and concurrency paradigms.

In this chapter, you will learn how to:

- Use the `net/http` package to serve HTTP requests
- Deliver template-driven content to users' browsers
- Satisfy a Go interface to build our own `http.Handler` types
- Use Go's goroutines to allow an application to perform multiple tasks concurrently
- Use channels to share information between running Go routines
- Upgrade HTTP requests to use modern features such as web sockets

- Add tracing to the application to better understand its inner workings
- Write a complete Go package using test-driven development practices
- Return unexported types through exported interfaces

Complete source code for this project can be found at `https://github.com/matryer/goblueprints/tree/master/chapter1/chat`. The source code was periodically committed so the history in GitHub actually follows the flow of this chapter too.

A simple web server

The first thing our chat application needs is a web server that has two main responsibilities: it must serve the HTML and JavaScript chat clients that run in the user's browser and accept web socket connections to allow the clients to communicate.

Create a `main.go` file inside a new folder called `chat` in your `GOPATH` and add the following code:

```
package main

import (
  "log"
  "net/http"
)

func main() {

  http.HandleFunc("/", func(w http.ResponseWriter, r
*http.Request) {
    w.Write([]byte(`
      <html>
        <head>
          <title>Chat</title>
        </head>
        <body>
          Let's chat!
        </body>
      </html>
    `))
  })
```

```
    // start the web server
    if err := http.ListenAndServe(":8080", nil); err != nil {
      log.Fatal("ListenAndServe:", err)
    }
  }
```

This is a complete albeit simple Go program that will:

- Listen to the root path using the `net/http` package
- Write out the hardcoded HTML when a request is made
- Start a web server on port `:8080` using the `ListenAndServe` method

The `http.HandleFunc` function maps the path pattern `"/"` to the function we pass as the second argument, so when the user hits `http://localhost:8080/`, the function will be executed. The function signature of `func(w http.ResponseWriter, r *http.Request)` is a common way of handling HTTP requests throughout the Go standard library.

> We are using `package main` because we want to build and run our program from the command line. However, if we were building a reusable chatting package, we might choose to use something different, such as `package chat`.

In a terminal, run the program by navigating to the `main.go` file you just created and execute:

```
go run main.go
```

Open a browser to `localhost:8080` to see the **Let's chat!** message.

Having the HTML code embedded within our Go code like this works, but it is pretty ugly and will only get worse as our projects grow. Next, we will see how templates can help us clean this up.

Templates

Templates allow us to blend generic text with specific text, for instance, injecting a user's name into a welcome message. For example, consider the following template:

```
Hello {name}, how are you?
```

We are able to replace the `{name}` text in the preceding template with the real name of a person. So if Laurie signs in, she might see:

```
Hello Laurie, how are you?
```

The Go standard library has two main template packages: one called `text/template` for text and one called `html/template` for HTML. The `html/template` package does the same as the text version except that it understands the context in which data will be injected into the template. This is useful because it avoids script injection attacks and resolves common issues such as having to encode special characters for URLs.

Initially, we just want to move the HTML code from inside our Go code to its own file, but won't blend any text just yet. The template packages make loading external files very easy, so it's a good choice for us.

Create a new folder under our `chat` folder called `templates` and create a `chat.html` file inside it. We will move the HTML from `main.go` to this file, but we will make a minor change to ensure our changes have taken effect.

```
<html>
  <head>
    <title>Chat</title>
  </head>
  <body>
    Let's chat (from template)
  </body>
</html>
```

Now, we have our external HTML file ready to go, but we need a way to compile the template and serve it to the user's browser.

Compiling a template is a process by which the source template is interpreted and prepared for blending with various data, which must happen before a template can be used but only needs to happen once.

We are going to write our own `struct` type that is responsible for loading, compiling, and delivering our template. We will define a new type that will take a `filename` string, compile the template once (using the `sync.Once` type), keep the reference to the compiled template, and then respond to HTTP requests. You will need to import the `text/template`, `path/filepath`, and `sync` packages in order to build your code.

In `main.go`, insert the following code above the `func main()` line:

```
// templ represents a single template
type templateHandler struct {
  once     sync.Once
  filename string
  templ    *template.Template
}
// ServeHTTP handles the HTTP request.
func (t *templateHandler) ServeHTTP(w http.ResponseWriter, r
*http.Request) {
  t.once.Do(func() {
    t.templ =
template.Must(template.ParseFiles(filepath.Join("templates",
t.filename)))
  })
  t.templ.Execute(w, nil)
}
```

The `templateHandler` type has a single method called `ServeHTTP` whose signature looks suspiciously like the method we passed to `http.HandleFunc` earlier. This method will load the source file, compile the template and execute it, and write the output to the specified `http.ResponseWriter` object. Because the `ServeHTTP` method satisfies the `http.Handler` interface, we can actually pass it directly to `http.Handle`.

A quick look at the Go standard library source code, which is located at `http://golang.org/pkg/net/http/#Handler`, will reveal that the interface definition for `http.Handler` specifies that only the `ServeHTTP` method need be present in order for a type to be used to serve HTTP requests by the `net/http` package.

Doing things once

We only need to compile the template once, and there are a few different ways to approach this in Go. The most obvious is to have a `NewTemplateHandler` function that creates the type and calls some initialization code to compile the template. If we were sure the function would be called by only one goroutine (probably the main one during the setup in the `main` function), this would be a perfectly acceptable approach. An alternative, which we have employed in the preceding section, is to compile the template once inside the `ServeHTTP` method. The `sync.Once` type guarantees that the function we pass as an argument will only be executed once, regardless of how many goroutines are calling `ServeHTTP`. This is helpful because web servers in Go are automatically concurrent and once our chat application takes the world by storm, we could very well expect to have many concurrent calls to the `ServeHTTP` method.

Compiling the template inside the `ServeHTTP` method also ensures that our code does not waste time doing work before it is definitely needed. This lazy initialization approach doesn't save us much in our present case, but in cases where the setup tasks are time- and resource-intensive and where the functionality is used less frequently, it's easy to see how this approach would come in handy.

Using your own handlers

To implement our `templateHandler` type, we need to update the `main` body function so that it looks like this:

```
func main() {
  // root
  http.Handle("/", &templateHandler{filename: "chat.html"})
  // start the web server
  if err := http.ListenAndServe(":8080", nil); err != nil {
    log.Fatal("ListenAndServe:", err)
  }
}
```

The `templateHandler` structure is a valid `http.Handler` type so we can pass it directly to the `http.Handle` function and ask it to handle requests that match the specified pattern. In the preceding code, we created a new object of the type `templateHandler` specifying the filename as `chat.html` that we then take the address of (using the `&` **address of** operator) and pass it to the `http.Handle` function. We do not store a reference to our newly created `templateHandler` type, but that's OK because we don't need to refer to it again.

In your terminal, exit the program by pressing *Ctrl* + *C* before re-running it, then refresh your browser and notice the addition of the (from template) text. Now our code is much simpler than an HTML code and free from those ugly blocks.

Properly building and executing Go programs

Running Go programs using a `go run` command is great when our code is made up of a single `main.go` file. However, often we might quickly need to add other files. This requires us to properly build the whole package into an executable binary before running it. This is simple enough, and from now on, this is how you will build and run your programs in a terminal:

```
go build -o {name}
./{name}
```

The `go build` command creates the output binary using all the `.go` files in the specified folder, and the `-o` flag indicates the name of the generated binary. You can then just run the program directly by calling it by name.

For example, in the case of our chat application, we could run:

```
go build -o chat
./chat
```

Since we are compiling templates the first time the page is served, we will need to restart your web server program every time anything changes in order to see the changes take effect.

Modeling a chat room and clients on the server

All users (clients) of our chat application will automatically be placed in one big public room where everyone can chat with everyone else. The `room` type will be responsible for managing client connections and routing messages in and out, while the `client` type represents the connection to a single client.

[Go refers to classes as types and instances of those classes as objects.]

To manage our web sockets, we are going to use one of the most powerful aspects of the Go community—open source third-party packages. Every day new packages solving real-world problems are released, ready for you to use in your own projects and even allow you to add features, report and fix bugs, and get support.

> It is often unwise to reinvent the wheel unless you have a very good reason. So before embarking on building a new package, it is worth searching for any existing projects that might have already solved your very problem. If you find one similar project that doesn't quite satisfy your needs, consider contributing to the project and adding features. Go has a particularly active open source community (remember that Go itself is open source) that is always ready to welcome new faces or avatars.

We are going to use Gorilla Project's `websocket` package to handle our server-side sockets rather than write our own. If you're curious about how it works, head over to the project home page on GitHub, `https://github.com/gorilla/websocket`, and browse the open source code.

Modeling the client

Create a new file called `client.go` alongside `main.go` in the `chat` folder and add the following code:

```
package main
import (
  "github.com/gorilla/websocket"
)
// client represents a single chatting user.
type client struct {
  // socket is the web socket for this client.
  socket *websocket.Conn
  // send is a channel on which messages are sent.
  send chan []byte
  // room is the room this client is chatting in.
  room *room
}
```

In the preceding code, socket will hold a reference to the web socket that will allow us to communicate with the client, and the `send` field is a buffered channel through which received messages are queued ready to be forwarded to the user's browser (via the socket). The `room` field will keep a reference to the room that the client is chatting in—this is required so that we can forward messages to everyone else in the room.

If you try to build this code, you will notice a few errors. You must ensure that you have called `go get` to retrieve the `websocket` package, which is as easy as opening a terminal and typing the following:

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

Building the code again will yield another error:

```
./client.go:17 undefined: room
```

The problem is that we have referred to a `room` type without defining it anywhere. To make the compiler happy, create a file called `room.go` and insert the following placeholder code:

```
package main
type room struct {
  // forward is a channel that holds incoming messages
  // that should be forwarded to the other clients.
  forward chan []byte
}
```

We will improve this definition later once we know a little more about what our room needs to do, but for now, this will allow us to proceed. Later, the `forward` channel is what we will use to send the incoming messages to all other clients.

You can think of channels as an in-memory thread-safe message queue where senders pass data and receivers read data in a non-blocking, thread-safe way.

In order for a client to do any work, we must define some methods that will do the actual reading and writing to and from the web socket. Adding the following code to `client.go` outside (underneath) the `client` struct will add two methods called `read` and `write` to the `client` type:

```
func (c *client) read() {
  for {
    if _, msg, err := c.socket.ReadMessage(); err == nil {
      c.room.forward <- msg
    } else {
      break
    }
  }
  c.socket.Close()
}
func (c *client) write() {
  for msg := range c.send {
    if err := c.socket.WriteMessage(websocket.TextMessage, msg);
err != nil {
      break
    }
  }
  c.socket.Close()
}
```

The `read` method allows our client to read from the socket via the `ReadMessage` method, continually sending any received messages to the `forward` channel on the `room` type. If it encounters an error (such as `'the socket has died'`), the loop will break and the socket will be closed. Similarly, the `write` method continually accepts messages from the `send` channel writing everything out of the socket via the `WriteMessage` method. If writing to the socket fails, the `for` loop is broken and the socket is closed. Build the package again to ensure everything compiles.

Modeling a room

We need a way for clients to join and leave rooms in order to ensure that the `c.room.forward <- msg` code in the preceding section actually forwards the message to all the clients. To ensure that we are not trying to access the same data at the same time, a sensible approach is to use two channels: one that will add a client to the room and another that will remove it. Let's update our `room.go` code to look like this:

```
package main

type room struct {

  // forward is a channel that holds incoming messages
  // that should be forwarded to the other clients.
  forward chan []byte
  // join is a channel for clients wishing to join the room.
  join chan *client
  // leave is a channel for clients wishing to leave the room.
  leave chan *client
  // clients holds all current clients in this room.
  clients map[*client]bool
}
```

We have added three fields: two channels and a map. The `join` and `leave` channels exist simply to allow us to safely add and remove clients from the `clients` map. If we were to access the map directly, it is possible that two Go routines running concurrently might try to modify the map at the same time resulting in corrupt memory or an unpredictable state.

Concurrency programming using idiomatic Go

Now we get to use an extremely powerful feature of Go's concurrency offerings—the `select` statement. We can use `select` statements whenever we need to synchronize or modify shared memory, or take different actions depending on the various activities within our channels.

Beneath the `room` structure, add the following `run` method that contains two of these `select` clauses:

```
func (r *room) run() {
  for {
    select {
    case client := <-r.join:
      // joining
      r.clients[client] = true
    case client := <-r.leave:
      // leaving
      delete(r.clients, client)
      close(client.send)
    case msg := <-r.forward:
      // forward message to all clients
      for client := range r.clients {
        select {
        case client.send <- msg:
          // send the message
        default:
          // failed to send
          delete(r.clients, client)
          close(client.send)
        }
      }
    }
  }
}
```

Although this might seem like a lot of code to digest, once we break it down a little, we will see that it is fairly simple, although extremely powerful. The top `for` loop indicates that this method will run forever, until the program is terminated. This might seem like a mistake, but remember, if we run this code as a Go routine, it will run in the background, which won't block the rest of our application. The preceding code will keep watching the three channels inside our room: `join`, `leave`, and `forward`. If a message is received on any of those channels, the `select` statement will run the code for that particular case. It is important to remember that it will only run one block of case code at a time. This is how we are able to synchronize to ensure that our `r.clients` map is only ever modified by one thing at a time.

If we receive a message on the `join` channel, we simply update the `r.clients` map to keep a reference of the client that has joined the room. Notice that we are setting the value to `true`. We are using the map more like a slice, but do not have to worry about shrinking the slice as clients come and go through time—setting the value to `true` is just a handy, low-memory way of storing the reference.

If we receive a message on the `leave` channel, we simply delete the `client` type from the map, and close its `send` channel. Closing a channel has special significance in Go, which becomes clear when we look at our final `select` case.

If we receive a message on the `forward` channel, we iterate over all the clients and `send` the message down each client's send channel. Then, the `write` method of our client type will pick it up and send it down the socket to the browser. If the `send` channel is closed, then we know the client is not receiving any more messages, and this is where our second `select` clause (specifically the default case) takes the action of removing the client from the room and tidying things up.

Turning a room into an HTTP handler

Now we are going to turn our `room` type into an `http.Handler` type like we did with the template handler earlier. As you will recall, to do this, we must simply add a method called `ServeHTTP` with the appropriate signature. Add the following code to the bottom of the `room.go` file:

```
const (
  socketBufferSize  = 1024
  messageBufferSize = 256
)
```

```
var upgrader = &websocket.Upgrader{ReadBufferSize:
socketBufferSize, WriteBufferSize: socketBufferSize}
func (r *room) ServeHTTP(w http.ResponseWriter, req *http.Request) {
  socket, err := upgrader.Upgrade(w, req, nil)
  if err != nil {
    log.Fatal("ServeHTTP:", err)
    return
  }
  client := &client{
    socket: socket,
    send:   make(chan []byte, messageBufferSize),
    room:   r,
  }
  r.join <- client
  defer func() { r.leave <- client }()
  go client.write()
  client.read()
}
```

The `ServeHTTP` method means a room can now act as a handler. We will implement it shortly, but first let's have a look at what is going on in this snippet of code.

In order to use web sockets, we must upgrade the HTTP connection using the `websocket.Upgrader` type, which is reusable so we need only create one. Then, when a request comes in via the `ServeHTTP` method, we get the socket by calling the `upgrader.Upgrade` method. All being well, we then create our client and pass it into the `join` channel for the current room. We also defer the leaving operation for when the client is finished, which will ensure everything is tidied up after a user goes away.

The `write` method for the client is then called as a Go routine, as indicated by the three characters at the beginning of the line `go ` (the word `go` followed by a space character). This tells Go to run the method in a different thread or goroutine.

> Compare the amount of code needed to achieve multithreading or concurrency in other languages with the three key presses that achieve it in Go, and you will see why it has become a favorite among systems developers.

Finally, we call the `read` method in the main thread, which will block operations (keeping the connection alive) until it's time to close it. Adding constants at the top of the snippet is a good practice for declaring values that would otherwise be hardcoded throughout the project. As these grow in number, you might consider putting them in a file of their own, or at least at the top of their respective files so they remain easy to read and modify.

Use helper functions to remove complexity

Our room is almost ready to use, although in order for it to be of any use, the channels and map need to be created. As it is, this could be achieved by asking the developer to use the following code to be sure to do this:

```
r := &room{
  forward: make(chan []byte),
  join:    make(chan *client),
  leave:   make(chan *client),
  clients: make(map[*client]bool),
}
```

Another, slightly more elegant, solution is to instead provide a `newRoom` function that does this for us. This removes the need for others to know about exactly what needs to be done in order for our room to be useful. Underneath the `type room struct` definition, add this function:

```
// newRoom makes a new room that is ready to go.
func newRoom() *room {
  return &room{
    forward: make(chan []byte),
    join:    make(chan *client),
    leave:   make(chan *client),
    clients: make(map[*client]bool),
  }
}
```

Now the users of our code need only call the `newRoom` function instead of the more verbose six lines of code.

Creating and using rooms

Let's update our `main` function in `main.go` to first create and then run a room for everybody to connect to:

```
func main() {
  r := newRoom()
  http.Handle("/", &templateHandler{filename: "chat.html"})
  http.Handle("/room", r)
  // get the room going
  go r.run()
  // start the web server
  if err := http.ListenAndServe(":8080", nil); err != nil {
    log.Fatal("ListenAndServe:", err)
  }
}
```

We are running the room in a separate Go routine (notice the `go` keyword again) so that the chatting operations occur in the background, allowing our main thread to run the web server. Our server is now finished and successfully built, but remains useless without clients to interact with.

Building an HTML and JavaScript chat client

In order for the users of our chat application to interact with the server and therefore other users, we need to write some client-side code that makes use of the web sockets found in modern browsers. We are already delivering HTML content via the template when users hit the root of our application, so we can enhance that.

Update the `chat.html` file in the `templates` folder with the following markup:

```
<html>
  <head>
    <title>Chat</title>
    <style>
      input { display: block; }
      ul    { list-style: none; }
    </style>
  </head>
  <body>
    <ul id="messages"></ul>
    <form id="chatbox">
      <textarea></textarea>
      <input type="submit" value="Send" />
       </form>  </body>
</html>
```

The preceding HTML will render a simple web form on the page containing a text area and a **Send** button—this is how our users will submit messages to the server. The `messages` element in the preceding code will contain the text of the chat messages so that all the users can see what is being said. Next, we need to add some JavaScript to add some functionality to our page. Underneath the `form` tag, above the closing `</body>` tag, insert the following code:

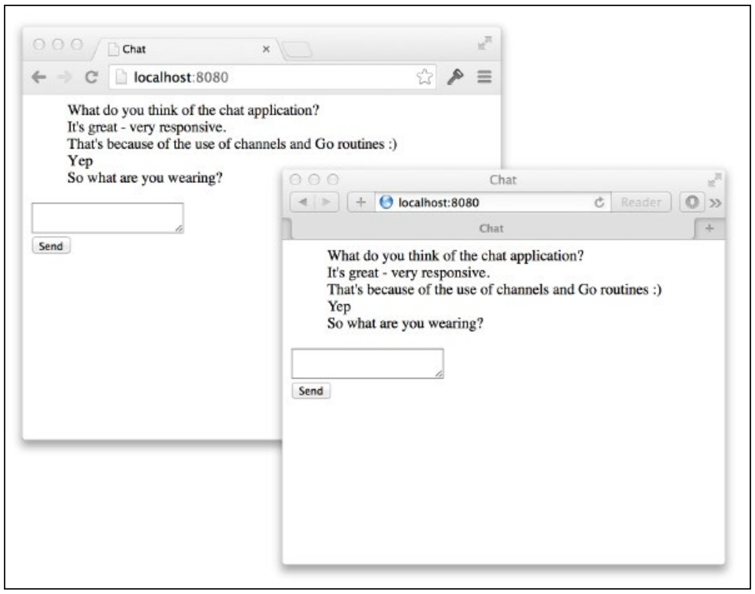
```
    <script
src="//ajax.googleapis.com/ajax/libs/jquery/1.11.1/jquery.min.js">
</script>
    <script>
      $(function(){
        var socket = null;
        var msgBox = $("#chatbox textarea");
        var messages = $("#messages");
        $("#chatbox").submit(function(){
          if (!msgBox.val()) return false;
          if (!socket) {
            alert("Error: There is no socket connection.");
            return false;
          }
          socket.send(msgBox.val());
          msgBox.val("");
          return false;
        });
        if (!window["WebSocket"]) {
          alert("Error: Your browser does not support web
sockets.")
        } else {
          socket = new WebSocket("ws://localhost:8080/room");
          socket.onclose = function() {
            alert("Connection has been closed.");
          }
          socket.onmessage = function(e) {
            messages.append($("<li>").text(e.data));
          }
        }
      });
    </script>
```

The `socket = new WebSocket("ws://localhost:8080/room")` line is where we open the socket and add event handlers for two key events: `onclose` and `onmessage`. When the socket receives a message, we use jQuery to append the message to the list element and thus present it to the user.

Submitting the HTML form triggers a call to `socket.send`, which is how we send messages to the server.

Build and run the program again to ensure the templates recompile so these changes are represented.

Navigate to `http://localhost:8080/` in two separate browsers (or two tabs of the same browser) and play with the application. You will notice that messages sent from one client appear instantly in the other clients.



What do you think of the chat application?
It's great - very responsive.
That's because of the use of channels and Go routines :)
Yep
So what are you wearing?

Getting more out of templates

Currently, we are using templates to deliver static HTML, which is nice because it gives us a clean and simple way to separate the client code from the server code. However, templates are actually much more powerful, and we are going to tweak our application to make some more realistic use of them.

The host address of our application (`:8080`) is hardcoded in two places at the moment. The first instance is in `main.go` where we start the web server:

```
if err := http.ListenAndServe(":8080", nil); err != nil {
  log.Fatal("ListenAndServe:", err)
}
```

The second time it is hardcoded in the JavaScript when we open the socket:

```
socket = new WebSocket("ws://localhost:8080/room");
```

Our chat application is pretty stubborn if it insists on only running locally on port `8080`, so we are going to use command-line flags to make it configurable and then use the injection capabilities of templates to make sure our JavaScript knows the right host.

Update your `main` function in `main.go`:

```
func main() {
  var addr = flag.String("addr", ":8080", "The addr of the
application.")
  flag.Parse() // parse the flags
  r := newRoom()
  http.Handle("/", &templateHandler{filename: "chat.html"})
  http.Handle("/room", r)
  // get the room going
  go r.run()
  // start the web server
  log.Println("Starting web server on", *addr)
  if err := http.ListenAndServe(*addr, nil); err != nil {
    log.Fatal("ListenAndServe:", err)
  }
}
```

You will need to import the `flag` package in order for this code to build. The definition for the `addr` variable sets up our flag as a string that defaults to `:8080` (with a short description of what the value is intended for). We must call `flag.Parse()` that parses the arguments and extracts the appropriate information. Then, we can reference the value of the host flag by using `*addr`.

The call to `flag.String` returns a type of `*string`, which is to say it returns the address of a string variable where the value of the flag is stored. To get the value itself (and not the address of the value), we must use the pointer indirection operator, `*`.

We also added a `log.Println` call to output the address in the terminal so we can be sure that our changes have taken effect.

We are going to modify the `templateHandler` type we wrote so that it passes the details of the request as data into the template's `Execute` method. In `main.go`, update the `ServeHTTP` function to pass the request `r` as the `data` argument to the `Execute` method:

```
func (t *templateHandler) ServeHTTP(w http.ResponseWriter, r
*http.Request) {
  t.once.Do(func() {
    t.templ =
template.Must(template.ParseFiles(filepath.Join("templates",
t.filename)))
  })
  t.templ.Execute(w, r)
}
```

This tells the template to render itself using data that can be extracted from `http.Request`, which happens to include the host address that we need.

To use the `Host` value of `http.Request`, we can then make use of the special template syntax that allows us to inject data. Update the line where we create our socket in the `chat.html` file:

```
socket = new WebSocket("ws://{{.Host}}/room");
```

The double curly braces represent an annotation and the way we tell our template source to inject data. `{{.Host}}` is essentially the equivalent of telling it to replace the annotation with the value from `request.Host` (since we passed the request `r` object in as data).

We have only scratched the surface of the power of the templates built into Go's standard library. The `text/template` package documentation is a great place to learn more about what you can achieve. You can find out more about it at `http://golang.org/pkg/text/template`.

Rebuild and run the chat program again, but this time notice that the chatting operations no longer produce an error, whichever host we specify:

```
go build -o chat
./chat -addr=":3000"
```

View the source of the page in the browser and notice that `{{.Host}}` has been replaced with the actual host of the application. Valid hosts aren't just port numbers; you can also specify the IP addresses or other hostnames—provided they are allowed in your environment, for example, `-addr="192.168.0.1:3000"`.

Tracing code to get a look under the hood

The only way we will know that our application is working is by opening two or more browsers and using our UI to send messages. In other words, we are manually testing our code. This is fine for experimental projects such as our chat application or small projects that aren't expected to grow, but if our code is to have a longer life or be worked on by more than one person, manual testing of this kind becomes a liability. We are not going to tackle **Test-driven Development** (**TDD**) for our chat program, but we should explore another useful debugging technique called **tracing**.

Tracing is a practice by which we log or print key steps in the flow of a program to make what is going on under the covers visible. In the previous section, we added a `log.Println` call to output the address that the chat program was binding to. In this section, we are going to formalize this and write our own complete tracing package.

We are going to explore TDD practices when writing our tracing code because it is a perfect example of a package that we are likely to reuse, add to, share, and hopefully, even open source.

Writing a package using TDD

Packages in Go are organized into folders, with one package per folder. It is a build error to have differing package declarations within the same folder because all sibling files are expected to contribute to a single package. Go has no concept of subpackages, which means nested packages (in nested folders) exist only for aesthetic or informational reasons but do not inherit any functionality or visibility from super packages. In our chat application, all of our files contributed to the `main` package because we wanted to build an executable tool. Our tracing package will never be run directly, so it can and should use a different package name. We will also need to think about the **Application Programming Interface** (**API**) of our package, considering how to model a package so that it remains as extensible and flexible as possible for users. This includes the fields, functions, methods, and types that should be exported (visible to the user) and remain hidden for simplicity's sake.

Go uses capitalization of names to denote which items are exported such that names that begin with a capital letter (for example, `Tracer`) are visible to users of a package, and names that begin with a lowercase letter (for example, `templateHandler`) are hidden or private.

Create a new folder called `trace`, which will be the name of our tracing package, alongside the `chat` folder.

Before we jump into the code, let's agree on some design goals for our package by which we can measure success:

- The package should be easy to use
- Unit tests should cover the functionality
- Users should have the flexibility to replace the tracer with their own implementation

Interfaces

Interfaces in Go are an extremely powerful language feature that allow us to define an API without being strict or specific on the implementation details. Wherever possible, describing the basic building blocks of your packages using interfaces usually ends up paying dividends down the road, and this is where we will start for our tracing package.

Create a new file called `tracer.go` inside the `trace` folder and write the following code:

```
package trace
// Tracer is the interface that describes an object capable of
// tracing events throughout code.
type Tracer interface {
  Trace(...interface{})
}
```

The first thing to notice is that we have defined our package as `trace`.

While it is a good practice to have the folder name match the package name, Go tools do not enforce it, which means you are free to name them differently if it makes sense. Remember, when people import your package, they will type the name of the folder, and if suddenly a package with a different name is imported, it could get confusing.

Our `Tracer` type (the capital `T` means we intend this to be a publicly visible type) is an interface that describes a single method called `Trace`. The `...interface{}` argument type states that our `Trace` method will accept zero or more arguments of any type. You might think that this is redundant since the method should just take a single string (we want to just trace out some string of characters, don't we?). However, consider functions such as `fmt.Sprint` and `log.Fatal`, both of which follow a pattern littered through to Go's standard library that provides a helpful shortcut when trying to communicate multiple things in one go. Wherever possible, we should follow such patterns and practices because we want our own APIs to be familiar and clear to the Go community.

Unit tests

We promised ourselves we would follow test-driven practices, but interfaces are simply definitions that do not provide any implementation and so cannot be directly tested. But we are about to write a real implementation of a `Tracer` method, and we will indeed write the tests first.

Create a new file called `tracer_test.go` in the `trace` folder and insert the following scaffold code:

```
package trace
import (
  "testing"
)
func TestNew(t *testing.T) {
  t.Error("We haven't written our test yet")
}
```

Testing was built into the Go tool chain from the very beginning, making writing automatable tests a first-class citizen. The test code lives alongside the production code in files suffixed with `_test.go`. The Go tools will treat any function that starts with `Test` (taking a single `*testing.T` argument) as a unit test, and it will be executed when we run our tests. To run them for this package, navigate to the `trace` folder in a terminal and do the following:

```
go test
```

You will see that our tests fail because of our call to `t.Error` in the body of our `TestNew` function:

```
--- FAIL: TestNew (0.00 seconds)
```

```
    tracer_test.go:8: We haven't written our test yet
FAIL
exit status 1
FAIL  trace  0.011s
```

Clearing the terminal before each test run is a great way to make sure you aren't confusing previous runs with the most recent one. On Windows, you can use the `cls` command; on Unix machines, the `clear` command does the same thing.

Obviously, we haven't properly written our test and we don't expect it to pass yet, so let's update the `TestNew` function:

```
func TestNew(t *testing.T) {
  var buf bytes.Buffer
  tracer := New(&buf)
  if tracer == nil {
    t.Error("Return from New should not be nil")
  } else {
    tracer.Trace("Hello trace package.")
    if buf.String() != "Hello trace package.\n" {
      t.Errorf("Trace should not write '%s'.", buf.String())
    }
  }
}
```

Most packages throughout the book are available from the Go standard library, so you can add an `import` statement for the appropriate package in order to access the package. Others are external, and that's when you need to use `go get` to download them before they can be imported. For this case, you'll need to add `import "bytes"` to the top of the file.

We have started designing our API by becoming the first user of it. We want to be able to capture the output of our tracer in a `bytes.Buffer` so that we can then ensure that the string in the buffer matches the expected value. If it does not, a call to `t.Errorf` will fail the test. Before that, we check to make sure the return from a made-up `New` function is not `nil`; again, if it is, the test will fail because of the call to `t.Error`.

Red-green testing

Running `go test` now actually produces an error; it complains that there is no `New` function. We haven't made a mistake here; we are following a practice known as red-green testing. Red-green testing proposes that we first write a unit test, see it fail (or produce an error), write the minimum amount of code possible to make that test pass, and rinse and repeat it again. The key point here being that we want to make sure the code we add is actually doing something as well as ensuring that the test code we write is testing something meaningful.

Consider a meaningless test for a minute:

```
if true == true {
  t.Error("True should be true")
}
```

It is logically impossible for true to not be true (if true ever equals false, it's time to get a new computer), and so our test is pointless. If a test or claim cannot fail, there is no value whatsoever to be found in it.

Replacing `true` with a variable that you expect to be set to `true` under certain conditions would mean that such a test can indeed fail (like when the code being tested is misbehaving)—at this point, you have a meaningful test that is worth contributing to the code base.

You can treat the output of `go test` like a to-do list, solving only one problem at a time. Right now, the complaint about the missing `New` function is all we will address. In the `tracer.go` file, let's add the minimum amount of code possible to progress with things; add the following snippet underneath the interface type definition:

```
func New() {}
```

Running `go test` now shows us that things have indeed progressed, albeit not very far. We now have two errors:

```
./tracer_test.go:11: too many arguments in call to New
./tracer_test.go:11: New(&buf) used as value
```

The first error tells us that we are passing arguments to our `New` function, but the `New` function doesn't accept any. The second error says that we are using the return of the `New` function as a value, but that the `New` function doesn't return anything. You might have seen this coming, and indeed as you gain more experience writing test-driven code, you will most likely jump over such trivial details. However, to properly illustrate the method, we are going to be pedantic for a while. Let's address the first error by updating our `New` function to take in the expected argument:

```
func New(w io.Writer) {}
```

We are taking an argument that satisfies the `io.Writer` interface, which means that the specified object must have a suitable `Write` method.

> Using existing interfaces, especially ones found in the Go standard library, is an extremely powerful and often necessary way to ensure that your code is as flexible and elegant as possible.

Accepting `io.Writer` means that the user can decide where the tracing output will be written. This output could be the standard output, a file, network socket, `bytes.Buffer` as in our test case, or even some custom-made object, provided it implements the `Write` method of the `io.Writer` interface

Running `go test` again shows us that we have resolved the first error and we only need add a return type in order to progress past our second error:

```
func New(w io.Writer) Tracer {}
```

We are stating that our `New` function will return a `Tracer`, but we do not return anything, which `go test` happily complains about:

```
./tracer.go:13: missing return at end of function
```

Fixing this is easy; we can just return `nil` from the `New` function:

```
func New(w io.Writer) Tracer {
  return nil
}
```

Of course, our test code has asserted that the return should not be `nil`, so `go test` now gives us a failure message:

```
tracer_test.go:14: Return from New should not be nil
```

> You can see how a strict adherence to the red-green principle can get a little tedious, but it is vital that we do not jump too far ahead. If we were to write a lot of implementation code in one go, we will very likely have code that is not covered by a unit test.
>
> The ever-thoughtful core team has even solved this problem for us by providing code coverage statistics which we can generate by running the following command:
>
> ```
> go test -cover
> ```
>
> Provided that all tests pass, adding the `-cover` flag will tell us how much of our code was touched during the execution of the tests. Obviously, the closer we get to 100 percent the better.

Implementing the interface

To satisfy this test, we need something that we can properly return from the `New` method because `Tracer` is only an interface and we have to return something real. Let's add an implementation of a tracer to our `tracer.go` file:

```
type tracer struct {
  out io.Writer
}

func (t *tracer) Trace(a ...interface{}) {}
```

Our implementation is extremely simple; the `tracer` type has an `io.Writer` field called `out` which is where we will write the trace output to. And the `Trace` method exactly matches the method required by the `Tracer` interface, although it doesn't do anything yet.

Now we can finally fix the `New` method:

```
func New(w io.Writer) Tracer {
  return &tracer{out: w}
}
```

Running `go test` again shows us that our expectation was not met because nothing was written during our call to `Trace`:

```
tracer_test.go:18: Trace should not write ''.
```

Let's update our `Trace` method to write the blended arguments to the specified `io.Writer` field:

```
func (t *tracer) Trace(a ...interface{}) {
  t.out.Write([]byte(fmt.Sprint(a...)))
  t.out.Write([]byte("\n"))
}
```

When the `Trace` method is called, we call `Write` on the `io.Writer` stored in the `out` field and use `fmt.Sprint` to format the `a` arguments. We convert the string return type from `fmt.Sprint` to `string` and then to `[]byte` because that is what is expected by the `io.Writer` interface.

Have we finally satisfied our test?

```
go test -cover

PASS
```

```
coverage: 100.0% of statements
ok    trace  0.011s
```

Congratulations! We have successfully passed our test and have `100.0%` test coverage. Once we have finished our glass of champagne, we can take a minute to consider something very interesting about our implementation.

Unexported types being returned to users

The `tracer` struct type we wrote is unexported because it begins with a lowercase `t`, so how is it that we are able to return it from the exported `New` function? After all, doesn't the user receive the returned object? This is perfectly acceptable and valid Go code; the user will only ever see an object that satisfies the `Tracer` interface and will never even know about our private `tracer` type. Since they only ever interact with the interface anyway, it wouldn't matter if our `tracer` implementation exposed other methods or fields; they would never be seen. This allows us to keep the public API of our package clean and simple.

This hidden implementation technique is used throughout the Go standard library, for example, the `ioutil.NopCloser` method is a function that turns a normal `io.Reader` into `io.ReadCloser` whereas the `Close` method does nothing (used for when `io.Reader` objects that don't need to be closed are passed into functions that require `io.ReadCloser` types). The method returns `io.ReadCloser` as far as the user is concerned, but under the hood, there is a secret `nopCloser` type hiding the implementation details.

To see this for yourself, browse the Go standard library source code at `http://golang.org/src/pkg/io/ioutil/ioutil.go` and search for the `nopCloser` struct.

Using our new trace package

Now that we have completed the first version of our `trace` package, we can use it in our chat application in order to better understand what is going on when users send messages through the user interface.

In `room.go`, let's import our new package and make some calls to the `Trace` method. The path to the `trace` package we just wrote will depend on your `GOPATH` environment variable because the import path is relative to the `$GOPATH/src` folder. So if you create your `trace` package in `$GOPATH/src/mycode/trace`, then you would need to import `mycode/trace`.

Update the `room` type and the `run()` method like this:

```
type room struct {
  // forward is a channel that holds incoming messages
  // that should be forwarded to the other clients.
  forward chan []byte
  // join is a channel for clients wishing to join the room.
  join chan *client
  // leave is a channel for clients wishing to leave the room.
  leave chan *client
  // clients holds all current clients in this room.
  clients map[*client]bool
  // tracer will receive trace information of activity
  // in the room.
  tracer trace.Tracer
}
func (r *room) run() {
  for {
    select {
    case client := <-r.join:
      // joining
      r.clients[client] = true
      r.tracer.Trace("New client joined")
    case client := <-r.leave:
      // leaving
      delete(r.clients, client)
      close(client.send)
      r.tracer.Trace("Client left")
    case msg := <-r.forward:
      r.tracer.Trace("Message received: ", string(msg))
      // forward message to all clients
      for client := range r.clients {
        select {
        case client.send <- msg:
          // send the message
          r.tracer.Trace(" -- sent to client")
        default:
          // failed to send
          delete(r.clients, client)
          close(client.send)
          r.tracer.Trace(" -- failed to send, cleaned up client")
        }
      }
    }
  }
}
```

We added a `trace.Tracer` field to our `room` type and then made periodic calls to the `Trace` method peppered throughout the code. If we run our program and try to send messages, you'll notice that the application panics because the `tracer` field is `nil`. We can remedy this for now by making sure we create and assign an appropriate object when we create our `room` type. Update the `main.go` file to do this:

```
r := newRoom()
r.tracer = trace.New(os.Stdout)
```

We are using our `New` method to create an object that will send the output to the `os.Stdout` standard output pipe (this is a technical way of saying we want it to print the output to our terminal).

Now rebuild and run the program and use two browsers to play with the application, and notice that the terminal now has some interesting trace information for us:

```
New client joined
New client joined
Message received: Hello Chat
 -- sent to client
 -- sent to client
Message received: Good morning :)
 -- sent to client
 -- sent to client
Client left
Client left
```

Now we are able to use the debug information to get an insight into what the application is doing, which will assist us when developing and supporting our project.

Making tracing optional

Once the application is released, the sort of tracing information we are generating will be pretty useless if it's just printed out to some terminal somewhere, or even worse, if it creates a lot of noise for our systems administrators. Also, remember that when we don't set a tracer for our `room` type, our code panics, which isn't a very user-friendly situation. To resolve these two issues, we are going to enhance our `trace` package with a `trace.Off()` method that will return an object that satisfies the `Tracer` interface but will not do anything when the `Trace` method is called.

Let's add a test that calls the `Off` function to get a silent tracer before making a call to `Trace` to ensure the code doesn't panic. Since the tracing won't happen, that's all we can do in our test code. Add the following test function to the `tracer_test.go` file:

```
func TestOff(t *testing.T) {
  var silentTracer Tracer = Off()
  silentTracer.Trace("something")
}
```

To make it pass, add the following code to the `tracer.go` file:

```
type nilTracer struct{}
func (t *nilTracer) Trace(a ...interface{}) {}
// Off creates a Tracer that will ignore calls to Trace.
func Off() Tracer {
  return &nilTracer{}
}
```

Our `nilTracer` struct has defined a `Trace` method that does nothing, and a call to the `Off()` method will create a new `nilTracer` struct and return it. Notice that our `nilTracer` struct differs from our `tracer` struct in that it doesn't take an `io.Writer`; it doesn't need one because it isn't going to write anything.

Now let's solve our second problem by updating our `newRoom` method in the `room.go` file:

```
func newRoom() *room {
  return &room{
    forward: make(chan []byte),
    join:    make(chan *client),
    leave:   make(chan *client),
    clients: make(map[*client]bool),
    tracer:  trace.Off(),
  }
}
```

By default, our `room` type will be created with a `nilTracer` struct and any calls to `Trace` will just be ignored. You can try this out by removing the `r.tracer = trace.New(os.Stdout)` line from the `main.go` file: notice that nothing gets written to the terminal when you use the application and there is no panic.

Clean package APIs

A quick glance at the API (in this context, the exposed variables, methods, and types) for our `trace` package highlights that a simple and obvious design has emerged:

- The `New()` method
- The `Off()` method
- The `Tracer` interface

I would be very confident to give this package to a Go programmer without any documentation or guidelines, and I'm pretty sure they would know what do to with it.

In Go, adding documentation is as simple as adding comments to the line before each item. The blog post on the subject is a worthwhile read (`http://blog.golang.org/godoc-documenting-go-code`), where you can see a copy of the hosted source code for `tracer.go` that is an example of how you might annotate the `trace` package. For more information, refer to `github.com/matryer/goblueprints/blob/master/chapter1/trace/tracer.go`.

Summary

In this chapter, we developed a complete concurrent chat application and our own simple package to trace the flow of our programs to help us better understand what is going on under the hood.

We used the `net/http` package to quickly build what turned out to be a very powerful concurrent HTTP web server. In one particular case, we then upgraded the connection to open a web socket between the client and server. This means that we can easily and quickly communicate messages to the user's web browser without having to write messy polling code. We explored how templates are useful to separate the code from the content as well as to allow us to inject data into our template source, which let us make the host address configurable. Command-line flags helped us give simple configuration control to the people hosting our application while also letting us specify sensible defaults.

Our chat application made use of Go's powerful concurrency capabilities that allowed us to write clear *threaded* code in just a few lines of idiomatic Go. By controlling the coming and going of clients through channels, we were able to set synchronization points in our code that prevented us from corrupting memory by attempting to modify the same objects at the same time.

We learned how interfaces such as `http.Handler` and our own `trace.Tracer` allow us to provide disparate implementations without having to touch the code that makes use of them, and in some cases, without having to expose even the name of the implementation to our users. We saw how just by adding a `ServeHTTP` method to our `room` type, we turned our custom room concept into a valid HTTP handler object, which managed our web socket connections.

We aren't actually very far away from being able to properly release our application, except for one major oversight: you cannot see who sent each message. We have no concept of users or even user names, and for a real chat application, this is not acceptable.

In the next chapter, we will add the names of the people responding to their messages in order to make them feel like they are having a real conversation with other humans.

2 Adding Authentication

The chat application we built in the previous chapter focused on high-performance transmission of messages from the clients to the server and back again, but our users have no way of knowing who they are talking to. One solution to this problem is building of some kind of signup and login functionality and letting our users create accounts and authenticate themselves before they can open the chat page.

Whenever we are about to build something from scratch, we must ask ourselves how others have solved this problem before (it is extremely rare to encounter genuinely original problems), and whether any open solutions or standards already exist that we can make use of. Authorization and authentication are hardly new problems, especially in the world of the Web, with many different protocols out there to choose from. So how do we decide on the best option to pursue? As always, we must look at this question from the point of view of the user.

A lot of websites these days allow you to sign in using your accounts existing elsewhere on a variety of social media or community websites. This saves users the tedious job of entering all their account information over and over again as they decide to try out different products and services. It also has a positive effect on the conversion rates for new sites.

In this chapter, we will enhance our chat codebase to add authentication, which will allow our users to sign in using Google, Facebook, or GitHub and you'll see how easy it is to add other sign-in portals too. In order to join the chat, users must first sign in. Following this, we will use the authorized data to augment our user experience so everyone knows who is in the room, and who said what.

In this chapter, you will learn to:

- Use the decorator pattern to wrap `http.Handler` types to add additional functionality to handlers
- Serve HTTP endpoints with dynamic paths

- Use the Gomniauth open source project to access authentication services
- Get and set cookies using the `http` package
- Encode objects as Base64 and back to normal again
- Send and receive JSON data over a web socket
- Give different types of data to templates
- Work with channels of your own types

Handlers all the way down

For our chat application, we implemented our own `http.Handler` type in order to easily compile, execute, and deliver HTML content to browsers. Since this is a very simple but powerful interface, we are going to continue to use it wherever possible when adding functionality to our HTTP processing.

In order to determine whether a user is authenticated, we will create an authentication wrapper handler that performs the check, and passes execution on to the inner handler only if the user is authenticated.

Our wrapper handler will satisfy the same `http.Handler` interface as the object inside it, allowing us to wrap any valid handler. In fact, even the authentication handler we are about to write could be later encapsulated inside a similar wrapper if needed.

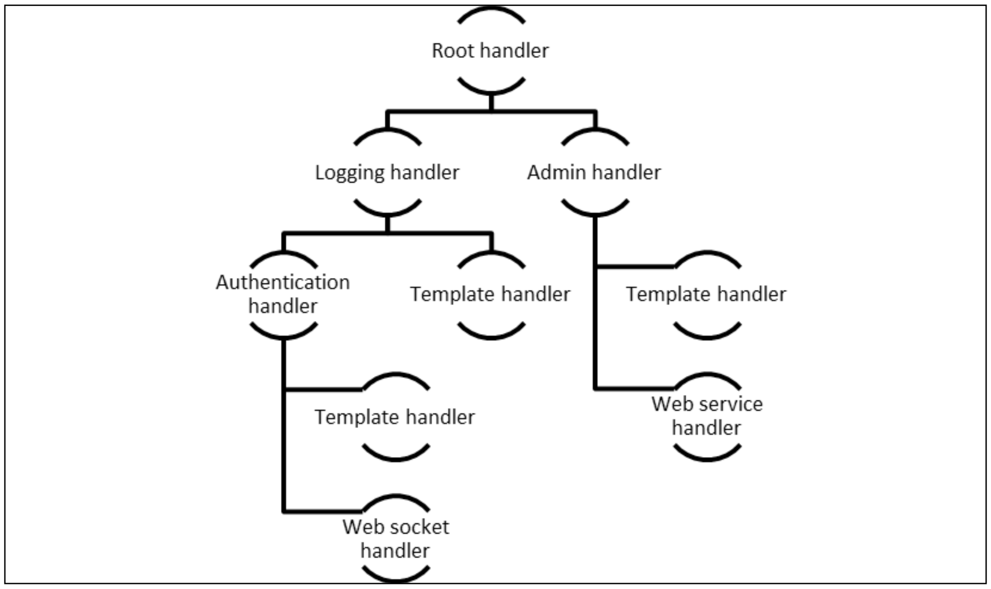


Diagram of a chaining pattern when applied to HTTP handlers

The preceding figure shows how this pattern could be applied in a more complicated HTTP handler scenario. Each object implements the `http.Handler` interface, which means that object could be passed into the `http.Handle` method to directly handle a request, or it can be given to another object, which adds some kind of extra functionality. The `Logging` handler might write to a logfile before and after the `ServeHTTP` method is called on the inner handler. Because the inner handler is just another `http.Handler`, any other handler can be wrapped in (or decorated with) the `Logging` handler.

It is also common for an object to contain logic that decides which inner handler should be executed. For example, our authentication handler will either pass the execution to the wrapped handler, or handle the request itself by issuing a redirect to the browser.

That's plenty of theory for now; let's write some code. Create a new file called `auth.go` in the `chat` folder:

```
package main
import (
  "net/http"
)
type authHandler struct {
  next http.Handler
}
func (h *authHandler) ServeHTTP(w http.ResponseWriter, r
*http.Request) {
  if _, err := r.Cookie("auth"); err == http.ErrNoCookie {
    // not authenticated
    w.Header().Set("Location", "/login")
    w.WriteHeader(http.StatusTemporaryRedirect)
  } else if err != nil {
    // some other error
    panic(err.Error())
  } else {
    // success - call the next handler
    h.next.ServeHTTP(w, r)
  }
}
func MustAuth(handler http.Handler) http.Handler {
  return &authHandler{next: handler}
}
```

The `authHandler` type not only implements the `ServeHTTP` method (which satisfies the `http.Handler` interface) but also stores (wraps) `http.Handler` in the next field. Our `MustAuth` helper function simply creates `authHandler` that wraps any other `http.Handler`. Let's tweak the following root mapping line:

```
http.Handle("/", &templateHandler{filename: "chat.html"})
```

Let's change the first argument to make it explicit about the page meant for chatting. Next, let's use the `MustAuth` function to wrap `templateHandler` for the second argument:

```
http.Handle("/chat", MustAuth(&templateHandler{filename:
"chat.html"}))
```

Wrapping `templateHandler` with the `MustAuth` function will cause execution to run first through our `authHandler`, and only to `templateHandler` if the request is authenticated.

The `ServeHTTP` method in our `authHandler` will look for a special cookie called `auth`, and use the `Header` and `WriteHeader` methods on `http.ResponseWriter` to redirect the user to a login page if the cookie is missing.

Build and run the chat application and try to hit `http://localhost:8080/chat`:

```
go build -o chat
./chat -host=":8080"
```

You need to delete your cookies to clear out previous auth tokens, or any other cookies that might be left over from other development projects served through localhost.

If you look in the address bar of your browser, you will notice that you are immediately redirected to the `/login` page. Since we cannot handle that path yet, you'll just get a **404 page not found** error.

Making a pretty social sign-in page

So far we haven't paid much attention to making our application look nice, after all this book is about Go and not user-interface development. However, there is no excuse for building ugly apps, and so we will build a social sign-in page that is as pretty as it is functional.

Bootstrap is a frontend framework used to develop responsive projects on the Web. It provides CSS and JavaScript code that solve many user-interface problems in a consistent and good-looking way. While sites built using Bootstrap all tend to look the same (although there are plenty of ways in which the UI can be customized), it is a great choice for early versions of apps, or for developers who don't have access to designers.

> If you build your application using the semantic standards set forth by Bootstrap, it becomes easy for you to make a Bootstrap theme for your site or application and you know it will slot right into your code.

We will use the version of Bootstrap hosted on a CDN so we don't have to worry about downloading and serving our own version through our chat application. This means that in order to render our pages properly, we will need an active Internet connection, even during development.

> If you prefer to download and host your own copy of Bootstrap, you can do so. Keep the files in an `assets` folder and add the following call to your `main` function (it uses `http.Handle` to serve the assets via your application):
>
> ```
> http.Handle("/assets/", http.StripPrefix("/assets",
> http.FileServer(http.Dir("/path/to/assets/"))))
> ```
>
> Notice how the `http.StripPrefix` and `http.FileServer` functions return objects that satisfy the `http.Handler` interface as per the decorator pattern that we implement with our `MustAuth` helper function.

In `main.go`, let's add an endpoint for the login page:

```
http.Handle("/chat", MustAuth(&templateHandler{filename:
"chat.html"}))
http.Handle("/login", &templateHandler{filename: "login.html"})
http.Handle("/room", r)
```

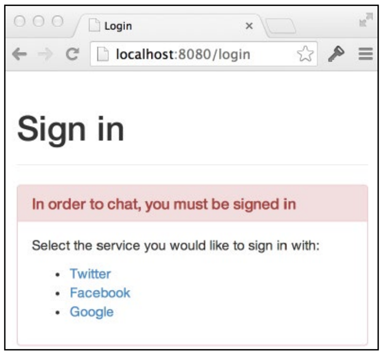
Obviously, we do not want to use the `MustAuth` method for our login page because it will cause an infinite redirection loop.

Create a new file called `login.html` inside our `templates` folder, and insert the following HTML code:

```
<html>
  <head>
    <title>Login</title>
    <link rel="stylesheet"
```

```
        href="//netdna.bootstrapcdn.com/bootstrap/3.1.1/css/
        bootstrap.min.css">
    </head>
    <body>
      <div class="container">
        <div class="page-header">
          <h1>Sign in</h1>
        </div>
        <div class="panel panel-danger">
          <div class="panel-heading">
            <h3 class="panel-title">In order to chat, you must be
              signed in</h3>
          </div>
          <div class="panel-body">
            <p>Select the service you would like to sign in
              with:</p>
            <ul>
              <li>
                <a href="/auth/login/facebook">Facebook</a>
              </li>
              <li>
                <a href="/auth/login/github">GitHub</a>
              </li>
              <li>
                <a href="/auth/login/google">Google</a>
              </li>
            </ul>
          </div>
        </div>
      </div>
    </body>
  </html>
```

Restart the web server and navigate to `http://localhost:8080/login`. You will notice that it now displays our sign-in page:



Sign in

In order to chat, you must be signed in

Select the service you would like to sign in with:

- Twitter
- Facebook
- Google

Endpoints with dynamic paths

Pattern matching for the `http` package in the Go standard library isn't the most comprehensive and fully featured implementation out there. For example, Ruby on Rails makes it much easier to have dynamic segments inside the path:

```
"auth/:action/:provider_name"
```

This then provides a data map (or dictionary) containing the values that the framework automatically extracted from the matched path. So if you visit `auth/login/google`, then `params[:provider_name]` would equal `google`, and `params[:action]` would equal `login`.

The most the `http` package lets us specify by default is a path prefix, which we can do by leaving a trailing slash at the end of the pattern:

```
"auth/"
```

We would then have to manually parse the remaining segments to extract the appropriate data. This is acceptable for relatively simple cases, which suits our needs for the time being since we only need to handle a few different paths such as:

- `/auth/login/google`
- `/auth/login/facebook`
- `/auth/callback/google`
- `/auth/callback/facebook`

If you need to handle more advanced routing situations, you might want to consider using dedicated packages such as Goweb, Pat, Routes, or mux. For extremely simple cases such as ours, the built-in capabilities will do.

We are going to create a new handler that powers our login process. In `auth.go`, add the following `loginHandler` code:

```
// loginHandler handles the third-party login process.
// format: /auth/{action}/{provider}
func loginHandler(w http.ResponseWriter, r *http.Request) {
  segs := strings.Split(r.URL.Path, "/")
  action := segs[2]
  provider := segs[3]
  switch action {
  case "login":
    log.Println("TODO handle login for", provider)
```

```
      default:
         w.WriteHeader(http.StatusNotFound)
         fmt.Fprintf(w, "Auth action %s not supported", action)
      }
   }
```

In the preceding code, we break the path into segments using `strings.Split` before pulling out the values for `action` and `provider`. If the action value is known, we will run the specific code; otherwise, we will write out an error message and return an `http.StatusNotFound` status code (which in the language of HTTP status code, is a `404` code).

> We will not bullet-proof our code right now but it's worth noticing that if someone hits `loginHandler` with too few segments, our code will panic because it expects `segs[2]` and `segs[3]` to exist.
>
> For extra credit, see whether you can protect against this and return a nice error message instead of a panic if someone hits `/auth/nonsense`.

Our `loginHandler` is only a function and not an object that implements the `http.Handler` interface. This is because, unlike other handlers, we don't need it to store any state. The Go standard library supports this, so we can use the `http.HandleFunc` function to map it in a way similar to how we used `http.Handle` earlier. In `main.go`, update the handlers:

```
   http.Handle("/chat", MustAuth(&templateHandler{filename:
   "chat.html"}))
   http.Handle("/login", &templateHandler{filename: "login.html"})
   http.HandleFunc("/auth/", loginHandler)
   http.Handle("/room", r)
```

Rebuild and run the chat application:

```
go build -o chat
./chat -host=":8080"
```

Hit the following URLs and notice the output logged in the terminal:

- `http://localhost:8080/auth/login/google` outputs `TODO handle login for google`
- `http://localhost:8080/auth/login/facebook` outputs `TODO handle login for facebook`

We have successfully implemented a dynamic path-matching mechanism that so far just prints out to-do messages; next we need to write code that integrates with the authentication services.

OAuth2

OAuth2 is an open authentication and authorization standard designed to allow resource owners to give clients delegated access to private data (such as wall posts or tweets) via an access token exchange handshake. Even if you do not wish to access the private data, OAuth2 is a great option that allows people to sign in using their existing credentials, without exposing those credentials to a third-party site. In this case, we are the third party and we want to allow our users to sign in using services that support OAuth2.

From a user's point of view, the OAuth2 flow is:

1. A user selects provider with whom they wish to sign in to the client app.
2. The user is redirected to the provider's website (with a URL that includes the client app ID) where they are asked to give permission to the client app.
3. The user signs in from the OAuth2 service provider and accepts the permissions requested by the third-party application.
4. The user is redirected back to the client app with a request code.
5. In the background, the client app sends the grant code to the provider, who sends back an auth token.
6. The client app uses the access token to make authorized requests to the provider, such as to get user information or wall posts.

To avoid reinventing the wheel, we will look at a few open source projects that have already solved this problem for us.

Open source OAuth2 packages

Andrew Gerrand has been working on the core Go team since February 2010, that is two years before Go 1.0 was officially released. His `goauth2` package (see `https://code.google.com/p/goauth2/`) is an elegant implementation of the OAuth2 protocol written entirely in Go.

Andrew's project inspired Gomniauth (see `https://github.com/stretchr/gomniauth`). An open source Go alternative to Ruby's `omniauth` project, Gomniauth provides a unified solution to access different OAuth2 services. In the future, when OAuth3 (or whatever next-generation authentication protocol it is) comes out, in theory, Gomniauth could take on the pain of implementing the details, leaving the user code untouched.

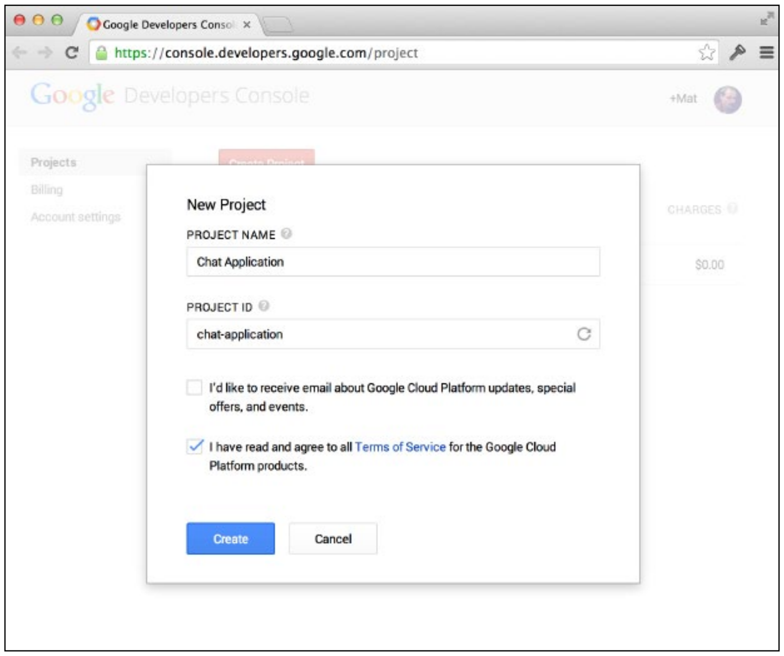
For our application, we will use Gomniauth to access OAuth services provided by Google, Facebook, and GitHub, so make sure you have it installed by running the following command:

```
go get github.com/stretchr/gomniauth
```

Some of the project dependencies of Gomniauth are kept in Bazaar repositories, so you'll need to head over to `http://wiki.bazaar.canonical.com` to download them.

Tell the authentication providers about your app

Before we ask an authentication provider to help our users sign in, we must tell them about our application. Most providers have some kind of web tool or console where you can create applications to kick-start the process. Here's one from Google:


Google Developers Console
https://console.developers.google.com/project
Google Developers Console
+Mat
Projects
Billing
Account settings
CHARGES
$0.00
Create
Cancel

New Project

PROJECT NAME: Chat Application

PROJECT ID: chat-application

☐ I'd like to receive email about Google Cloud Platform updates, special offers, and events.

✓ I have read and agree to all Terms of Service for the Google Cloud Platform products.

In order to identify the client application, we need to create a client ID and secret. Despite the fact that OAuth2 is an open standard, each provider has their own language and mechanism to set things up, so you will most likely have to play around with the user interface or the documentation to figure it out in each case.

At the time of writing this, in **Google Developer Console**, you navigate to **APIs & auth | Credentials** and click on the **Create new Client ID** button.

In most cases, for added security, you have to be explicit about the host URLs from where requests will come. For now, since we're hosting our app locally on `localhost:8080`, you should use that. You will also be asked for a redirect URI that is the endpoint in our chat application and to which the user will be redirected after successfully signing in. The callback will be another action on our `loginHandler`, so the redirection URL for the Google client will be `http://localhost:8080/auth/callback/google`.

Once you finish the authentication process for the providers you want to support, you will be given a client ID and secret for each provider. Make a note of these, because we will need them when we set up the providers in our chat application.

If we host our application on a real domain, we have to create new client IDs and secrets, or update the appropriate URL fields on our authentication providers to ensure that they point to the right place. Either way, it's not bad practice to have a different set of development and production keys for security.

Implementing external logging in

In order to make use of the projects, clients, or accounts that we created on the authentication provider sites, we have to tell Gomniauth which providers we want to use, and how we will interact with them. We do this by calling the `WithProviders` function on the primary Gomniauth package. Add the following code snippet to `main.go` (just underneath the `flag.Parse()` line towards the top of the `main` function):

```
// set up gomniauth
gomniauth.SetSecurityKey("some long key")
gomniauth.WithProviders(
  facebook.New("key", "secret",
    "http://localhost:8080/auth/callback/facebook"),
  github.New("key", "secret",
    "http://localhost:8080/auth/callback/github"),
  google.New("key", "secret",
    "http://localhost:8080/auth/callback/google"),
)
```

You should replace the `key` and `secret` placeholders with the actual values you noted down earlier. The third argument represents the callback URL that should match the ones you provided when creating your clients on the provider's website. Notice the second path segment is `callback`; while we haven't implemented this yet, this is where we handle the response from the authentication process.

As usual, you will need to ensure all the appropriate packages are imported:

```
import (
  "github.com/stretchr/gomniauth/providers/facebook"
  "github.com/stretchr/gomniauth/providers/github"
  "github.com/stretchr/gomniauth/providers/google"
)
```

Gomniauth requires the `SetSecurityKey` call because it sends state data between the client and server along with a signature checksum, which ensures that the state values haven't been tempered with while transmitting. The security key is used when creating the hash in a way that it is almost impossible to recreate the same hash without knowing the exact security key. You should replace `some long key` with a security hash or phrase of your choice.

Logging in

Now that we have configured Gomniauth, we need to redirect users to the provider's authentication page when they land on our `/auth/login/{provider}` path. We just have to update our `loginHandler` function in `auth.go`:

```
func loginHandler(w http.ResponseWriter, r *http.Request) {
  segs := strings.Split(r.URL.Path, "/")
  action := segs[2]
  provider := segs[3]
  switch action {
  case "login":
    provider, err := gomniauth.Provider(provider)
    if err != nil {
      log.Fatalln("Error when trying to get provider", provider,
"-", err)
    }
    loginUrl, err := provider.GetBeginAuthURL(nil, nil)
    if err != nil {
      log.Fatalln("Error when trying to GetBeginAuthURL for",
provider, "-", err)
    }
    w.Header().Set("Location",loginUrl)
    w.WriteHeader(http.StatusTemporaryRedirect)
  default:
    w.WriteHeader(http.StatusNotFound)
    fmt.Fprintf(w, "Auth action %s not supported", action)
  }
}
```

We do two main things here. First, we use the `gomniauth.Provider` function to get the provider object that matches the object specified in the URL (such as `google` or `github`). Then we use the `GetBeginAuthURL` method to get the location where we must send users in order to start the authentication process.

The `GetBeginAuthURL(nil, nil)` arguments are for the state and options respectively, which we are not going to use for our chat application.

The first argument is a state map of data that is encoded, and signed and sent to the authentication provider. The provider doesn't do anything with the state, it just sends it back to our callback endpoint. This is useful if, for example, we want to redirect the user back to the original page they were trying to access before the authentication process intervened. For our purpose, we have only the `/chat` endpoint, so we don't need to worry about sending any state.

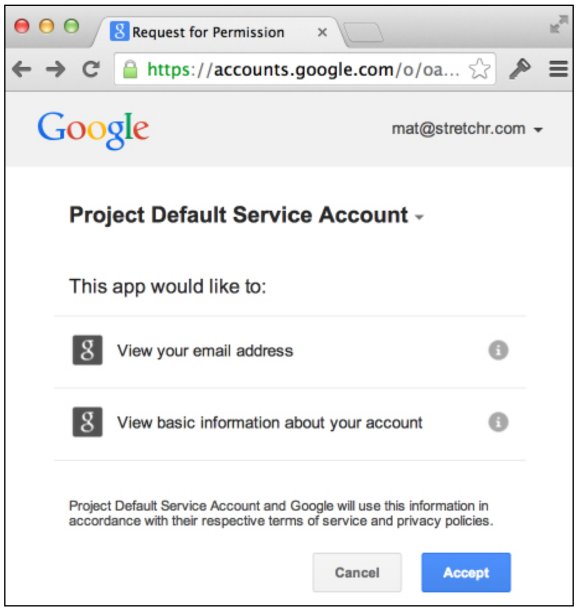
The second argument is a map of additional options that will be sent to the authentication provider, which somehow modifies the behavior of the authentication process. For example, you can specify your own `scope` parameter, which allows you to make a request for permission to access additional information from the provider. For more information about the available options, search for OAuth2 on the Internet or read the documentation for each provider, as these values differ from service to service.

If our code gets no error from the `GetBeginAuthURL` call, we simply redirect the user's browser to the returned URL.

Rebuild and run the chat application:

```
go build -o chat
./chat -host=":8080"
```

Open the main chat page by accessing `http://localhost:8080/chat`. As we aren't logged in yet, we are redirected to our sign-in page. Click on the Google option to sign in using your Google account, and you will notice that you are presented with a Google-specific sign-in page (if you are not already signed in to Google). Once you are signed in, you will be presented with a page asking you to give permission for our chat application before you can view basic information about your account:



Project Default Service Account

This app would like to:

View your email address

View basic information about your account

Project Default Service Account and Google will use this information in accordance with their respective terms of service and privacy policies.

This is the same flow that users of our chat application will experience when signing in.

Click on **Accept** and you will notice that you are redirected back to our application code, but presented with an `Auth action callback not supported` error. This is because we haven't yet implemented the callback functionality in `loginHandler`.

Handling the response from the provider

Once the user clicks on **Accept** on the provider's website (or if they click on the equivalent of **Cancel**), they will be redirected back to the callback endpoint in our application.

A quick glance at the complete URL that comes back shows us the grant code that the provider has given us.

```
http://localhost:8080/auth/callback/google?code=4/Q92xJ-
BQfoX6PHhzkjhgtyfLc0Ylm.QqV4u9AbA9sYguyfbjFEsNoJKMOjQI
```

We don't have to worry about what to do with this code because Gomniauth will process the OAuth URL parameters for us (by sending the grant code to Google servers and exchanging it for an access token as per the OAuth specification), so we can simply jump to implementing our callback handler. However, it's worth knowing that this code will be exchanged by the authentication provider for a token that allows us to access private user data. For added security, this additional step happens behind the scenes, from server to server rather than in the browser.

In `auth.go`, we are ready to add another switch case to our action path segment. Insert the following code above the default case:

```
case "callback":

  provider, err := gomniauth.Provider(provider)
  if err != nil {
    log.Fatalln("Error when trying to get provider", provider, "-
", err)
  }

  creds, err :=
provider.CompleteAuth(objx.MustFromURLQuery(r.URL.RawQuery))
  if err != nil {
    log.Fatalln("Error when trying to complete auth for",
provider, "-", err)
  }

  user, err := provider.GetUser(creds)
  if err != nil {
    log.Fatalln("Error when trying to get user from", provider, "-
", err)
  }

  authCookieValue := objx.New(map[string]interface{}{
    "name": user.Name(),
  }).MustBase64()
  http.SetCookie(w, &http.Cookie{
    Name:  "auth",
    Value: authCookieValue,
    Path:  "/"})

  w.Header()["Location"] = []string{"/chat"}
  w.WriteHeader(http.StatusTemporaryRedirect)
```

When the authentication provider redirects the users back after they have granted permission, the URL specifies that it is a callback action. We look up the authentication provider as we did before, and call its `CompleteAuth` method. We parse the `RawQuery` from the `http.Request` (the `GET` request that the user's browser is now making) into `objx.Map` (the multi-purpose map type that Gomniauth uses) and the `CompleteAuth` method uses the URL query parameter values to complete the authentication handshake with the provider. All being well, we will be given some authorized credentials with which we access our user's basic data. We then use the `GetUser` method for the provider and Gomniauth uses the specified credentials to access some basic information about the user.

Once we have the user data, we Base64-encode the `Name` field in a JSON object and store it as the value to our `auth` cookie for later use.

Base64-encoding of data ensures it won't contain any special or unpredictable characters, like passing data in a URL or storing it in a cookie. Remember that although Base64-encoded data looks encrypted, it is not—you can easily decode Base64-encoded data back into the original text with little effort. There are online tools that do this for you.

After setting the cookie, we redirect the user to the chat page, which we can safely assume was the original destination.

Once you build and run the code again and hit the `/chat` page, you will notice that the signup flow works, and we are finally allowed back to the chat page. Most browsers have an inspector or a console—a tool that allows you to view the cookies that the server has sent you—that you can use to see whether the `auth` cookie has appeared:

```
go build -o chat
./chat -host=":8080"
```

In our case, the cookie value is `eyJuYW1lIjoiTWF0IFJ5ZXIifQ==`, which is a Base64-encoded version of `{"name":"Mat Ryer"}`. Remember, we never typed in a name in our chat application; instead, Gomniauth asked Google for a name when we opted to sign in with Google. Storing non-signed cookies like this is fine for incidental information such as a user's name, however, you should avoid storing any sensitive information using non-signed cookies, as it's easy for people to access and change the data.

Presenting the user data

Having the user data inside a cookie is a good start, but nontechnical people will never even know it's there, so we must bring the data to the fore. We will do this by enhancing our `templateHandler` method that first passes the user data into the template's `Execute` method; this allows us to use template annotations in our HTML to display the user data to the users.

Update the `ServeHTTP` method of our `templateHandler` in `main.go`:

```
func (t *templateHandler) ServeHTTP(w http.ResponseWriter, r
*http.Request) {
  t.once.Do(func() {
    t.templ =
template.Must(template.ParseFiles(filepath.Join("templates",
t.filename)))
  })
  data := map[string]interface{}{
    "Host": r.Host,
  }
  if authCookie, err := r.Cookie("auth"); err == nil {
    data["UserData"] = objx.MustFromBase64(authCookie.Value)
  }

  t.templ.Execute(w, data)
}
```

Instead of just passing the entire `http.Request` object to our template as data, we are creating a new `map[string]interface{}` definition for a data object that potentially has two fields: `Host` and `UserData` (the latter will only appear if an `auth` cookie is present). By specifying the map type followed by curly braces, we are able to add the `Host` entry at the same time as making our map. We then pass this new `data` object as the second argument to the `Execute` method on our template.

Now we add an HTML file to our template source to display the name. Update the `chatbox` form in `chat.html`:

```
<form id="chatbox">
  {{.UserData.name}}:<br/>
  <textarea></textarea>
  <input type="submit" value="Send" />
</form>
```

The `{{.UserData.name}}` annotation tells the template engine to insert our user's name before the `textarea` control.

[Since we're using the `objx` package, don't forget to run `go get http://github.com/stretchr/objx`, and import it.]

Rebuild and run the chat application again, and you will notice the addition of your name before the chat box:

```
go build -o chat
./chat -host=":8080"
```

Augmenting messages with additional data

So far, our chat application has only transmitted messages as slices of bytes or `[]byte` types between the client and the server; therefore, our forward channel for our room has the `chan []byte` type. In order to send data (such as who sent it and when) in addition to the message itself, we enhance our forward channel and also how we interact with the web socket on both ends.

Define a new type that will replace the `[]byte` slice by creating a new file called `message.go` in the `chat` folder:

```
package main
import (
  "time"
)
// message represents a single message
type message struct {
  Name    string
  Message string
  When    time.Time
}
```

The `message` type will encapsulate the message string itself, but we have also added the `Name` and `When` fields that respectively hold the user's name and a timestamp of when the message was sent.

Since the `client` type is responsible for communicating with the browser, it needs to transmit and receive more than just the single message string. As we are talking to a JavaScript application (that is the chat client running in the browser) and the Go standard library has a great JSON implementation, this seems the perfect choice to encode additional information in the messages. We will change the `read` and `write` methods in `client.go` to use the `ReadJSON` and `WriteJSON` methods on the socket, and we will encode and decode our new `message` type:

```
func (c *client) read() {
  for {
    var msg *message
    if err := c.socket.ReadJSON(&msg); err == nil {
      msg.When = time.Now()
      msg.Name = c.userData["name"].(string)
      c.room.forward <- msg
    } else {
      break
    }
  }
  c.socket.Close()
}
func (c *client) write() {
  for msg := range c.send {
    if err := c.socket.WriteJSON(msg); err != nil {
      break
    }
  }
  c.socket.Close()
}
```

When we receive a message from the browser, we will expect to populate only the `Message` field, which is why we set the `When` and `Name` fields ourselves in the preceding code.

You will notice that when you try to build the preceding code, it complains about a few things. The main reason is that we are trying to send a `*message` object down our `forward` and `send chan []byte` channels. This is not allowed until we change the type of the channel. In `room.go`, change the `forward` field to be of type `chan *message`, and do the same for the `send chan` type in `client.go`.

We must update the code that initializes our channels since the types have now changed. Alternatively, you can wait for the compiler to raise these issues and fix them as you go. In `room.go`, you need to make the following changes:

- Change `forward: make(chan []byte)` to `forward: make(chan *message)`
- Change `r.tracer.Trace("Message received: ", string(msg))` to `r.tracer.Trace("Message received: ", msg.Message)`
- Change `send: make(chan []byte, messageBufferSize)` to `send: make(chan *message, messageBufferSize)`

The compiler will also complain about the lack of user data on a client, which is a fair point because the `client` type has no idea about the new user data we have added to the cookie. Update the `client` struct to include a new `map[string]interface{}` called `userData`:

```
// client represents a single chatting user.
type client struct {
  // socket is the web socket for this client.
  socket *websocket.Conn
  // send is a channel on which messages are sent.
  send chan *message
  // room is the room this client is chatting in.
  room *room
  // userData holds information about the user
  userData map[string]interface{}
}
```

The user data comes from the client cookie that we access through the `http.Request` objects's `Cookie` method. In `room.go`, update `ServeHTTP` with the following changes:

```
func (r *room) ServeHTTP(w http.ResponseWriter, req *http.Request) {
  socket, err := upgrader.Upgrade(w, req, nil)
  if err != nil {
    log.Fatal("ServeHTTP:", err)
    return
  }

  authCookie, err := req.Cookie("auth")
  if err != nil {
    log.Fatal("Failed to get auth cookie:", err)
    return
  }
```

```
    client := &client{
      socket:   socket,
      send:     make(chan *message, messageBufferSize),
      room:     r,
      userData: objx.MustFromBase64(authCookie.Value),
    }
    r.join <- client
    defer func() { r.leave <- client }()
    go client.write()
    client.read()
  }
```

We use the `Cookie` method on the `http.Request` type to get our user data before passing it to the client. We are using the `objx.MustFromBase64` method to convert our encoded cookie value back into a usable map object.

Now that we have changed the type being sent and received on the socket from `[]byte` to `*message`, we must tell our JavaScript client that we are sending JSON instead of just a plain string. Also we must ask that it send JSON back to the server when a user submits a message. In `chat.html`, first update the `socket.send` call:

```
  socket.send(JSON.stringify({"Message": msgBox.val()}));
```

We are using `JSON.stringify` to serialize the specified JSON object (containing just the `Message` field) into a string, which is then sent to the server. Our Go code will decode (or unmarshal) the JSON string into a `message` object, matching the field names from the client JSON object with those of our `message` type.

Finally, update the `socket.onmessage` callback function to expect JSON, and also add the name of the sender to the page:

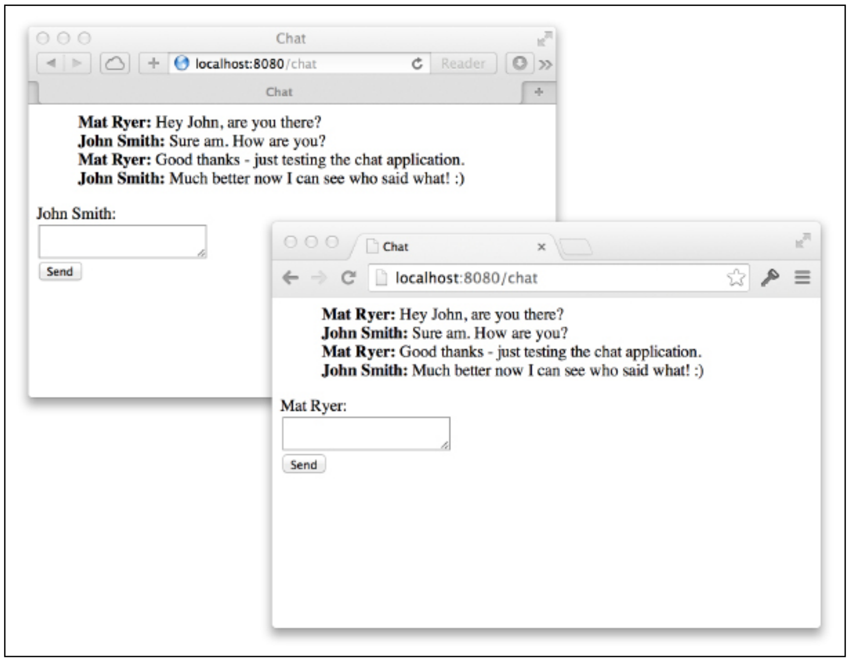
```
  socket.onmessage = function(e) {
    var msg = eval("("+e.data+")");
    messages.append(
      $("<li>").append(
        $("<strong>").text(msg.Name + ": "),
        $("<span>").text(msg.Message)
      )
    );
  }
```

In the preceding code snippet, we've used JavaScript's `eval` function to turn the JSON string into a JavaScript object, and then access the fields to build up the elements needed to properly display them.

Build and run the application, and if you can, log in with two different accounts in two different browsers (or invite a friend to help you test it):

```
go build -o chat
./chat -host=":8080"
```

The following screenshot shows the chat application's browser chat screens:



Mat Ryer: Hey John, are you there?
John Smith: Sure am. How are you?
Mat Ryer: Good thanks - just testing the chat application.
John Smith: Much better now I can see who said what! :)

John Smith:

Mat Ryer: Hey John, are you there?
John Smith: Sure am. How are you?
Mat Ryer: Good thanks - just testing the chat application.
John Smith: Much better now I can see who said what! :)

Mat Ryer:

Summary

In this chapter, we added a useful and necessary feature to our chat application by asking users to authenticate themselves using OAuth2 service providers, before allowing them to join the conversation. We made use of several open source packages such as `Objx` and `Gomniauth`, which dramatically reduced the amount of multi-server complexity we would otherwise need to deal with.

We implemented a pattern when we wrapped `http.Handler` types to allow us to easily specify which paths require the user to be authenticated, and which were available even without an `auth` cookie. Our `MustAuth` helper function allowed us to generate the wrapper types in a fluent and simple way, without adding clutter and confusion to our code.

We saw how to use cookies and Base64-encoding to safely (although not securely) store the state of particular users in their browser, and to make use of that data over normal connections and through web sockets. We took more control of the data available to our templates in order to provide the name of the user to the UI, and saw how to only provide certain data under specific conditions.

Since we needed to send and receive additional information over the web socket, we learned how easy it was to change channels of native types into channels that work with types of our own such as our `message` type. We also learned how to transmit JSON objects over the socket, rather than just slices of bytes. Thanks to the type safety of Go, and the ability to specify types for channels, the compiler helps ensure that we do not send anything other than `message` objects through `chan *message`. Attempting to do so would result in a compiler error, alerting us to the fact right away.

To see the name of the person chatting is a great leap forward in usability from the application we built in the previous chapter, but it's very formal and might not attract modern users of the Web, who are used to a much more visual experience. We are missing pictures of people chatting, and in the next chapter, we will explore different ways in which we can allow users to better represent themselves in our application.

As an extra assignment, see if you can make use of the `time.Time` field that we put into the `message` type to tell users when the messages were sent.

3
Three Ways to Implement Profile Pictures

So far, our chat application has made use of the OAuth2 protocol to allow users to sign in to our application so that we know who is saying what. In this chapter, we are going to add profile pictures to make the chatting experience more engaging.

We will look at the following ways to add pictures or avatars alongside the messages in our application:

- Using the avatar picture provided by the authentication server
- Using the `Gravatar.com` web service to look up a picture by the user's e-mail address
- Allowing the user to upload their own picture and host it themselves

The first two options allow us to delegate the hosting of pictures to a third party—either an authentication service or `Gravatar.com`—which is great because it reduces the cost of hosting our application (in terms of storage costs and bandwidth, since the user's browsers will actually download the pictures from the servers of the authenticating service, not ours). The third option requires us to host pictures ourselves at a location that is web accessible.

These options aren't mutually exclusive; you will most likely use some combination of them in a real-world production application. Towards the end of the chapter, we will see how the flexible design that emerges allows us to try each implementation in turn, until we find an appropriate avatar.

We are going to be agile with our design throughout this chapter, doing the minimum work needed to accomplish each milestone. This means that at the end of each section, we will have working implementations that are demonstrable in the browser. This also means that we will refactor code as and when we need to and discuss the rationale behind the decisions we make as we go.

Specifically, in this chapter, you will learn the following:

- What are good practices to get additional information from authentication services, even when there are no standards in place
- When it is appropriate to build abstractions into our code
- How Go's zero-initialization pattern can save time and memory
- How reusing an interface allows us to work with collections and individual objects in the same way as the existing interface did
- How to use the `Gravatar.com` web service
- How to do MD5 hashing in Go
- How to upload files over HTTP and store them on a server
- How to serve static files through a Go web server
- How to use unit tests to guide the refactoring of code
- How and when to abstract functionality from `struct` types into interfaces

Avatars from the authentication server

It turns out that most authentication servers already have images for their users, and they make them available through the protected user resource that we already know how to access in order to get our users' names. To use this avatar picture, we need to get the URL from the provider, store it in the cookie for our user, and send it through a web socket so that every client can render the picture alongside the corresponding message.

Getting the avatar URL

The schema for user or profile resources is not part of the OAuth2 spec, which means that each provider is responsible for deciding how to represent that data. Indeed, providers do things differently, for example, the avatar URL in a GitHub user resource is stored in a field called `avatar_url`, whereas in Google, the same field is called `picture`. Facebook goes even further by nesting the avatar URL value in a `url` field inside an object called `picture`. Luckily, Gomniauth abstracts this for us; its `GetUser` call on a provider standardizes the interface to get common fields.

In order to make use of the avatar URL field, we need to go back and store its information in our cookie. In `auth.go`, look inside the `callback` action switch case and update the code that creates the `authCookieValue` object as follows:

```
authCookieValue := objx.New(map[string]interface{}{
  "name":       user.Name(),
  "avatar_url": user.AvatarURL(),
}).MustBase64()
```

The `AvatarURL` method called in the preceding code will return the appropriate URL value which we then store in the `avatar_url` field which will be stored in the cookie.

Gomniauth defines a `User` type of interface and each provider implements their own version. The generic `map[string]interface{}` data returned from the authentication server is stored inside each object, and the method calls access the appropriate value using the right field name for that provider. This approach—describing the way information is accessed without being strict about implementation details—is a great use of interfaces in Go.

Transmitting the avatar URL

We need to update our `message` type so that it can also carry with it the avatar URL. In `message.go`, add the `AvatarURL` string field:

```
type message struct {
  Name      string
  Message   string
  When      time.Time
  AvatarURL string
}
```

So far, we have not actually assigned a value to `AvatarURL` like we do for the `Name` field, so we must update our `read` method in `client.go`:

```
func (c *client) read() {
  for {
    var msg *message
    if err := c.socket.ReadJSON(&msg); err == nil {
      msg.When = time.Now()
      msg.Name = c.userData["name"].(string)
      if avatarUrl, ok := c.userData["avatar_url"]; ok {
        msg.AvatarURL = avatarUrl.(string)
      }
      c.room.forward <- msg
    } else {
      break
    }
  }
  c.socket.Close()
}
```

All we have done here is we took the value from the `userData` field that represents what we put into the cookie and assigned it to the appropriate field in `message` if the value was present in the map. We will now take the additional step of checking whether the value is present because we cannot guarantee that the authentication service will provide a value for this field. And since it could be `nil`, it might cause a panic to assign it to a `string` type if it's actually missing.

Adding the avatar to the user interface

Now that our JavaScript client gets an avatar URL value via the socket, we can use it to display the image alongside the messages. We do this by updating the `socket.onmessage` code in `chat.html`:

```
socket.onmessage = function(e) {
  var msg = eval("("+e.data+")");
  messages.append(
    $("<li>").append(
      $("<img>").css({
        width:50,
        verticalAlign:"middle"
      }).attr("src", msg.AvatarURL),
      $("<strong>").text(msg.Name + ": "),
      $("<span>").text(msg.Message)
    )
  );
}
```

When we receive a message, we will insert an `img` tag with the source set to the `AvatarURL` field from the message. We will use jQuery's `css` method to force a width of `50` pixels. This protects us from massive pictures spoiling our interface and allows us to align the image to the middle of the surrounding text.

If we build and run our application having logged in with a previous version, you will find that the `auth` cookie that doesn't contain the avatar URL is still there. We are not asked to sign in again (since we are already logged in), and the code that adds the `avatar_url` field never gets a chance to run. We could delete our cookie and refresh the page, but we would have to keep doing so whenever we make changes during development. Let's solve this problem properly by adding a logout feature.

Logging out

The simplest way to log a user out is to get rid of the `auth` cookie and redirect the user to the chat page, which will in turn cause a redirect to the login page since we just removed the cookie. We do this by adding a new `HandleFunc` call to `main.go`:

```
http.HandleFunc("/logout", func(w http.ResponseWriter, r
*http.Request) {
  http.SetCookie(w, &http.Cookie{
    Name:   "auth",
    Value:  "",
    Path:   "/",
    MaxAge: -1,
  })
  w.Header()["Location"] = []string{"/chat"}
  w.WriteHeader(http.StatusTemporaryRedirect)
})
```

The preceding handler function uses `http.SetCookie` to update the cookie setting `MaxAge` to `-1`, which indicates that it should be deleted immediately by the browser. Not all browsers are forced to delete the cookie, which is why we also provide a new `Value` setting of an empty string, thus removing the user data that would previously have been stored.

As an additional assignment, you can bulletproof your app a little by updating the first line in `ServeHTTP` for your `authHandler` in `auth.go` to make it cope with the empty-value case as well as the missing-cookie case:

```
if cookie, err := r.Cookie("auth"); err ==
http.ErrNoCookie || cookie.Value == ""
```

Instead of ignoring the return of `r.Cookie`, we keep a reference to the returned cookie (if there was actually one) and also add an additional check to see whether the `Value` string of the cookie is empty or not.

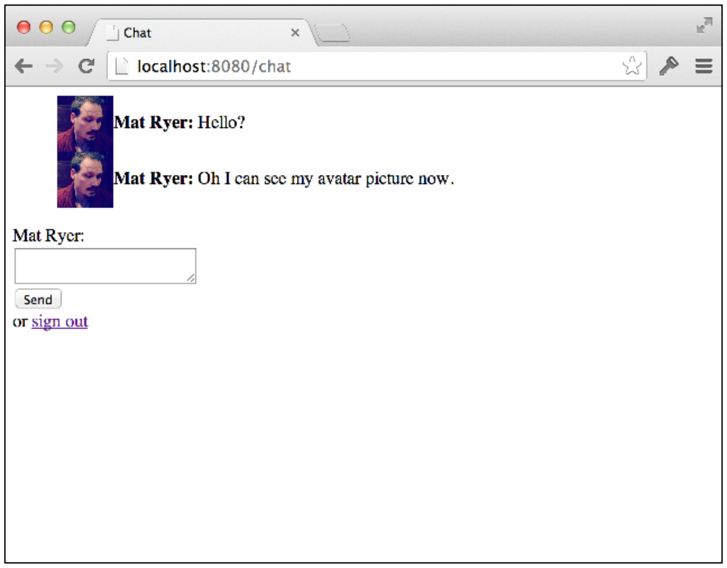
Before we continue, let's add a `Sign Out` link to make it even easier to get rid of the cookie, and also to allow our users to log out. In `chat.html`, update the `chatbox` form to insert a simple HTML link to the new `/logout` handler:

```
<form id="chatbox">
  {{.UserData.name}}:<br/>
  <textarea></textarea>
  <input type="submit" value="Send" />
  or <a href="/logout">sign out</a>
</form>
```

Now build and run the application and open a browser to `localhost:8080/chat`:

```
go build -o chat
./chat -host=:8080
```

Log out if you need to and log back in. When you click on **Send**, you will see your avatar picture appear next to your messages.



Mat Ryer: Hello?

Mat Ryer: Oh I can see my avatar picture now.

Mat Ryer:

Send

or sign out

Making things prettier

Our application is starting to look a little ugly, and it's time to do something about it. In the previous chapter, we implemented the Bootstrap library into our login page, and we are now going to extend its use to our chat page. We will make three changes in `chat.html`: include Bootstrap and tweak the CSS styles for our page, change the markup for our form, and tweak how we render messages on the page.

First, let's update the `style` tag at the top of the page and insert a `link` tag above it to include Bootstrap:

```
<link rel="stylesheet"
href="//netdna.bootstrapcdn.com/bootstrap/3.1.1/css/bootstrap.min.
css">
<style>
  ul#messages        { list-style: none; }
  ul#messages li     { margin-bottom: 2px; }
```

```
    ul#messages li img { margin-right: 10px; }
  </style>
```

Next, let's replace the markup at the top of the `body` tag (before the `script` tags) with the following code:

```
<div class="container">
  <div class="panel panel-default">
    <div class="panel-body">
      <ul id="messages"></ul>
    </div>
  </div>
  <form id="chatbox" role="form">
    <div class="form-group">
      <label for="message">Send a message as
        {{.UserData.name}}</label> or <a href="/logout">Sign
        out</a>
      <textarea id="message" class="form-control"></textarea>
    </div>
    <input type="submit" value="Send" class="btn btn-default" />
  </form>
</div>
```

This markup follows Bootstrap standards of applying appropriate classes to various items, for example, the `form-control` class neatly formats elements within `form` (you can check out the Bootstrap documentation for more information on what these classes do).

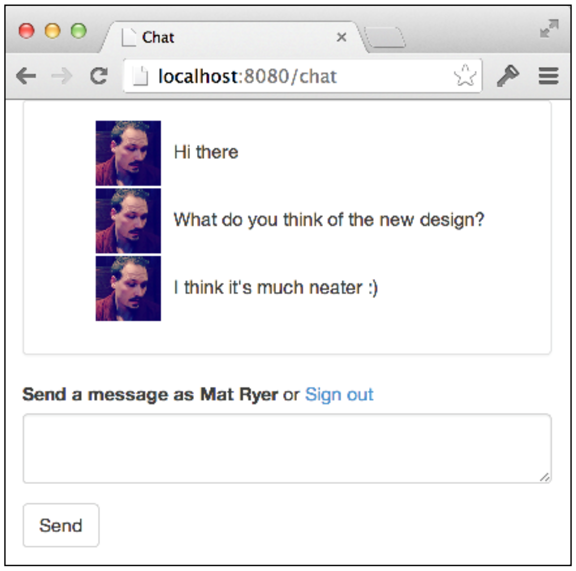
Finally, let's update our `socket.onmessage` JavaScript code to put the sender's name as the `title` attribute for our image. This makes our app display the image when you hover the mouse over it rather than displaying it next to every message:

```
socket.onmessage = function(e) {
  var msg = eval("("+e.data+")");
  messages.append(
    $("<li>").append(
      $("<img>").attr("title", msg.Name).css({
        width:50,
        verticalAlign:"middle"
      }).attr("src", msg.AvatarURL),
      $("<span>").text(msg.Message)
    )
  );
}
```

Build and run the application and refresh your browser to see whether a new design appears:

```
go build -o chat
./chat -host=:8080
```

The preceding command shows the following output:



Hi there

What do you think of the new design?

I think it's much neater :)

Send a message as Mat Ryer or Sign out

With relatively few changes to the code, we have dramatically improved the look and feel of our application.

Implementing Gravatar

Gravatar is a web service that allows users to upload a single profile picture and associate it with their e-mail address to make it available from any website. Developers, like us, can access those images for our application, just by performing a GET operation on a specific API endpoint. In this section, we will see how to implement Gravatar rather than use the picture provided by the authentication service.

Abstracting the avatar URL process

Since we have three different ways of obtaining the avatar URL in our application, we have reached the point where it would be sensible to learn how to abstract the functionality in order to cleanly implement the options. Abstraction refers to a process in which we separate the idea of something from its specific implementation. `http.Handler` is a great example of how a handler will be used along with its ins and outs, without being specific about what action is taken by each handler.

In Go, we start to describe our idea of getting an avatar URL by defining an interface. Let's create a new file called `avatar.go` and insert the following code:

```
package main
import (
  "errors"
)
// ErrNoAvatar is the error that is returned when the
// Avatar instance is unable to provide an avatar URL.
var ErrNoAvatarURL = errors.New("chat: Unable to get an avatar
URL.")
// Avatar represents types capable of representing
// user profile pictures.
type Avatar interface {
  // GetAvatarURL gets the avatar URL for the specified client,
  // or returns an error if something goes wrong.
  // ErrNoAvatarURL is returned if the object is unable to get
  // a URL for the specified client.
  GetAvatarURL(c *client) (string, error)
}
```

The `Avatar` interface describes the `GetAvatarURL` method that a type must satisfy in order to be able to get avatar URLs. We took the client as an argument so that we know for which user to return the URL. The method returns two arguments: a string (which will be the URL if things go well) and an error in case something goes wrong.

One of the things that could go wrong is simply that one of the specific implementations of `Avatar` is unable to get the URL. In that case, `GetAvatarURL` will return the `ErrNoAvatarURL` error as the second argument. The `ErrNoAvatarURL` error therefore becomes a part of the interface; it's one of the possible returns from the method and something that users of our code should probably explicitly handle. We mention this in the comments part of the code for the method, which is the only way to communicate such design decisions in Go.

> Because the error is initialized immediately using `errors.New` and stored in the `ErrNoAvatarURL` variable, only one of these objects will ever be created; passing the pointer of the error as a return is very inexpensive. This is unlike Java's checked exceptions—which serve a similar purpose—where expensive exception objects are created and used as part of the control flow.

The authentication service and avatar's implementation

The first implementation of `Avatar` we write will replace the existing functionality where we hardcoded the avatar URL obtained from the authentication service. Let's use a **Test-driven Development** (**TDD**) approach so we can be sure our code works without having to manually test it. Let's create a new file called `avatar_test.go` in the `chat` folder:

```
package main
import "testing"
func TestAuthAvatar(t *testing.T) {
  var authAvatar AuthAvatar
  client := new(client)
  url, err := authAvatar.GetAvatarURL(client)
  if err != ErrNoAvatarURL {
    t.Error("AuthAvatar.GetAvatarURL should return ErrNoAvatarURL
when no value present")
  }
  // set a value
  testUrl := "http://url-to-gravatar/"
  client.userData = map[string]interface{}{"avatar_url": testUrl}
  url, err = authAvatar.GetAvatarURL(client)
  if err != nil {
    t.Error("AuthAvatar.GetAvatarURL should return no error when
value present")
  } else {
    if url != testUrl {
      t.Error("AuthAvatar.GetAvatarURL should return correct URL")
    }
  }
}
```

This test file contains a test for our as-of-yet nonexistent `AuthAvatar` type's `GetAvatarURL` method. First, it uses a client with no user data and ensures that the `ErrNoAvatarURL` error is returned. After setting a suitable value, our test calls the method again—this time to assert that it returns the correct value. However, building this code fails because the `AuthAvatar` type doesn't exist, so we'll declare `authAvatar` next.

Before we write our implementation, it's worth noticing that we only declare the `authAvatar` variable as the `AuthAvatar` type, but never actually assign anything to it so its value remains `nil`. This is not a mistake; we are actually making use of Go's zero-initialization (or default initialization) capabilities. Since there is no state needed for our object (we will pass `client` as an argument), there is no need to waste time and memory on initializing an instance of it. In Go, it is acceptable to call a method on a `nil` object, provided that the method doesn't try to access a field. When we actually come to writing our implementation, we will look at a way in which we can ensure this is the case.

Let's head back over to `avatar.go` and make our test pass. Add the following code to the bottom of the file:

```
type AuthAvatar struct{}
var UseAuthAvatar AuthAvatar
func (_ AuthAvatar) GetAvatarURL(c *client) (string, error) {
  if url, ok := c.userData["avatar_url"]; ok {
    if urlStr, ok := url.(string); ok {
      return urlStr, nil
    }
  }
  return "", ErrNoAvatarURL
}
```

Here, we define our `AuthAvatar` type as an empty struct and define the implementation of the `GetAvatarURL` method. We also create a handy variable called `UseAuthAvatar` that has the `AuthAvatar` type but which remains of `nil` value. We can later assign the `UseAuthAvatar` variable to any field looking for an `Avatar` interface type.

Normally, the receiver of a method (the type defined in parentheses before the name) will be assigned to a variable so that it can be accessed in the body of the method. Since, in our case, we assume the object can have `nil` value, we can use an underscore to tell Go to throw away the reference. This serves as an added reminder to ourselves that we should avoid using it.

The body of our implementation is otherwise relatively simple: we are safely looking for the value of `avatar_url` and ensuring it is a string before returning it. If anything fails along the way, we return the `ErrNoAvatarURL` error as defined in the interface.

Let's run the tests by opening a terminal and then navigating to the `chat` folder and typing the following:

```
go test
```

All being well, our tests will pass and we will have successfully created our first `Avatar` implementation.

Using an implementation

When we use an implementation, we could refer to either the helper variables directly or create our own instance of the interface whenever we need the functionality. However, this would defeat the very object of the abstraction. Instead, we use the `Avatar` interface type to indicate where we need the capability.

For our chat application, we will have a single way to obtain an avatar URL per chat room. So let's update the `room` type so it can hold an `Avatar` object. In `room.go`, add the following field definition to the type `room struct`:

```
// avatar is how avatar information will be obtained.
avatar Avatar
```

Update the `newRoom` function so we can pass in an `Avatar` implementation for use; we will just assign this implementation to the new field when we create our `room` instance:

```
// newRoom makes a new room that is ready to go.
func newRoom(avatar Avatar) *room {
  return &room{
    forward: make(chan *message),
    join:    make(chan *client),
    leave:   make(chan *client),
    clients: make(map[*client]bool),
    tracer:  trace.Off(),
    avatar:  avatar,
  }
}
```

Building the project now will highlight the fact that the call to `newRoom` in `main.go` is broken because we have not provided an `Avatar` argument; let's update it by passing in our handy `UseAuthAvatar` variable as follows:

```
r := newRoom(UseAuthAvatar)
```

We didn't have to create an instance of `AuthAvatar`, so no memory was allocated. In our case, this doesn't result in great savings (since we only have one room for our whole application), but imagine the size of the potential savings if our application has thousands of rooms. The way we named the `UseAuthAvatar` variable means that the preceding code is very easy to read and it also makes our intention obvious.

Thinking about code readability is important when designing interfaces. Consider a method that takes a Boolean input—just passing in true or false hides the real meaning if you don't know the argument names. Consider defining a couple of helper constants as in the following short example:

```
func move(animated bool) { /* ... */ }
const Animate = true
const DontAnimate = false
```

Think about which of the following calls to `move` are easier to understand:

```
move(true)
move(false)
move(Animate)
move(DontAnimate)
```

All that is left now is to change `client` to use our new `Avatar` interface. In `client.go`, update the `read` method as follows:

```
func (c *client) read() {
  for {
    var msg *message
    if err := c.socket.ReadJSON(&msg); err == nil {
      msg.When = time.Now()
      msg.Name = c.userData["name"].(string)
      msg.AvatarURL, _ = c.room.avatar.GetAvatarURL(c)
      c.room.forward <- msg
    } else {
      break
    }
  }
  c.socket.Close()
}
```

Here, we are asking the `avatar` instance on `room` to get the avatar URL for us instead of extracting it from `userData` ourselves.

When you build and run the application, you will notice that (although we have refactored things a little) the behavior and user experience hasn't changed at all. This is because we told our room to use the `AuthAvatar` implementation.

Now let's add another implementation to the room.

Gravatar implementation

The Gravatar implementation in `Avitar` will do the same job as the `AuthAvatar` implementation, except it will generate a URL for a profile picture hosted on `Gravatar.com`. Let's start by adding a test to our `avatar_test.go` file:

```
func TestGravatarAvatar(t *testing.T) {
  var gravatarAvitar GravatarAvatar
  client := new(client)
  client.userData = map[string]interface{}{"email":
"MyEmailAddress@example.com"}
  url, err := gravatarAvitar.GetAvatarURL(client)
  if err != nil {
    t.Error("GravatarAvitar.GetAvatarURL should not return an
error")
  }
  if url !=
"//www.gravatar.com/avatar/0bc83cb571cd1c50ba6f3e8a78ef1346" {
    t.Errorf("GravatarAvitar.GetAvatarURL wrongly returned %s",
url)
  }
}
```

Gravatar uses a hash of the e-mail address to generate a unique ID for each profile picture, so we set up a client and ensure `userData` contains an e-mail address. Next, we call the same `GetAvatarURL` method, but this time on an object that has the `GravatarAvatar` type. We then assert that a correct URL was returned. We already know this is the appropriate URL for the specified e-mail address because it is listed as an example in the Gravatar documentation—a great strategy to ensure our code is doing what it should be.

Recall that all the source code for this book is available on GitHub. You can save time on building the preceding core by copying and pasting bits and pieces from `https://github.com/matryer/goblueprints`. Hardcoding things such as the base URL is not usually a good idea; we have hardcoded throughout the book to make the code snippets easier to read and more obvious, but you are welcome to extract them as you go along if you like.

Running these tests (with `go test`) obviously causes errors because we haven't defined our types yet. Let's head back to `avatar.go` and add the following code while being sure to import the `io` package:

```
type GravatarAvatar struct{}
var UseGravatar GravatarAvatar
func (_ GravatarAvatar) GetAvatarURL(c *client) (string, error) {
  if email, ok := c.userData["email"]; ok {
    if emailStr, ok := email.(string); ok {
      m := md5.New()
      io.WriteString(m, strings.ToLower(emailStr))
      return fmt.Sprintf("//www.gravatar.com/avatar/%x",
m.Sum(nil)), nil
    }
  }
  return "", ErrNoAvatarURL
}
```

We used the same pattern as we did for `AuthAvatar`: we have an empty struct, a helpful `UseGravatar` variable, and the `GetAvatarURL` method implementation itself. In this method, we follow Gravatar's guidelines to generate an MD5 hash from the e-mail address (after we ensured it was lowercase) and append it to the hardcoded base URL.

It is very easy to achieve hashing in Go, thanks to the hard work put in by the writers of the Go standard library. The `crypto` package has an impressive array of cryptography and hashing capabilities—all very easy to use. In our case, we create a new `md5` hasher; because the hasher implements the `io.Writer` interface, we can use `io.WriteString` to write a string of bytes to it. Calling `Sum` returns the current hash for the bytes written.

You might have noticed that we end up hashing the e-mail address every time we need the avatar URL. This is pretty inefficient, especially at scale, but we should prioritize getting stuff done over optimization. If we need to, we can always come back later and change the way this works.

Running the tests now shows us that our code is working, but we haven't yet included an e-mail address in the `auth` cookie. We do this by locating the code where we assign to the `authCookieValue` object in `auth.go` and updating it to grab the `Email` value from Gomniauth:

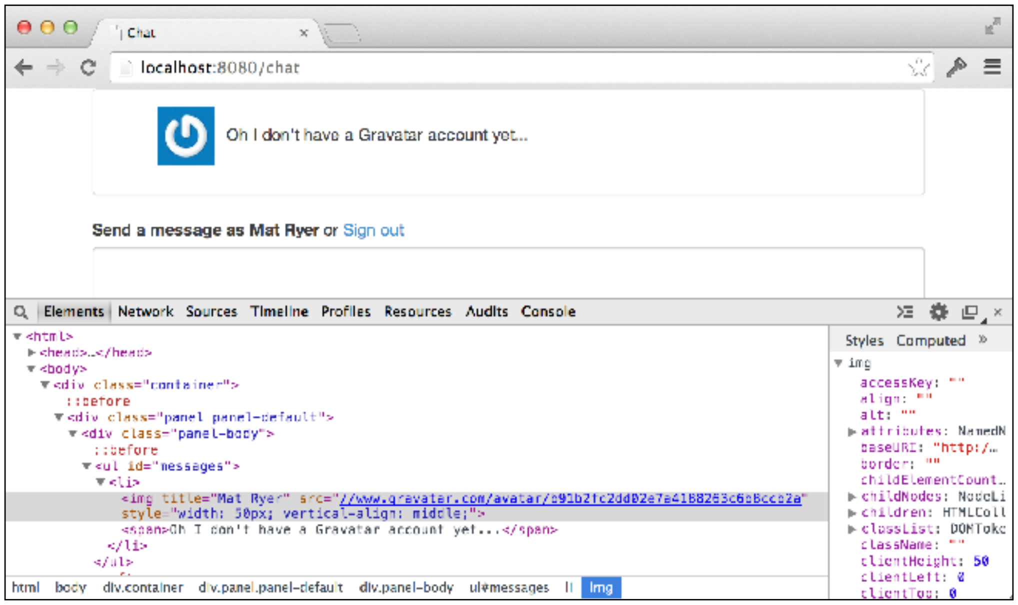
```
authCookieValue := objx.New(map[string]interface{}{
  "name":       user.Name(),
  "avatar_url": user.AvatarURL(),
  "email":      user.Email(),
}).MustBase64()
```

The final thing we must do is tell our room to use the Gravatar implementation instead of the `AuthAvatar` implementation. We do this by calling `newRoom` in `main.go` and making the following change:

```
r := newRoom(UseGravatar)
```

Build and run the chat program once again and head to the browser. Remember, since we have changed the information stored in the cookie, we must sign out and sign back in again in order to see our changes take effect.

Assuming you have a different image for your Gravatar account, you will notice that the system is now pulling the image from Gravatar instead of the authentication provider. Using your browser's inspector or debug tool will show you that the `src` attribute of the `img` tag has indeed changed.


Chat
localhost:8080/chat
Oh I don't have a Gravatar account yet...
Send a message as Mat Ryer or Sign out
Elements Network Sources Timeline Profiles Resources Audits Console
Styles Computed
html body div.container div.panel.panel-default div.panel-body ul#messages li img

```
▼<html>
  ▶<head>…</head>
  ▼<body>
    ▼<div class="container">
        ::before
      ▼<div class="panel panel-default">
        ▼<div class="panel-body">
            ::before
          ▼<ul id="messages">
            ▼<li>
                <img title="Mat Ryer" src="//www.gravatar.com/avatar/c91b2fc2dd02e7a41B8263c6bBccb2a"
                style="width: 50px; vertical-align: middle;">
                <span>Oh I don't have a Gravatar account yet...</span>
              </li>
            </ul>
```

```
▼ img
    accessKey: ""
    align: ""
    alt: ""
  ▶ attributes: NamedN
    baseURI: "http:/…
    border: ""
    childElementCount…
  ▶ childNodes: NodeLi
  ▶ children: HTMLColl
  ▶ classList: DOMToke
    className: ""
    clientHeight: 50
    clientLeft: 0
    clientTop: 0
```

If you don't have a Gravatar account, you'll likely see a default placeholder image in place of your profile picture.

Uploading an avatar picture

In the third and final approach of uploading a picture, we will look at how to allow users to upload an image from their local hard drive to use as their profile picture when chatting. We will need a way to associate a file with a particular user to ensure that we associate the right picture with the corresponding messages.

User identification

In order to uniquely identify our users, we are going to copy Gravatar's approach by hashing their e-mail address and using the resulting string as an identifier. We will store the user ID in the cookie along with the rest of the user-specific data. This will actually have the added benefit of removing from `GravatarAuth` the inefficiency associated with continuous hashing.

In `auth.go`, replace the code that creates the `authCookieValue` object with the following code:

```
m := md5.New()
io.WriteString(m, strings.ToLower(user.Name()))
userId := fmt.Sprintf("%x", m.Sum(nil))
// save some data
authCookieValue := objx.New(map[string]interface{}{
  "userid":     userId,
  "name":       user.Name(),
  "avatar_url": user.AvatarURL(),
  "email":      user.Email(),
}).MustBase64()
```

Here we have hashed the e-mail address and stored the resulting value in the `userid` field at the point at which the user logs in. Henceforth, we can use this value in our Gravatar code instead of hashing the e-mail address for every message. To do this, first we update the test by removing the following line from `avatar_test.go`:

```
client.userData = map[string]interface{}{"email":
"MyEmailAddress@example.com"}
```

We then replace the preceding line with this line:

```
client.userData = map[string]interface{}{"userid":
"0bc83cb571cd1c50ba6f3e8a78ef1346"}
```

We no longer need to set the `email` field since it is not used; instead, we just have to set an appropriate value to the new `userid` field. However, if you run `go test` in a terminal, you will see this test fail.

To make the test pass, in `avatar.go`, update the `GetAvatarURL` method for the `GravatarAuth` type:

```
func (_ GravatarAvatar) GetAvatarURL(c *client) (string, error) {
  if userid, ok := c.userData["userid"]; ok {
    if useridStr, ok := userid.(string); ok {
      return "//www.gravatar.com/avatar/" + useridStr, nil
    }
  }
  return "", ErrNoAvatarURL
}
```

This won't change the behavior, but it allows us to make an unexpected optimization, which is a great example of why you shouldn't optimize code too early—the inefficiencies that you spot early on may not last long enough to warrant the effort required to fix them.

An upload form

If our users are to upload a file as their avatar, they need a way to browse their local hard drive and submit the file to the server. We facilitate this by adding a new template-driven page. In the `chat/templates` folder, create a file called `upload.html`:

```
<html>
  <head>
    <title>Upload</title>
    <link rel="stylesheet"
      href="//netdna.bootstrapcdn.com/bootstrap/3.1.1/css/
      bootstrap.min.css">
  </head>
  <body>
    <div class="container">
      <div class="page-header">
        <h1>Upload picture</h1>
      </div>
      <form role="form" action="/uploader"
        enctype="multipart/form-data" method="post">
        <input type="hidden" name="userid"
          value="{{.UserData.userid}}" />
        <div class="form-group">
          <label for="message">Select file</label>
          <input type="file" name="avatarFile" />
        </div>
        <input type="submit" value="Upload" class="btn " />
      </form>
    </div>
  </body>
</html>
```

We used Bootstrap again to make our page look nice and also to make it fit in with the other pages. However, the key point to note here is the HTML form that will provide the user interface necessary for uploading files. The action points to `/uploader`, the handler for which we have yet to implement, and the `enctype` attribute must be `multipart/form-data` so the browser can transmit binary data over HTTP. Then, there is an `input` element of the type `file`, which will contain the reference to the file we want to upload. Notice also that we have included the `userid` value from the `UserData` map as a hidden input—this will tell us which user is uploading a file. It is important that the `name` attributes are correct, as this is how we will refer to the data when we implement our handler on the server.

Let's now map the new template to the `/upload` path in `main.go`:

```
http.Handle("/upload", &templateHandler{filename: "upload.html"})
```

Handling the upload

When the user clicks on **Upload** after selecting a file, the browser will send the data for the file as well as the user ID to `/uploader`, but right now, that data doesn't actually go anywhere. We will implement a new `HandlerFunc` that is capable of receiving the file, reading the bytes that are streamed through the connection, and saving it as a new file on the server. In the `chat` folder, let's create a new folder called `avatars`—this is where we will save the avatar image files.

Next, create a new file called `upload.go` and insert the following code—make sure to add the appropriate package name and imports (which are `ioutils`, `net/http`, `io`, and `path`):

```
func uploaderHandler(w http.ResponseWriter, req *http.Request) {
  userId := req.FormValue("userid")
  file, header, err := req.FormFile("avatarFile")
  if err != nil {
    io.WriteString(w, err.Error())
    return
  }
```

```
    data, err := ioutil.ReadAll(file)
    if err != nil {
      io.WriteString(w, err.Error())
      return
    }
    filename := path.Join("avatars", userId+path.Ext(header.Filename))
    err = ioutil.WriteFile(filename, data, 0777)
    if err != nil {
      io.WriteString(w, err.Error())
      return
    }
    io.WriteString(w, "Successful")
  }
```

Here, first `uploaderHandler` uses the `FormValue` method on `http.Request` to get the user ID that we placed in the hidden input in our HTML form. Then it gets an `io.Reader` type capable of reading the uploaded bytes by calling `req.FormFile`, which returns three arguments. The first argument represents the file itself with the `multipart.File` interface type, which is also an `io.Reader`. The second is a `multipart.FileHeader` object that contains metadata about the file, such as the filename. And finally, the third argument is an error that we hope will have a `nil` value.

What do we mean when we say that the `multipart.File` interface type is also an `io.Reader`? Well, a quick glance at the documentation at `http://golang.org/pkg/mime/multipart/#File` makes it clear that the type is actually just a wrapper interface for a few other more general interfaces. This means that a `multipart.File` type can be passed to methods that require `io.Reader`, since any object that implements `multipart.File` must therefore implement `io.Reader`.

Embedding standard library interfaces to describe new concepts is a great way to make sure your code works in as many contexts as possible. Similarly, you should try to write code that uses the simplest interface type you can find, ideally from the standard library. For example, if you wrote a method that needed to read the contents of a file, you could ask the user to provide an argument of the type `multipart.File`. However, if you ask for `io.Reader` instead, your code will become significantly more flexible because any type that has the appropriate `Read` method can be passed in, which includes user-defined types too.

The `ioutil.ReadAll` method will just keep reading from the specified `io.Reader` until all of the bytes have been received, so this is where we actually receive the stream of bytes from the client. We then use `path.Join` and `path.Ext` to build a new filename using `userid`, and copy the extension from the original filename that we can get from `multipart.FileHeader`.

We then use the `ioutil.WriteFile` method to create a new file in the `avatars` folder. We use `userid` in the filename to associate the image with the correct user, much in the same way as Gravatar does. The `0777` value specifies that the new file we create has full file permissions, which is a good default setting if you're not sure what other permissions should be set.

If an error occurs at any stage, our code will write it out to the response, which will help us debug it, or it will write **Successful** if everything went well.

In order to map this new handler function to `/uploader`, we need to head back to `main.go` and add the following line to `func main`:

```
http.HandleFunc("/uploader", uploaderHandler)
```

Now build and run the application and remember to log out and log back in again to give our code a chance to upload the `auth` cookie.

```
go build -o chat
./chat -host=:8080
```

Open `http://localhost:8080/upload` and click on **Choose File**, then select a file from your hard drive and click on **Upload**. Navigate to your `chat/avatars` folder and you will notice that the file was indeed uploaded and renamed to the value of your `userid` field.

Serving the images

Now that we have a place to keep our users' avatar images on the server, we need a way to make them accessible to the browser. We do this by using the `net/http` package's built-in file server. In `main.go`, add the following code:

```
http.Handle("/avatars/",
  http.StripPrefix("/avatars/",
    http.FileServer(http.Dir("./avatars"))))
```

This is actually a single line of code that has been broken up to improve readability. The `http.Handle` call should feel familiar: we are specifying that we want to map the `/avatars/` path with the specified handler—this is where things get interesting. Both `http.StripPrefix` and `http.FileServer` return `Handler`, and they make use of the decorator pattern we learned about in the previous chapter. The `StripPrefix` function takes `Handler` in, modifies the path by removing the specified prefix, and passes functionality onto an inner handler. In our case, the inner handler is an `http.FileServer` handler that will simply serve static files, provide index listings, and generate the `404 Not Found` error if it cannot find the file. The `http.Dir` function allows us to specify which folder we want to expose publicly.

If we didn't strip the `/avatars/` prefix from the requests with `http.StripPrefix`, the file server would look for another folder called `avatars` inside the actual `avatars` folder, that is, `/avatars/avatars/filename` instead of `/avatars/filename`.

Let's build the program and run it before opening `http://localhost:8080/avatars/` in a browser. You'll notice that the file server has generated a listing of the files inside our `avatars` folder. Clicking on a file will either download the file, or in the case of an image, simply display it. If you haven't done so already, go to `http://localhost:8080/upload` and upload a picture, then head back to the listing page and click on it to see it in the browser.

The Avatar implementation for local files

The final piece to making filesystem avatars work is to write an implementation of our `Avatar` interface that generates URLs that point to the filesystem endpoint we created in the last section.

Let's add a test function to our `avatar_test.go` file:

```
func TestFileSystemAvatar(t *testing.T) {

  // make a test avatar file
  filename := path.Join("avatars", "abc.jpg")
  ioutil.WriteFile(filename, []byte{}, 0777)
  defer func() { os.Remove(filename) }()

  var fileSystemAvatar FileSystemAvatar
  client := new(client)
  client.userData = map[string]interface{}{"userid": "abc"}
  url, err := fileSystemAvatar.GetAvatarURL(client)
  if err != nil {
    t.Error("FileSystemAvatar.GetAvatarURL should not return an
error")
```

```
  }
  if url != "/avatars/abc.jpg" {
    t.Errorf("FileSystemAvatar.GetAvatarURL wrongly returned %s",
url)
  }
}
```

This test is similar to, but slightly more involved than, the `GravatarAvatar` test because we are also creating a test file in our `avatars` folder and deleting it afterwards.

> The `defer` keyword is a great way to ensure the code runs regardless of what happens in the rest of the function. Even if our test code panics, the deferred functions will still be called.

The rest of the test is simple: we set a `userid` field in `client.userData` and call `GetAvatarURL` to ensure we get back the right value. Of course, running this test will fail, so let's go and add the following code to make it pass in `avatar.go`:

```
type FileSystemAvatar struct{}
var UseFileSystemAvatar FileSystemAvatar
func (_ FileSystemAvatar) GetAvatarURL(c *client) (string, error) {
  if userid, ok := c.userData["userid"]; ok {
    if useridStr, ok := userid.(string); ok {
      return "/avatars/" + useridStr + ".jpg", nil
    }
  }
  return "", ErrNoAvatarURL
}
```

As we see here, to generate the correct URL, we simply get the `userid` value and build the final string by adding the appropriate segments together. You may have noticed that we have hardcoded the file extension to `.jpg`, which means that the initial version of our chat application will only support JPEGs.

> Supporting only JPEGs might seem like a half-baked solution, but following Agile methodologies, this is perfectly fine; after all, custom JPEG profile pictures are better than no custom profile pictures at all.

Let's see our new code in action by updating `main.go` to use our new `Avatar` implementation:

```
r := newRoom(UseFileSystemAvatar)
```

Now build and run the application as usual and go to `http://localhost:8080/upload` and use a web form to upload a JPEG image to use as your profile picture. To make sure it's working correctly, choose a unique image that isn't your Gravatar picture or the image from the authentication service. Once you see the successful message after clicking on **Upload**, go to `http://localhost:8080/chat` and post a message. You will notice that the application has indeed used the profile picture that you uploaded.

To change your profile picture, go back to the `/upload` page and upload a different picture, then jump back to the `/chat` page and post more messages.

Supporting different file types

To support different file types, we have to make our `GetAvatarURL` method for the `FileSystemAvatar` type a little smarter.

Instead of just blindly building the string, we will use the very useful `ioutil.ReadDir` method to get a listing of the files. The listing also includes directories, so we will use the `IsDir` method to determine whether we should skip it or not.

We will then check to see whether each file starts with the `userid` field (remember that we named our files in this way) by a call to `path.Match`. If the filename matches the `userid` field, then we have found the file for that user and we return the path. If anything goes wrong or if we can't find the file, we return the `ErrNoAvatarURL` error as usual.

Update the appropriate method in `avatar.go` with the following code:

```
func (_ FileSystemAvatar) GetAvatarURL(c *client) (string, error) {
  if userid, ok := c.userData["userid"]; ok {
    if useridStr, ok := userid.(string); ok {
      if files, err := ioutil.ReadDir("avatars"); err == nil {
        for _, file := range files {
          if file.IsDir() {
            continue
          }
          if match, _ := path.Match(useridStr+"*", file.Name());
match {
            return "/avatars/" + file.Name(), nil
          }
        }
      }
    }
  }
  return "", ErrNoAvatarURL
}
```

Delete all the files in the `avatar` folder to prevent confusion and rebuild the program. This time upload an image of a different type and notice that our application has no difficulty handling it.

Refactoring and optimizing our code

When we look back at how our `Avatar` type is used, you will notice that every time someone sends a message, the application makes a call to `GetAvatarURL`. In our latest implementation, each time the method is called, we iterate over all the files in the `avatars` folder. For a particularly chatty user, this could mean that we end up iterating over and over again many times a minute. This is an obvious waste of resources and would, at some point very soon, become a scaling problem.

Instead of getting the avatar URL for every message, we will get it only once when the user first logs in and cache it in the `auth` cookie. Unfortunately, our `Avatar` interface type requires that we pass in a `client` object to the `GetAvatarURL` method and we do not have such an object at the point at which we are authenticating the user.

So did we make a mistake when we designed our `Avatar` interface? While this is a natural conclusion to come to, in fact we did the right thing. We designed the solution with the best information we had available at the time and therefore had a working chat application much sooner than if we'd tried to design for every possible future case. Software evolves and almost always changes during the development process and will definitely change throughout the lifetime of the code.

Replacing concrete types with interfaces

We have concluded that our `GetAvatarURL` method depends on a type that is not available to us at the point we need it, so what would be a good alternative? We could pass each required field as a separate argument but this would make our interface brittle, since as soon as an `Avatar` implementation needs a new piece of information, we'd have to change the method signature. Instead, we will create a new type that will encapsulate the information our `Avatar` implementations need while conceptually remaining decoupled from our specific case.

In `auth.go`, add the following code to the top of the page (underneath the `package` keyword of course):

```
import gomniauthcommon "github.com/stretchr/gomniauth/common"
type ChatUser interface {
  UniqueID() string
  AvatarURL() string
}
```

```
type chatUser struct {
  gomniauthcommon.User
  uniqueID string
}
func (u chatUser) UniqueID() string {
  return u.uniqueID
}
```

Here, the `import` statement imported the `common` package from Gomniauth and at the same time gave it a specific name through which it will be accessed: `gomniauthcommon`. This isn't entirely necessary since we have no package name conflicts. However, it makes the code easier to understand.

In the preceding code snippet, we also defined a new interface type called `ChatUser`, which exposes the information needed in order for our `Avatar` implementations to generate the correct URLs. Then, we defined an actual implementation called `chatUser` (notice the lowercase starting letter) that implements the interface. It also makes use of a very interesting feature in Go: type embedding. We actually embedded the interface type `gomniauth/common.User`, which means that our `struct` implements the interface automatically.

You may have noticed that we only actually implemented one of the two required methods to satisfy our `ChatUser` interface. We got away with this because the Gomniauth `User` interface happens to define the same `AvatarURL` method. In practice, when we instantiate our `chatUser` struct—provided we set an appropriate value for the implied Gomniauth `User` field—our object implements both Gomniauth's `User` interface and our own `ChatUser` interface at the same time.

Changing interfaces in a test-driven way

Before we can use our new type, we must update the `Avatar` interface and appropriate implementations to make use of it. As we will follow TDD practices, we are going to make these changes in our test file, see compiler errors when we try to build our code, and see failing tests once we fix those errors before finally making the tests pass.

Open `avatar_test.go` and replace `TestAuthAvatar` with the following code:

```
func TestAuthAvatar(t *testing.T) {
  var authAvatar AuthAvatar
  testUser := &gomniauthtest.TestUser{}
  testUser.On("AvatarURL").Return("", ErrNoAvatarURL)
  testChatUser := &chatUser{User: testUser}
  url, err := authAvatar.GetAvatarURL(testChatUser)
  if err != ErrNoAvatarURL {
    t.Error("AuthAvatar.GetAvatarURL should return ErrNoAvatarURL
when no value present")
  }
  testUrl := "http://url-to-gravatar/"
  testUser = &gomniauthtest.TestUser{}
  testChatUser.User = testUser
  testUser.On("AvatarURL").Return(testUrl, nil)
  url, err = authAvatar.GetAvatarURL(testChatUser)
  if err != nil {
    t.Error("AuthAvatar.GetAvatarURL should return no error when
value present")
  } else {
    if url != testUrl {
      t.Error("AuthAvatar.GetAvatarURL should return correct URL")
    }
  }
}
```

You will also need to import the `gomniauth/test` package as `gomniauthtest` like we did in the last section.

Using our new interface before we have defined it is a good way to check the sanity of our thinking, which is another advantage of practicing TDD. In this new test, we create `TestUser` provided by Gomniauth and embed it into a `chatUser` type. We then pass the new `chatUser` type into our `GetAvatarURL` calls and make the same assertions about output as we always have done.

Gomniauth's `TestUser` type is interesting as it makes use of the `Testify` package's mocking capabilities. See `https://github.com/stretchr/testify` for more information.

The `On` and `Return` methods allow us to tell `TestUser` what to do when specific methods are called. In the first case, we tell the `AvatarURL` method to return the error, and in the second case, we ask it to return the `testUrl` value, which simulates the two possible outcomes we are covering in this test.

Updating the `TestGravatarAvatar` and `TestFileSystemAvatar` tests is much simpler because they rely only on the `UniqueID` method, the value of which we can control directly.

Replace the other two tests in `avatar_test.go` with the following code:

```
func TestGravatarAvatar(t *testing.T) {
  var gravatarAvitar GravatarAvatar
  user := &chatUser{uniqueID: "abc"}
  url, err := gravatarAvitar.GetAvatarURL(user)
  if err != nil {
    t.Error("GravatarAvitar.GetAvatarURL should not return an
error")
  }
  if url != "//www.gravatar.com/avatar/abc" {
    t.Errorf("GravatarAvitar.GetAvatarURL wrongly returned %s",
url)
  }
}
func TestFileSystemAvatar(t *testing.T) {
  // make a test avatar file
  filename := path.Join("avatars", "abc.jpg")
  ioutil.WriteFile(filename, []byte{}, 0777)
  defer func() { os.Remove(filename) }()
  var fileSystemAvatar FileSystemAvatar
  user := &chatUser{uniqueID: "abc"}
  url, err := fileSystemAvatar.GetAvatarURL(user)
  if err != nil {
    t.Error("FileSystemAvatar.GetAvatarURL should not return an
error")
  }
  if url != "/avatars/abc.jpg" {
    t.Errorf("FileSystemAvatar.GetAvatarURL wrongly returned %s",
url)
  }
}
```

Of course, this test code won't even compile because we are yet to update our `Avatar` interface. In `avatar.go`, update the `GetAvatarURL` signature in the `Avatar` interface type to take a `ChatUser` type rather than a `client` type:

```
GetAvatarURL(ChatUser) (string, error)
```

Note that we are using the `ChatUser` interface (uppercase starting letter) rather than our internal `chatUser` implementation struct—after all, we want to be flexible about the types our `GetAvatarURL` methods accept.

Trying to build this will reveal that we now have broken implementations because all the `GetAvatarURL` methods are still asking for a `client` object.

Fixing existing implementations

Changing an interface like the one we have is a good way to automatically find the parts of our code that have been affected because they will cause compiler errors. Of course, if we were writing a package that other people would use, we would have to be far stricter towards changing the interfaces.

We are now going to update the three implementation signatures to satisfy the new interface and change the method bodies to make use of the new type. Replace the implementation for `FileSystemAvatar` with the following:

```
func (_ FileSystemAvatar) GetAvatarURL(u ChatUser) (string, error) {
  if files, err := ioutil.ReadDir("avatars"); err == nil {
    for _, file := range files {
      if file.IsDir() {
        continue
      }
      if match, _ := path.Match(u.UniqueID()+"*", file.Name());
match {
        return "/avatars/" + file.Name(), nil
      }
    }
  }
  return "", ErrNoAvatarURL
}
```

The key change here is that we no longer access the `userData` field on the client, and instead just call `UniqueID` directly on the `ChatUser` interface.

Next, we update the `AuthAvatar` implementation with the following code:

```
func (_ AuthAvatar) GetAvatarURL(u ChatUser) (string, error) {
  url := u.AvatarURL()
  if len(url) > 0 {
    return url, nil
  }
  return "", ErrNoAvatarURL
}
```

Our new design is proving to be much simpler; it's always a good thing if we can reduce the amount of code needed. The preceding code makes a call to get the `AvatarURL` value, and provided it isn't empty (or `len(url) > 0`), we return it; else, we return the `ErrNoAvatarURL` error instead.

Finally, update the `GravatarAvatar` implementation:

```
func (_ GravatarAvatar) GetAvatarURL(u ChatUser) (string, error) {
  return "//www.gravatar.com/avatar/" + u.UniqueID(), nil
}
```

Global variables versus fields

So far, we have assigned the `Avatar` implementation to the `room` type, which enables us to use different avatars for different rooms. However, this has exposed an issue: when our users sign in, there is no concept of which room they are headed to so we cannot know which `Avatar` implementation to use. Because our application only supports a single room, we are going to look at another approach toward selecting implementations: the use of global variables.

A global variable is simply a variable that is defined outside any type definition and is accessible from every part of the package (and from outside the package if it's exported). For a simple configuration, such as which type of `Avatar` implementation to use, they are an easy and simple solution. Underneath the `import` statements in `main.go`, add the following line:

```
// set the active Avatar implementation
var avatars Avatar = UseFileSystemAvatar
```

This defines `avatars` as a global variable that we can use when we need to get the avatar URL for a particular user.

Implementing our new design

We need to change the code that calls `GetAvatarURL` for every message to just access the value that we put into the `userData` cache (via the `auth` cookie). Change the line where `msg.AvatarURL` is assigned, as follows:

```
if avatarUrl, ok := c.userData["avatar_url"]; ok {
  msg.AvatarURL = avatarUrl.(string)
}
```

Find the code inside `loginHandler` in `auth.go` where we call `provider.GetUser` and replace it down to where we set the `authCookieValue` object with the following code:

```
user, err := provider.GetUser(creds)
if err != nil {
  log.Fatalln("Error when trying to get user from", provider, "-",
err)
}
```

```
chatUser := &chatUser{User: user}
m := md5.New()
io.WriteString(m, strings.ToLower(user.Name()))
chatUser.uniqueID = fmt.Sprintf("%x", m.Sum(nil))
avatarURL, err := avatars.GetAvatarURL(chatUser)
if err != nil {
  log.Fatalln("Error when trying to GetAvatarURL", "-", err)
}
```

Here, we created a new `chatUser` variable while setting the `User` field (which represents the embedded interface) to the `User` value returned from Gomniauth. We then saved the `userid` MD5 hash to the `uniqueID` field.

The call to `avatars.GetAvatarURL` is where all of our hard work has paid off, as we now get the avatar URL for the user far earlier in the process. Update the `authCookieValue` line in `auth.go` to cache the avatar URL in the cookie and remove the e-mail address since it is no longer needed:

```
authCookieValue := objx.New(map[string]interface{}{
  "userid":     chatUser.uniqueID,
  "name":       user.Name(),
  "avatar_url": avatarURL,
}).MustBase64()
```

However expensive the work that the `Avatar` implementation needs to do, like iterating over files on the filesystem, it is mitigated by the fact that the implementation only does so when the user first logs in, and not every time they send a message.

Tidying up and testing

Finally, we get to snip away some of the fat that has accumulated during our refactoring process.

Since we no longer store the `Avatar` implementation in `room`, let's remove the field and all references to it from the type. In `room.go`, delete the `avatar Avatar` definition from the `room` struct and update the `newRoom` method:

```
func newRoom() *room {
  return &room{
    forward: make(chan *message),
    join:    make(chan *client),
    leave:   make(chan *client),
    clients: make(map[*client]bool),
    tracer:  trace.Off(),
  }
}
```

[Remember to use the compiler as your to-do list where possible, and follow the errors to find where you have impacted other code.]

In `main.go`, remove the parameter passed into the `newRoom` function call since we are using our global variable instead of this one.

After this exercise, the end user experience remains unchanged. Usually, when refactoring the code, it is the internals that are modified while the public-facing interface remains stable and unchanged.

[It's usually a good idea to run tools such as `golint` and `go vet` against your code as well to make sure it follows good practices and doesn't contain any Go faux pas such as missing comments or badly named functions.]

Combining all three implementations

To close this chapter off with a bang, we will implement a mechanism in which each `Avatar` implementation takes a turn in trying to get the value. If the first implementation returns the `ErrNoAvatarURL` error, we will try the next and so on until we find a useable value.

In `avatar.go`, underneath the `Avatar` type, add the following type definition:

```
type TryAvatars []Avatar
```

The `TryAvatars` type is simply a slice of `Avatar` objects; therefore, we will add the following `GetAvatarURL` method:

```
func (a TryAvatars) GetAvatarURL(u ChatUser) (string, error) {
  for _, avatar := range a {
    if url, err := avatar.GetAvatarURL(u); err == nil {
      return url, nil
    }
  }
  return "", ErrNoAvatarURL
}
```

This means that `TryAvatars` is now a valid `Avatar` implementation and can be used in place of any specific implementation. In the preceding method, we iterated over the slice of `Avatar` objects in an order, calling `GetAvatarURL` for each one. If no error is returned, we return the URL; otherwise, we carry on looking. Finally, if we are unable to find a value, we just return `ErrNoAvatarURL` as per the interface design.

Update the `avatars` global variable in `main.go` to use our new implementation:

```
var avatars Avatar = TryAvatars{
  UseFileSystemAvatar,
  UseAuthAvatar,
  UseGravatar}
```

Here we created a new instance of our `TryAvatars` slice type while putting the other `Avatar` implementations inside it. The order matters since it iterates over the objects in the order in which they appear in the slice. So, first our code will check to see whether the user has uploaded a picture; if they haven't, the code will check whether the authentication service has a picture for us to use. If both the approaches fail, a Gravatar URL will be generated, which in the worst case (for example, if the user hasn't added a Gravatar picture), will render a default placeholder image.

To see our new functionality in action, perform the following steps:

1. Build and rerun the application:
   ```
   go build -o chat
   ./chat -host=:8080
   ```
2. Log out by visiting `http://localhost:8080/logout`.
3. Delete all the pictures from the `avatars` folder.
4. Log back in by navigating to `http://localhost:8080/chat`.
5. Send some messages and take note of your profile picture.
6. Visit `http://localhost:8080/upload` and upload a new profile picture.
7. Log out again and log back in as before.
8. Send some more messages and notice that your profile picture has updated.

Summary

In this chapter, we added three different implementations of profile pictures to our chat application. First we asked the authentication service to provide a URL for us to use. We did this by using Gomniauth's abstraction of the user resource data, which we then included as part of the user interface every time a user would send a message. Using Go's zero (or default) initialization pattern, we were able to refer to different implementations of our `Avatar` interface without actually creating any instances.

We stored data in a cookie for when the user would log in. Therefore, and also given the fact that cookies persist between builds of our code, we added a handy logout feature to help us validate our changes, which we also exposed to our users so that they could log out too. Other small changes to the code and the inclusion of Bootstrap on our chat page dramatically improved the look and feel of our application.

We used MD5 hashing in Go to implement the `Gravatar.com` API by hashing the e-mail address that the authentication service provided. If the e-mail address is not known to Gravatar, they will deliver a nice default placeholder image for us, which means our user interface will never be broken due to missing images.

We then built and completed an upload form and associated the server functionality that saved uploaded pictures in the `avatars` folder. We saw how to expose the saved uploaded pictures to users via the standard library's `http.FileServer` handler. As this introduced inefficiencies in our design by causing too much filesystem access, we refactored our solution with the help of our unit tests. By moving the `GetAvatarURL` call to the point at which users log in, rather than every time a message is sent, we made our code significantly more scalable.

Our special `ErrNoAvatarURL` error type was used as part of our interface design to allow us to inform the calling code when it was not possible to obtain an appropriate URL—this became particularly useful when we created our `Avatars` slice type. By implementing the `Avatar` interface on a slice of `Avatar` types, we were able to make a new implementation that took turns trying to get a valid URL from each of the different options available, starting with the filesystem, then the authentication service, and finally Gravatar. We achieved this with zero impact on how the user would interact with the interface. If an implementation returned `ErrNoAvatarURL`, we tried the next one.

Our chat application is ready to go live so we can invite our friends and have a real conversation. But first we need to choose a domain name to host it at, something we will look at in the next chapter.

4
Command-line Tools to Find Domain Names

The chat application we built in the previous chapters is ready to take the world by storm, but not before we give it a home on the Internet. Before we invite our friends to join the conversation, we need to pick a valid, catchy, and available domain name that we can point to the server running our Go code. Instead of sitting in front of our favorite domain name provider for hours on end trying different names, we are going to develop a few command-line tools that will help us find the right one. As we do so, we will see how the Go standard library allows us to interface with the terminal and other executing applications, as well as explore some patterns and practices to build command-line programs.

In this chapter, you will learn:

- How to build complete command-line applications with as little as a single code file
- How to ensure that the tools we build can be composed with other tools using standard streams
- How to interact with a simple third-party JSON RESTful API
- How to utilize the standard in and out pipes in Go code
- How to read from a streaming source one line at a time
- How to build a WHOIS client to look up domain information
- How to store and use sensitive or deployment-specific information in environment variables

Pipe design for command-line tools

We are going to build a series of command-line tools that use the standard streams (`stdin` and `stdout`) to communicate with the user and with other tools. Each tool will take input line by line via the standard in pipe, process it in some way, and then print the output line by line to the standard out pipe for the next tool or for the user.

By default, the standard input is connected to the user's keyboard, and the standard output is printed to the terminal from which the command was run; however, both can be redirected using redirection metacharacters. It's possible to throw the output away by redirecting it to `NUL` on Windows or `/dev/null` on Unix machines, or redirecting it to a file, which will cause the output to be saved to the disk. Alternatively, you can pipe (using the | pipe character) the output of one program into the input of another; it is this feature that we will make use of in order to connect our various tools together. For example, you could pipe the output from one program to the input of another program in a terminal by using this code:

```
one | two
```

Our tools will work with lines of strings where each line (separated by a linefeed character) represents one string. When run without any pipe redirection, we will be able to interact directly with the programs using the default in and out, which will be useful when testing and debugging our code.

Five simple programs

In this chapter, we will build five small programs that we will combine together at the end. The key features of the programs are as follows:

- **Sprinkle**: This program will add some web-friendly sprinkle words to increase the chances of finding available domain names
- **Domainify**: This program will ensure words are acceptable for a domain name by removing unacceptable characters and replacing spaces with hyphens and adding an appropriate top-level domain (such as `.com` and `.net`) to the end
- **Coolify**: This program will make a boring old normal word into Web 2.0 by fiddling around with vowels
- **Synonyms**: This program will use a third-party API to find synonyms
- **Available**: This program will check to see whether the domain is available or not using an appropriate WHOIS server

Five programs might seem like a lot for one chapter, but don't forget how small entire programs can be in Go.

Sprinkle

Our first program augments incoming words with some sugar terms in order to improve the odds of finding available names. Many companies use this approach to keep the core messaging consistent while being able to afford the `.com` domain. For example, if we pass in the word `chat`, it might pass out `chatapp`; alternatively, if we pass in `talk`, we may get back `talk time`.

Go's `math/rand` package allows us to break away from the predictability of computers to give a chance or opportunity to get involved in our program's process and make our solution feel a little more intelligent than it actually is.

To make our Sprinkle program work, we will:

- Define an array of transformations using a special constant to indicate where the original word will appear
- Use the `bufio` package to scan input from `stdin` and `fmt.Println` to write output to `stdout`
- Use the `math/rand` package to randomly select which transformation to apply to the word, such as appending "app" or prefixing the term with "get"

All of our programs will reside in the `$GOPATH/src` directory. For example, if your `GOPATH` is `~/Work/projects/go`, you would create your program folders in the `~/Work/projects/go/src` folder.

In the `$GOPATH/src` directory, create a new folder called `sprinkle` and add a `main.go` file containing the following code:

```
package main
import (
  "bufio"
  "fmt"
  "math/rand"
  "os"
  "strings"
  "time"
)
const otherWord = "*"
var transforms = []string{
  otherWord,
  otherWord,
  otherWord,
  otherWord,
  otherWord + "app",
  otherWord + "site",
  otherWord + "time",
  "get" + otherWord,
  "go" + otherWord,
  "lets " + otherWord,
}
func main() {
  rand.Seed(time.Now().UTC().UnixNano())
  s := bufio.NewScanner(os.Stdin)
  for s.Scan() {
    t := transforms[rand.Intn(len(transforms))]
    fmt.Println(strings.Replace(t, otherWord, s.Text(), -1))
  }
}
```

From now on, it is assumed that you will sort out the appropriate `import` statements yourself.

The preceding code represents our complete Sprinkle program. It defines three things: a constant, a variable, and the obligatory `main` function, which serves as the entry point to Sprinkle. The `otherWord` constant string is a helpful token that allows us to specify where the original word should occur in each of our possible transformations. It lets us write code such as `otherWord+"extra"`, which makes it clear that, in this particular case, we want to add the word extra to the end of the original word.

The possible transformations are stored in the `transforms` variable that we declare as a slice of strings. In the preceding code, we defined a few different transformations such as adding `app` to the end of a word or `lets` before it. Feel free to add some more in there; the more creative, the better.

In the `main` function, the first thing we do is use the current time as a random seed. Computers can't actually generate random numbers, but changing the seed number for the random algorithms gives the illusion that it can. We use the current time in nanoseconds because it's different each time the program is run (provided the system clock isn't being reset before each run).

We then create a `bufio.Scanner` object (called `bufio.NewScanner`) and tell it to read input from `os.Stdin`, which represents the standard in stream. This will be a common pattern in our five programs since we are always going to read from standard in and write to standard out.

[The `bufio.Scanner` object actually takes `io.Reader` as its input source, so there is a wide range of types that we could use here. If you were writing unit tests for this code, you could specify your own `io.Reader` for the scanner to read from, removing the need for you to worry about simulating the standard input stream.]

As the default case, the scanner allows us to read, one at a time, blocks of bytes separated by defined delimiters such as a carriage return and linefeed characters. We can specify our own split function for the scanner or use one of the options built in the standard library. For example, there is `bufio.ScanWords` that scans individual words by breaking on whitespace rather than linefeeds. Since our design specifies that each line must contain a word (or a short phrase), the default line-by-line setting is ideal.

A call to the `Scan` method tells the scanner to read the next block of bytes (the next line) from the input, and returns a `bool` value indicating whether it found anything or not. This is how we are able to use it as the condition for the `for` loop. While there is content to work on, `Scan` returns `true` and the body of the `for` loop is executed, and when `Scan` reaches the end of the input, it returns `false`, and the loop is broken. The bytes that have been selected are stored in the `Bytes` method of the scanner, and the handy `Text` method that we use converts the `[]byte` slice into a string for us.

Inside the `for` loop (so for each line of input), we use `rand.Intn` to select a random item from the `transforms` slice, and use `strings.Replace` to insert the original word where the `otherWord` string appears. Finally, we use `fmt.Println` to print the output to the default standard output stream.

Let's build our program and play with it:

```
go build -o sprinkle
./sprinkle
```

Once the program is running, since we haven't piped any content in, or specified a source for it to read from, we will use the default behavior where it reads the user input from the terminal. Type in `chat` and hit return. The scanner in our code notices the linefeed character at the end of the word and runs the code that transforms it, outputting the result. For example, if you type `chat` a few times, you might see output like:

```
chat
go chat
chat
lets chat
chat
chat app
```

Sprinkle never exits (meaning the `Scan` method never returns `false` to break the loop) because the terminal is still running; in normal execution, the in pipe will be closed by whatever program is generating the input. To stop the program, hit *Ctrl* + *C*.

Before we move on, let's try running Sprinkle specifying a different input source, we are going to use the `echo` command to generate some content, and pipe it into our Sprinkle program using the pipe character:

```
echo "chat" | ./sprinkle
```

The program will randomly transform the word, print it out, and exit since the `echo` command generates only one line of input before terminating and closing the pipe.

We have successfully completed our first program, which has a very simple but useful function, as we will see.

Exercise – configurable transformations

As an extra assignment, rather than hardcoding the `transformations` array as we have done, see if you can externalize it into a text file or database.

Domainify

Some of the words that output from Sprinkle contain spaces and perhaps other characters that are not allowed in domains, so we are going to write a program, called Domainify, that converts a line of text into an acceptable domain segment and add an appropriate **Top-level Domain** (**TLD**) to the end. Alongside the `sprinkle` folder, create a new one called `domainify`, and add a `main.go` file with the following code:

```
package main
var tlds = []string{"com", "net"}
const allowedChars = "abcdefghijklmnopqrstuvwxyz0123456789_-"
func main() {
  rand.Seed(time.Now().UTC().UnixNano())
  s := bufio.NewScanner(os.Stdin)
  for s.Scan() {
    text := strings.ToLower(s.Text())
    var newText []rune
    for _, r := range text {
      if unicode.IsSpace(r) {
        r = '-'
      }
      if !strings.ContainsRune(allowedChars, r) {
        continue
      }
```

```
            newText = append(newText, r)
          }
          fmt.Println(string(newText) + "." +
                      tlds[rand.Intn(len(tlds))])
        }
      }
```

You will notice a few similarities between the Domainify and Sprinkle programs: we set the random seed using `rand.Seed`, generate a `NewScanner` method wrapping the `os.Stdin` reader, and scan each line until there is no more input.

We then convert the text to lowercase and build up a new slice of `rune` types called `newText`. The `rune` types consist only of characters that appear in the `allowedChars` string, which `strings.ContainsRune` lets us know. If `rune` is a space that we determine by calling `unicode.IsSpace`, we replace it with a hyphen, which is an acceptable practice in domain names.

Ranging over a string returns the index of each character and a `rune` type, which is a numerical value (specifically `int32`) representing the character itself. For more information about runes, characters, and strings, refer to `http://blog.golang.org/strings`.

Finally, we convert `newText` from a `[]rune` slice to a string and add either `.com` or `.net` to the end before printing it out using `fmt.Println`.

Build and run Domainify:

```
go build -o domainify
./domainify
```

Type in some of these options to see how `domainify` reacts:

- `Monkey`
- `Hello Domainify`
- `"What's up?"`
- `One (two) three!`

You can see that, for example, `One (two) three!` might yield `one-two-three.com`.

We are now going to compose Sprinkle and Domainify to see them work together. In your terminal, navigate to the parent folder (probably `$GOPATH/src`) of `sprinkle` and `domainify`, and run the following command:

```
./sprinkle/sprinkle | ./domainify/domainify
```

Here we ran the Sprinkle program and piped the output into the Domainify program. By default, `sprinkle` uses the terminal as the input and `domanify` outputs to the terminal. Try typing in `chat` a few times again, and notice the output is similar to what Sprinkle was outputting previously, except now the words are acceptable for domain names. It is this piping between programs that allows us to compose command-line tools together.

Exercise – making top-level domains configurable

Only supporting `.com` and `.net` top-level domains is fairly limiting. As an additional assignment, see if you can accept a list of TLDs via a command-line flag.

Coolify

Often domain names for common words such as `chat` are already taken and a common solution is to play around with the vowels in the words. For example, we might remove the `a` leaving `cht` (which is actually less likely to be available), or add an `a` to produce `chaat`. While this clearly has no actual effect on coolness, it has become a popular, albeit slightly dated, way to secure domain names that still sound like the original word.

Our third program, Coolify, will allow us to play with the vowels of words that come in via the input, and write the modified versions to the output.

Create a new folder called `coolify` alongside `sprinkle` and `domainify`, and create the `main.go` code file with the following code:

```
package main
const (
  duplicateVowel bool   = true
  removeVowel    bool   = false
)
func randBool() bool {
  return rand.Intn(2) == 0
}
func main() {
  rand.Seed(time.Now().UTC().UnixNano())
  s := bufio.NewScanner(os.Stdin)
  for s.Scan() {
    word := []byte(s.Text())
    if randBool() {
      var vI int = -1
```

```
        for i, char := range word {
          switch char {
          case 'a', 'e', 'i', 'o', 'u', 'A', 'E', 'I', 'O', 'U':
            if randBool() {
              vI = i
            }
          }
        }
        if vI >= 0 {
          switch randBool() {
          case duplicateVowel:
            word = append(word[:vI+1], word[vI:]...)
          case removeVowel:
            word = append(word[:vI], word[vI+1:]...)
          }
        }
      }
      fmt.Println(string(word))
    }
  }
```

While the preceding Coolify code looks very similar to the codes of Sprinkle and Domainify, it is slightly more complicated. At the very top of the code we declare two constants, `duplicateVowel` and `removeVowel`, that help make Coolify code more readable. The `switch` statement decides whether we duplicate or remove a vowel. Also, using these constants, we are able to express our intent very clearly, rather than using just `true` or `false`.

We then define the `randBool` helper function that just randomly returns `true` or `false` by asking the `rand` package to generate a random number, and checking whether if that number comes out as zero. It will be either `0` or `1`, so there's a 50/50 chance of it being `true`.

The `main` function for Coolify starts the same way as the `main` functions for Sprinkle and Domainify—by setting the `rand.Seed` method and creating a scanner of the standard input stream before executing the loop body for each line of input. We call `randBool` first to decide whether we are even going to mutate a word or not, so Coolify will only affect half of the words passed through it.

We then iterate over each rune in the string and look for a vowel. If our `randBool` method returns `true`, we keep the index of the vowel character in the `vI` variable. If not, we keep looking through the string for another vowel, which allows us to randomly select a vowel from the words rather than always modifying the same one.

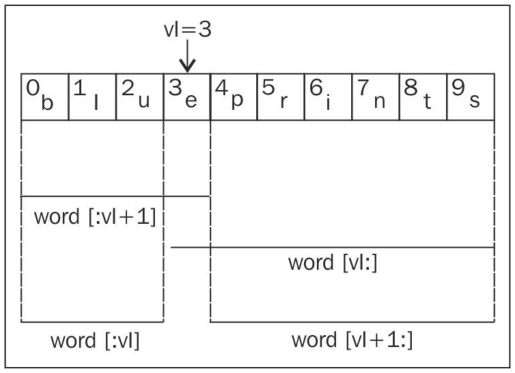
Once we have selected a vowel, we then use `randBool` again to randomly decide what action to take.

This is where the helpful constants come in; consider the following alternative switch statement:

```
switch randBool() {
case true:
  word = append(word[:vI+1], word[vI:]...)
case false:
  word = append(word[:vI], word[vI+1:]...)
}
```

In the preceding code snippet, it's difficult to tell what is going on because `true` and `false` don't express any context. On the other hand, using `duplicateVowel` and `removeVowel` tells anyone reading the code what we mean by the result of `randBool`.

The three dots following the slices cause each item to pass as a separate argument to the `append` function. This is an idiomatic way of appending one slice to another. Inside the `switch` case, we do some slice manipulation to either duplicate the vowel or remove it altogether. We are reslicing our `[]byte` slice and using the `append` function to build a new one made up of sections of the original word. The following diagram shows which sections of the string we access in our code:



If we take the value `blueprints` as an example word, and assume that our code selected the first `e` character as the vowel (so that `vI` is `3`), we can see what each new slice of word represents in this table:

Code	**Value**	**Description**
`word[:vI+1]`	`blue`	Describes a slice from the beginning of the word slice to the selected vowel. The `+1` is required because the value following the colon does not include the specified index; rather it slices up to that value.
`word[vI:]`	`eprints`	Describes a slice starting at and including the selected vowel to the end of the slice.
`word[:vI]`	`blu`	Describes a slice from the beginning of the word slice up to, but not including, the selected vowel.
`word[vI+1:]`	`prints`	Describes a slice from the item following the selected vowel to the end of the slice.

After we modify the word, we print it out using `fmt.Println`.

Let's build Coolify and play with it to see what it can do:

```
go build -o coolify
./coolify
```

When Coolify is running, try typing `blueprints` to see what sort of modifications it comes up with:

```
blueprnts
bleprints
bluepriints
blueprnts
blueprints
bluprints
```

Let's see how Coolify plays with Sprinkle and Domainify by adding their names to our pipe chain. In the terminal, navigate back (using the `cd` command) to the parent folder and run the following commands:

```
./coolify/coolify | ./sprinkle/sprinkle | ./domainify/domainify
```

We will first spice up a word with extra pieces and make it cooler by tweaking the vowels before finally transforming it into a valid domain name. Play around by typing in a few words and seeing what suggestions our code makes.

Synonyms

So far, our programs have only modified words, but to really bring our solution to life, we need to be able to integrate a third-party API that provides word synonyms. This allows us to suggest different domain names while retaining the original meaning. Unlike Sprinkle and Domainify, Synonyms will write out more than one response for each word given to it. Our architecture of piping programs together means this is no problem; in fact we do not even have to worry about it since each of the three programs is capable of reading multiple lines from the input source.

The Big Hugh Thesaurus at `bighughlabs.com` has a very clean and simple API that allows us to make a single HTTP `GET` request in order to look up synonyms.

> If in the future the API we are using changes or disappears (after all, this is the Internet!), you will find some options at `https://github.com/matryer/goblueprints`.

Before you can use the Big Hugh Thesaurus, you'll need an API key, which you can get by signing up to the service at `http://words.bighugelabs.com/`.

Using environment variables for configuration

Your API key is a sensitive piece of configuration information that you won't want to share with others. We could store it as `const` in our code, but that would not only mean we couldn't share our code without sharing our key (not good, especially if you love open source projects), but also, and perhaps more importantly, you would have to recompile your project if the key expires or if you want to use a different one.

A better solution is using an environment variable to store the key, as this will allow you to easily change it if you need to. You could also have different keys for different deployments; perhaps you have one key for development or testing and another for production. This way, you can set a specific key for a particular execution of code, so you can easily switch keys without having to change your system-level settings. Either way, different operating systems deal with environment variables in similar ways, so they are a perfect choice if you are writing cross-platform code.

Create a new environment variable called BHT_APIKEY and set your API key as its value.

For machines running a bash shell, you can modify your ~/.bashrc file or similar to include export commands such as:

```
export BHT_APIKEY=abc123def456ghi789jkl
```

On Windows machines, you can navigate to the properties of your computer and look for **Environment Variables** in the **Advanced** section.

Consuming a web API

Making a request for http://words.bighugelabs.com/apisample.php?v=2&format=json in a web browser shows us what the structure of JSON response data looks like when finding synonyms for the word love:

```
{
  "noun":{
    "syn":[
      "passion",
      "beloved",
      "dear"
    ]
  },
  "verb":{
    "syn":[
      "love",
      "roll in the hay",
      "make out"
    ],
    "ant":[
      "hate"
    ]
  }
}
```

The real API returns a lot more actual words than what is printed here, but the structure is the important thing. It represents an object where the keys describe the types of words (verbs, nouns, and so on) and values are objects that contain arrays of strings keyed on syn or ant (for synonym and antonym respectively); it is the synonyms we are interested in.

To turn this JSON string data into something we can use in our code, we must decode it into structures of our own using capabilities found in the `encoding/json` package. Because we're writing something that could be useful outside the scope of our project, we will consume the API in a reusable package rather than directly in our program code. Create a new folder called `thesaurus` alongside your other program folders (in `$GOPATH/src`) and insert the following code into a new `bighugh.go` file:

```
package thesaurus
import (
  "encoding/json"
  "errors"
  "net/http"
)
type BigHugh struct {
  APIKey string
}
type synonyms struct {
  Noun *words `json:"noun"`
  Verb *words `json:"verb"`
}
type words struct {
  Syn []string `json:"syn"`
}
func (b *BigHugh) Synonyms(term string) ([]string, error) {
  var syns []string
  response, err := http.Get("http://words.bighugelabs.com/api/2/"
+ b.APIKey + "/" + term + "/json")
  if err != nil {
    return syns, errors.New("bighugh: Failed when looking for
synonyms for \"" + term + "\"" + err.Error())
  }
  var data synonyms
  defer response.Body.Close()
  if err := json.NewDecoder(response.Body).Decode(&data); err !=
nil {
    return syns, err
  }
  syns = append(syns, data.Noun.Syn...)
  syns = append(syns, data.Verb.Syn...)
  return syns, nil
}
```

In the preceding code, the `BigHugh` type we define houses the necessary API key and provides the `Synonyms` method that will be responsible for doing the work of accessing the endpoint, parsing the response, and returning the results. The most interesting parts of this code are the `synonyms` and `words` structures. They describe the JSON response format in Go terms, namely an object containing noun and verb objects, which in turn contain a slice of strings in a variable called `Syn`. The tags (strings in backticks following each field definition) tell the `encoding/json` package which fields to map to which variables; this is required since we have given them different names.

> Typically, JSON keys have lowercase names, but we have to use capitalized names in our structures so that the `encoding/json` package knows that the fields exist. If we didn't, the package would simply ignore the fields. However, the types themselves (`synonyms` and `words`) do not need to be exported.

The `Synonyms` method takes a `term` argument and uses `http.Get` to make a web request to the API endpoint in which the URL contains not only the API key value, but also the `term` value itself. If the web request fails for some reason, we will make a call to `log.Fatalln`, which writes the error out to the standard error stream and exits the program with a non-zero exit code (actually an exit code of `1`)—this indicates that an error has occurred.

If the web request is successful, we pass the response body (another `io.Reader`) to the `json.NewDecoder` method and ask it to decode the bytes into the `data` variable that is of our `synonyms` type. We defer the closing of the response body in order to keep memory clean before using Go's built-in `append` function to concatenate both `noun` and `verb` synonyms to the `syns` slice that we then return.

Although we have implemented the `BigHugh` thesaurus, it isn't the only option out there, and we can express this by adding a `Thesaurus` interface to our package. In the `thesaurus` folder, create a new file called `thesaurus.go`, and add the following interface definition to the file:

```
package thesaurus
type Thesaurus interface {
  Synonyms(term string) ([]string, error)
}
```

This simple interface just describes a method that takes a `term` string and returns either a slice of strings containing the synonyms, or an error (if something goes wrong). Our `BigHugh` structure already implements this interface, but now other users could add interchangeable implementations for other services, such as `Dictionary.com` or the Merriam-Webster Online service.

Next we are going to use this new package in a program. Change directory in terminal by backing up a level to `$GOPATH/src`, create a new folder called `synonyms`, and insert the following code into a new `main.go` file you will place in that folder:

```
func main() {
  apiKey := os.Getenv("BHT_APIKEY")
  thesaurus := &thesaurus.BigHugh{APIKey: apiKey}
  s := bufio.NewScanner(os.Stdin)
  for s.Scan() {
    word := s.Text()
    syns, err := thesaurus.Synonyms(word)
    if err != nil {
      log.Fatalln("Failed when looking for synonyms for
\""+word+"\"", err)
    }
    if len(syns) == 0 {
      log.Fatalln("Couldn't find any synonyms for \"" + word +
"\"")
    }
    for _, syn := range syns {
      fmt.Println(syn)
    }
  }
}
```

When you manage your imports again, you will have written a complete program capable of looking up synonyms for words by integrating the Big Huge Thesaurus API.

In the preceding code, the first thing our `main` function does is get the `BHT_APIKEY` environment variable value via the `os.Getenv` call. To bullet proof your code, you might consider double-checking to ensure this value is properly set, and report an error if it is not. For now, we will assume that everything is configured properly.

Next, the preceding code starts to look a little familiar since it scans each line of input again from `os.Stdin` and calls the `Synonyms` method to get a list of replacement words.

Let's build a program and see what kind of synonyms the API comes back with when we input the word `chat`:

```
go build -o synonyms
./synonyms
chat
confab
```

```
confabulation
schmooze
New World chat
Old World chat
conversation
thrush
wood warbler
chew the fat
shoot the breeze
chitchat
chatter
```

The results you get will most likely differ from what we have listed here since we're hitting a live API, but the important aspect here is that when we give a word or term as input to the program, it returns a list of synonyms as output, one per line.

> Try chaining your programs together in various orders to see what result you get. Regardless, we will do this together later in the chapter.

Getting domain suggestions

By composing the four programs we have built so far in this chapter, we already have a useful tool for suggesting domain names. All we have to do now is run the programs while piping the output into input in the appropriate way. In a terminal, navigate to the parent folder and run the following single line:

```
./synonyms/synonyms | ./sprinkle/sprinkle | ./coolify/coolify |
./domainify/domainify
```

Because the `synonyms` program is first in our list, it will receive the input from the terminal (whatever the user decides to type in). Similarly, because `domainify` is last in the chain, it will print its output to the terminal for the user to see. At each step, the lines of words will be piped through the other programs, giving them each a chance to do their magic.

Type in some words to see some domain suggestions, for example, if you type `chat` and hit return, you might see:

```
getcnfab.com
confabulationtim.com
getschmoozee.net
```

```
schmosee.com
neew-world-chatsite.net
oold-world-chatsite.com
conversatin.net
new-world-warblersit.com
gothrush.net
lets-wood-wrbler.com
chw-the-fat.com
```

The number of suggestions you get will actually depend on the number of synonyms, since it is the only program that generates more lines of output than we give it.

We still haven't solved our biggest problem—the fact that we have no idea whether the suggested domain names are actually available or not, so we still have to sit and type each of them into a website. In the next section, we will address this issue.

Available

Our final program, Available, will connect to a WHOIS server to ask for details about domains passed into it—of course, if no details are returned, we can safely assume that the domain is available for purchase. Unfortunately, the WHOIS specification (see `http://tools.ietf.org/html/rfc3912`) is very small and contains no information about how a WHOIS server should reply when you ask it for details about a domain. This means programmatically parsing the response becomes a messy endeavor. To address this issue for now, we will integrate with only a single WHOIS server that we can be sure will have `No match` somewhere in the response when it has no records for the domain.

> A more robust solution might be to have a WHOIS interface with well-defined structures for the details, and perhaps an error message for the cases when the domain doesn't exist—with different implementations for different WHOIS servers. As you can imagine, it's quite a project; perfect for an open source effort.

Create a new folder called `available` alongside the others in `$GOPATH/src` and add a `main.go` file
in it containing the following function code:

```
func exists(domain string) (bool, error) {
  const whoisServer string = "com.whois-servers.net"
  conn, err := net.Dial("tcp", whoisServer+":43")
  if err != nil {
    return false, err
  }
  defer conn.Close()
  conn.Write([]byte(domain + "\r\n"))
  scanner := bufio.NewScanner(conn)
  for scanner.Scan() {
    if strings.Contains(strings.ToLower(scanner.Text()), "no
match") {
      return false, nil
    }
  }
  return true, nil
}
```

The `exists` function implements what little there is in the WHOIS specification by opening a connection to port `43` on the specified `whoisServer` instance with a call to `net.Dial`. We then defer the closing of the connection, which means that however the function exits (successfully or with an error, or even a panic), `Close()` will still be called on the connection `conn`. Once the connection is open, we simply write the domain followed by `\r\n` (the carriage return and line feed characters). This is all the specification tells us, so we are on our own from now on.

Essentially, we are looking for some mention of no match in the response, and that is how we will decide whether a domain exists or not (`exists` in this case is actually just asking the WHOIS server if it has a record for the domain we specified). We use our favorite `bufio.Scanner` method to help us iterate over the lines in the response. Passing the connection into `NewScanner` works because `net.Conn` is actually an `io.Reader` too. We use `strings.ToLower` so we don't have to worry about case sensitivity, and `strings.Contains` to see if any of the lines contains the no match text. If it does, we return `false` (since the domain doesn't exist), otherwise we return `true`.

The `com.whois-servers.net` WHOIS service supports domain names for `.com` and `.net`, which is why the Domainify program only adds these types of domains. If you used a server that had WHOIS information for a wider selection of domains, you could add support for additional TLDs.

Let's add a `main` function that uses our `exists` function to check to see whether the incoming domains are available or not. The check mark and cross mark symbols in the following code are optional—if your terminal doesn't support them you are free to substitute them with simple `Yes` and `No` strings.

Add the following code to `main.go`:

```
  var marks = map[bool]string{true: "✓", false: "×"}
  func main() {
    s := bufio.NewScanner(os.Stdin)
    for s.Scan() {
      domain := s.Text()
      fmt.Print(domain, " ")
      exist, err := exists(domain)
      if err != nil {
        log.Fatalln(err)
      }
      fmt.Println(marks[!exist])
      time.Sleep(1 * time.Second)
    }
  }
```

In the preceding code for the `main` function, we simply iterate over each line coming in via `os.Stdin`, printing out the domain with `fmt.Print` (but not `fmt.Println`, as we do not want the linefeed yet), calling our `exists` function to see whether the domain exists or not, and printing out the result with `fmt.Println` (because we *do* want a linefeed at the end).

Finally, we use `time.Sleep` to tell the process to do nothing for `1` second in order to make sure we take it easy on the WHOIS server.

Most WHOIS servers will be limited in various ways in order to prevent you from taking up too much resources. So slowing things down is a sensible way to make sure we don't make the remote servers angry.

Consider what this also means for unit tests. If a unit test was actually making real requests to a remote WHOIS server, every time your tests run, you will be clocking up stats against your IP address. A much better approach would be to stub the WHOIS server to simulate real responses.

The `marks` map at the top of the preceding code is a nice way to map the Boolean response from `exists` to human-readable text, allowing us to just print the response in a single line using `fmt.Println(marks[!exist])`. We are saying not exist because our program is checking whether the domain is available or not (logically the opposite of whether it exists in the WHOIS server or not).

We can use the check and cross characters in our code happily because all Go code files are UTF-8 compliant—the best way to actually get these characters is to search the Web for them, and use copy and paste to bring them into code; else there are platform-dependent ways to get such special characters.

After fixing the `import` statements for the `main.go` file, we can try out Available to see whether domain names are available or not:

```
go build -o available
./available
```

Once Available is running, type in some domain names:

```
packtpub.com
packtpub.com ×
google.com
google.com ×
madeupdomain1897238746234.net
madeupdomain1897238746234.net ✓
```

As you can see, for domains that are obviously not available, we get our little cross mark, but when we make up a domain name using random numbers, we see that it is indeed available.

Composing all five programs

Now that we have completed all five of our programs, it's time to put them all together so that we can use our tool to find an available domain name for our chat application. The simplest way to do this is to use the technique we have been using throughout this chapter: using pipes in a terminal to connect the output and input.

In the terminal, navigate to the parent folder of the five programs and run the following single line of code:

```
./synonyms/synonyms | ./sprinkle/sprinkle | ./coolify/coolify |
./domainify/domainify | ./available/available
```

Once the programs are running, type in a starting word and see how it generates suggestions before checking their availability.

For example, typing in `chat` might cause the programs to take the following actions:

1. The word `chat` goes into `synonyms` and out comes a series of synonyms:
   - `confab`
   - `confabulation`
   - `schmooze`

2. The synonyms flow into `sprinkle` where they are augmented with web-friendly prefixes and suffixes such as:
    - `confabapp`
    - `goconfabulation`
    - `schmooze time`
3. These new words flow into `coolify`, where the vowels are potentially tweaked:
    - `confabaapp`
    - `goconfabulatioon`
    - `schmoooze time`
4. The modified words then flow into `domainify` where they are turned into valid domain names:
    - `confabaapp.com`
    - `goconfabulatioon.net`
    - `schmooze-time.com`
5. Finally, the domain names flow into `available` where they are checked against the WHOIS server to see whether somebody has already taken the domain or not:
    - `confabaapp.com` ✗
    - `goconfabulatioon.net` ✓
    - `schmooze-time.com` ✓

One program to rule them all

Running our solution by piping programs together is an elegant architecture, but it doesn't have a very elegant interface. Specifically, whenever we want to run our solution, we have to type the long messy line where each program is listed separated by pipe characters. In this section, we are going to write a Go program that uses the `os/exec` package to run each subprogram while piping the output from one into the input of the next as per our design.

Create a new folder called `domainfinder` alongside the other five programs, and create another new folder called `lib` inside that folder. The `lib` folder is where we will keep builds of our subprograms, but we don't want to be copying and pasting them every time we make a change. Instead, we will write a script that builds the subprograms and copies the binaries to the `lib` folder for us.

Create a new file called `build.sh` on Unix machines or `build.bat` for Windows and insert the following code:

```
#!/bin/bash
echo Building domainfinder...
go build -o domainfinder
echo Building synonyms...
cd ../synonyms
go build -o ../domainfinder/lib/synonyms
echo Building available...
cd ../available
go build -o ../domainfinder/lib/available
cd ../build
echo Building sprinkle...
cd ../sprinkle
go build -o ../domainfinder/lib/sprinkle
cd ../build
echo Building coolify...
cd ../coolify
go build -o ../domainfinder/lib/coolify
cd ../build
echo Building domainify...
cd ../domainify
go build -o ../domainfinder/lib/domainify
cd ../build
echo Done.
```

The preceding script simply builds all of our subprograms (including `domainfinder`, which we are yet to write) telling `go build` to place them in our `lib` folder. Be sure to give the new script execution rights by doing `chmod +x build.sh`, or something similar. Run this script from a terminal and look inside the `lib` folder to ensure that it has indeed placed the binaries for our subprograms in there.

[Don't worry about the `no buildable Go source files` error for now, it's just Go telling us that the `domainfinder` program doesn't have any `.go` files to build.]

Create a new file called `main.go` inside `domainfinder` and insert the following code in the file:

```
package main
var cmdChain = []*exec.Cmd{
  exec.Command("lib/synonyms"),
  exec.Command("lib/sprinkle"),
  exec.Command("lib/coolify"),
  exec.Command("lib/domainify"),
  exec.Command("lib/available"),
}
func main() {

  cmdChain[0].Stdin = os.Stdin
  cmdChain[len(cmdChain)-1].Stdout = os.Stdout

  for i := 0; i < len(cmdChain)-1; i++ {
    thisCmd := cmdChain[i]
    nextCmd := cmdChain[i+1]
    stdout, err := thisCmd.StdoutPipe()
    if err != nil {
      log.Fatalln(err)
    }
    nextCmd.Stdin = stdout
  }

  for _, cmd := range cmdChain {
    if err := cmd.Start(); err != nil {
      log.Fatalln(err)
    } else {
      defer cmd.Process.Kill()
    }
  }

  for _, cmd := range cmdChain {
    if err := cmd.Wait(); err != nil {
      log.Fatalln(err)
    }
  }

}
```

The `os/exec` package gives us everything we need to work with running external programs or commands from within Go programs. First, our `cmdChain` slice contains `*exec.Cmd` commands in the order in which we want to join them together.

At the top of the `main` function, we tie the `Stdin` (standard in stream) of the first program to the `os.Stdin` stream for this program, and the `Stdout` (standard out stream) of the last program to the `os.Stdout` stream for this program. This means that, like before, we will be taking input through the standard input stream and writing output to the standard output stream.

Our next block of code is where we join the subprograms together by iterating over each item and setting its `Stdin` to the `Stdout` of the program before it.

The following table shows each program, with a description of where it gets its input from, and where its output goes:

Program	**Input (Stdin)**	**Output (Stdout)**
`synonyms`	The same `Stdin` as `domainfinder`	`sprinkle`
`sprinkle`	`synonyms`	`coolify`
`coolify`	`sprinkle`	`domainify`
`domainify`	`coolify`	`available`
`available`	`domainify`	The same `Stdout` as `domainfinder`

We then iterate over each command calling the `Start` method, which runs the program in the background (as opposed to the `Run` method which will block our code until the subprogram exits—which of course is no good since we have to run five programs at the same time). If anything goes wrong, we bail with `log.Fatalln`, but if the program starts successfully, we then defer a call to kill the process. This helps us ensure the subprograms exit when our `main` function exits, which will be when the `domainfinder` program ends.

Once all of the programs are running, we then iterate over every command again and wait for it to finish. This is to ensure that `domainfinder` doesn't exit early and kill off all the subprograms too soon.

Run the `build.sh` or `build.bat` script again and notice that the `domainfinder` program has the same behavior as we have seen before, with a much more elegant interface.

Summary

In this chapter, we learned how five small command-line programs can, when composed together, produce powerful results while remaining modular. We avoided tightly coupling our programs so they are still useful in their own right. For example, we can use our available program just to check if domain names we manually enter are available or not, or we can use our `synonyms` program just as a command-line thesaurus.

We learned how standard streams could be used to build different flows of these types of programs, and how redirection of the standard input and the standard output lets us play around with different flows very easily.

We learned how simple it is in Go to consume a JSON RESTful APIs web service when we needed to get synonyms from the Big Hugh Thesaurus. We kept it simple at first by coding it inline and later refactoring the code to abstract the `Thesaurus` type into its own package, which is ready to share. We also consumed a non-HTTP API when we opened a connection to the WHOIS server and wrote data over raw TCP.

We saw how the `math/rand` package can bring a little variety and unpredictability, by allowing us to use pseudo random numbers and decisions in our code, which meant that each time we run our program, we get different results.

Finally, we built our `domainfinder` super program that composes all the subprograms together giving our solution a simple, clean, and elegant interface.

5
Building Distributed Systems and Working with Flexible Data

In this chapter, we will explore transferrable skills that allow us to use schemaless data and distributed technologies to solve big data problems. The system we will build in this chapter will prepare us for a future where democratic elections all happen online—on Twitter of course. Our solution will collect and count votes by querying Twitter's streaming API for mentions of specific hashtags, and each component will be capable of horizontally scaling to meet demand. Our use case is a fun and interesting one, but the core concepts we'll learn and specific technology choices we'll make are the real focus of this chapter. The ideas discussed here are directly applicable to any system that needs true-scale capabilities.

> Horizontal scaling refers to adding nodes, such as physical machines, to a system in order to improve its availability, performance, and/or capacity. Big data companies such as Google can scale by adding affordable and easy-to-obtain hardware (commonly referred to as commodity hardware) due to the way they write their software and architect their solutions. Vertical scaling is synonymous with increasing the resource available to a single node, such as adding additional RAM to a box, or a processor with more cores.

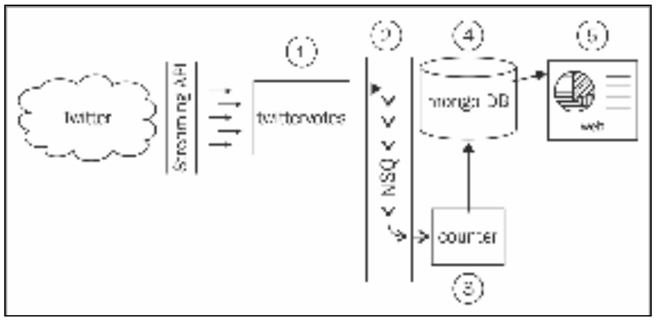
In this chapter, you will:

- Learn about distributed NoSQL datastores; specifically how to interact with MongoDB
- Learn about distributed messaging queues; specifically Bit.ly's NSQ and how to use the `go-nsq` package to easily publish and subscribe to events

- Stream live tweet data through Twitter's streaming APIs and manage long running net connections
- Learn about how to properly stop programs with many internal goroutines
- Learn how to use low memory channels for signaling

System design

Having a basic design sketched out is often useful, especially in distributed systems where many components will be communicating with each other in different ways. We don't want to spend too long on this stage because our design is likely to evolve as we get stuck into the details, but we will look at a high-level outline so we can discuss the constituents and how they fit together.



The preceding image shows the basic overview of the system we are going to build:

- Twitter is the social media network we all know and love.
- Twitter's streaming API allows long-running connections where tweet data is streamed as quickly as possible.
- `twittervotes` is a program we will write that reads tweets and pushes the votes into the messaging queue. `twittervotes` pulls the relevant tweet data, figures out what is being voted for (or rather, which options are mentioned), and pushes the vote into NSQ.
- NSQ is an open source, real-time distributed messaging platform designed to operate at scale, built and maintained by Bit.ly. NSQ carries the message across its instances making it available to anyone who has expressed an interest in the vote data.

- `counter` is a program we will write that listens out for votes on the messaging queue, and periodically saves the results in the MongoDB database. `counter` receives the vote messages from NSQ and keeps an in-memory tally of the results, periodically pushing an update to persist the data.
- MongoDB is an open source document database designed to operate at scale.
- `web` is a web server program that will expose the live results that we will write in the next chapter.

It could be argued that a single Go program could be written that reads the tweets, counts the votes, and pushes them to a user interface but such a solution, while being a great proof of concept, would be very limited in scale. In our design, any one of the components can be horizontally scaled as the demand for that particular capability increases. If we have relatively few polls, but lots of people viewing the data, we can keep the `twittervotes` and `counter` instances down and add more `web` and MongoDB nodes, or vice versa if the situation is reversed.

Another key advantage to our design is redundancy; since we can have many instances of our components working at the same time, if one of our boxes disappears (due to a system crash or power cut, for example) the others can pick up the slack. Modern architectures often distribute such a system over the geographical expanse to protect from local natural disasters too. All of these options are available to use if we build our solution in this way.

We chose the specific technologies in this chapter because of their links to Go (NSQ, for example, is written entirely in Go), and the availability of well-tested drivers and packages. Conceptually, however, you can drop in a variety of alternatives as you see fit.

Database design

We will call our MongoDB database `ballots`. It will contain a single collection called `polls` which is where we will store the poll details, such as the title, the options, and the results (in a single JSON document). The code for a poll will look something like this:

```
{
  "_id": "???",
  "title": "Poll title",
  "options": ["one", "two", "three"],
  "results": {
    "one": 100,
    "two": 200,
    "three": 300
  }
}
```

The `_id` field is automatically generated by MongoDB and will be how we identify each poll. The `options` field contains an array of string options; these are the hashtags we will look for on Twitter. The `results` field is a map where the key represents the option, and the value represents the total number of votes for each item.

Installing the environment

The code we write in this chapter has real external dependencies that we need to get set up before we can start to build our system.

> Be sure to check out the chapter notes at `https://github.com/matryer/goblueprints` if you get stuck on installing any of the dependencies.

In most cases, services such as `mongod` and `nsqd` will have to be started before we can run our programs. Since we are writing components of a distributed system, we will have to run each program at the same time, which is as simple as opening many terminal windows.

NSQ

NSQ is a messaging queue that allows one program to send messages or events to another, or to many other programs running either locally on the same machine, or on different nodes connected by a network. NSQ guarantees the delivery of messages, which means it keeps undelivered messages cached until all interested parties have received them. This means that, even if we stop our `counter` program, we won't miss any votes. You can contrast this capability with fire-and-forget message queues where information is deemed out-of-date, and therefore is forgotten if it isn't delivered in time, and where the sender of the messages doesn't care if the consumer received them or not.

A message queue abstraction allows you to have different components of a system running in different places, provided they have network connectivity to the queue. Your programs are decoupled from others; instead, your designs start to care about the ins and outs of specialized micro-services, rather than the flow of data through a monolithic program.

NSQ transfers raw bytes, which means it is up to us how we encode data into those bytes. For example, we could encode the data as JSON or in a binary format depending on our needs. In our case, we are going to send the vote option as a string without any additional encoding, since we are only sharing a single data field.

Open `http://nsq.io/deployment/installing.html` in a browser (or search `install nsq`) and follow the instructions for your environment. You can either download pre-compiled binaries or build your own from the source. If you have homebrew installed, installing NSQ is as simple as typing:

```
brew install nsq
```

Once you have installed NSQ, you will need to add the `bin` folder to your `PATH` environment variable so that the tools are available in a terminal.

To validate that NSQ is properly installed, open a terminal and run `nsqlookupd`; if the program successfully starts, you should see some output similar to the following:

```
nsqlookupd v0.2.27 (built w/go1.3)
TCP: listening on [::]:4160
HTTP: listening on [::]:4161
```

We are going to use the default ports to interact with NSQ so take note of the TCP and HTTP ports listed in the output, as we will be referring to them in our code.

Press *Ctrl* + *C* to stop the process for now; we'll start them properly later.

The key tools from the NSQ install that we are going to use are `nsqlookupd` and `nsqd`. The `nsqlookupd` program is a daemon that manages topology information about the distributed NSQ environment; it keeps track of all the `nsqd` producers for specific topics and provides interfaces for clients to query such information. The `nsqd` program is a daemon that does the heavy lifting for NSQ such as receiving, queuing, and delivering messages from and to interested parties. For more information and background on NSQ, visit `http://nsq.io/`.

NSQ driver for Go

The NSQ tools themselves are written in Go, so it is logical that the Bit.ly team already has a Go package that makes interacting with NSQ very easy. We will need to use it, so in a terminal, get it using `go get`:

```
go get github.com/bitly/go-nsq
```

MongoDB

MongoDB is a document database, which basically allows you to store and query JSON documents and the data within them. Each document goes into a collection that can be used to group the documents together without enforcing any schema on the data inside them. Unlike rows in a traditional RDBMS such as Oracle, Microsoft SQL Server, or MySQL, it is perfectly acceptable for documents to have a different shape. For example, a `people` collection can contain the following three JSON documents at the same time:

```
{"name":"Mat","lang":"en","points":57}
{"name":"Laurie","position":"Scrum Master"}
{"position":"Traditional Manager","exists":false}
```

This flexibility allows data with varying structure to coexist without impacting performance or wasting space. It is also extremely useful if you expect your software to evolve over time, as we really always should.

MongoDB was designed to scale while also remaining very easy to work with on single-box install such as our development machine. When we host our application for production, we would likely install a more complex multi-sharded, replicated system, which is distributed across many nodes and locations, but for now, just running `mongod` will do.

Head over to `http://www.mongodb.org/downloads` to grab the latest version of MongoDB and install it, making sure to register the `bin` folder with your `PATH` environment variable as usual.

To validate that MongoDB is successfully installed, run the `mongod` command, then hit *Ctrl* + *C* to stop it for now.

MongoDB driver for Go

Gustavo Niemeyer has done a great job in simplifying interactions with MongoDB with his `mgo` (pronounced "mango") package hosted at `http://labix.org/mgo`, which is *go gettable* with the following command:

```
go get gopkg.in/mgo.v2
```

Starting the environment

Now that we have all the pieces we need installed, we need to start our environment. In this section, we are going to:

- Start `nsqlookupd` so that our `nsqd` instances are discoverable
- Start `nsqd` and tell it which `nsqlookupd` to use
- Start `mongod` for data services

Each of these daemons should run in their own terminal window, which will make it easy for us to stop them by just hitting *Ctrl* + *C*.

> Remember the page number for this section as you will likely revisit it a few times as you work through this chapter.

In a terminal window, run:

```
nsqlookupd
```

Take note of the TCP port, which by default is `4160`, and in another terminal window, run:

```
nsqd --lookupd-tcp-address=localhost:4160
```

Make sure the port number in the `--lookupd-tcp-address` flag matches the TCP port of the `nsqlookupd` instance. Once you start `nsqd`, you will notice some output is printed to the terminal from both `nsqlookupd` and `nsqd`; this indicates that the two processes are talking to each other.

In yet another window or tab, start MongoDB by running:

```
mongod --dbpath ./db
```

The `dbpath` flag tells MongoDB where to store the data files for our database. You can pick any location you like, but you'll have to make sure the folder exists before `mongod` will run.

> By deleting the `dbpath` folder at any time, you can effectively erase all data and start afresh. This is especially useful during development.

Now that our environment is running, we are ready to start building our components.

Votes from Twitter

In your `$GOPATH/src` folder, alongside other projects, create a new folder called `socialpoll` for this chapter. This folder won't be a Go package or program by itself, but will contain our three component programs. Inside `socialpoll`, create a new folder called `twittervotes` and add the obligatory `main.go` template (this is important as `main` packages without a `main` function won't compile):

```
package main
func main(){}
```

Our `twittervotes` program is going to:

- Load all polls from the MongoDB database using `mgo`, and collect all options from the `options` array in each document
- Open and maintain a connection to Twitter's streaming APIs looking for any mention of the options
- For each tweet that matches the filter, figure out which option is mentioned and push that option through to NSQ
- If the connection to Twitter is dropped (which is common in long-running connections as it is actually part of Twitter's streaming API specification) after a short delay (so we do not bombard Twitter with connection requests), reconnect and continue
- Periodically re-query MongoDB for the latest polls and refresh the connection to Twitter to make sure we are always looking out for the right options
- When the user terminates the program by hitting *Ctrl* + *C*, it will gracefully stop itself

Authorization with Twitter

In order to use the streaming API, we will need authentication credentials from Twitter's Application Management console, much in the same way we did for our Gomniauth service providers in *Chapter 3*, *Three Ways to Implement Profile Pictures*. Head over to `https://apps.twitter.com` and create a new app called something like `SocialPoll` (the names have to be unique, so you can have some fun here; the choice of name doesn't affect the code either way). When your app has been created, visit the **API Keys** tab and locate the **Your access token** section where you need to create a new access token. After a short delay, refresh the page and notice that you in fact have two sets of keys and secrets; an API key and a secret, and an access token and the corresponding secret. Following good coding practices, we are going to set these values as environment variables so that our program can have access to them without us having to hardcode them in our source files.

The keys we will use in this chapter are:

- `SP_TWITTER_KEY`
- `SP_TWITTER_SECRET`
- `SP_TWITTER_ACCESSTOKEN`
- `SP_TWITTER_ACCESSSECRET`

You can set the environment variables however you like, but since the app relies on them in order to work, creating a new file called `setup.sh` (for bash shells) or `setup.bat` (on Windows) is a good idea since you can check such files into your source code repository. Insert the following code in `setup.sh` or `setup.bat` by copying the appropriate values from the Twitter app page:

```
#!/bin/bash
export SP_TWITTER_KEY=yCwwKKnuBnUBrelyTN...
export SP_TWITTER_SECRET=6on0YRYniT1sI3f...
export SP_TWITTER_ACCESSTOKEN=2427-13677...
export SP_TWITTER_ACCESSSECRET=SpnZf336u...
```

Run the file with the source or call commands to have the values appropriately set, or add them to your `.bashrc` or `C:\cmdauto.cmd` files to save you running them every time you open a new terminal window.

Extracting the connection

The Twitter streaming API supports HTTP connections that stay open for a long time, and given the design of our solution, we are going to need to access the `net.Conn` object in order to close it from outside of the goroutine in which requests occur. We can achieve this by providing our own `dial` method to an `http.Transport` object that we will create.

Create a new file called `twitter.go` inside `twittervotes` (which is where all things Twitter-related will live), and insert the following code:

```
var conn net.Conn
func dial(netw, addr string) (net.Conn, error) {
  if conn != nil {
    conn.Close()
    conn = nil
  }
  netc, err := net.DialTimeout(netw, addr, 5*time.Second)
  if err != nil {
    return nil, err
  }
  conn = netc
  return netc, nil
}
```

Our bespoke `dial` function first ensures `conn` is closed, and then opens a new connection keeping the `conn` variable updated with the current connection. If a connection dies (Twitter's API will do this from time to time) or is closed by us, we can redial without worrying about zombie connections.

We will periodically close the connection ourselves and initiate a new one, because we want to reload the options from the database at regular intervals. To do this, we need a function that closes the connection, and also closes an `io.ReadCloser` that we will use to read the body of the responses. Add the following code to `twitter.go`:

```
var reader io.ReadCloser
func closeConn() {
  if conn != nil {
    conn.Close()
  }
  if reader != nil {
    reader.Close()
  }
}
```

Now we can call `closeConn` at any time to break the ongoing connection with Twitter and tidy things up. In most cases, our code will load the options from the database again and open a new connection right away, but if we're shutting the program down (in response to a *Ctrl* + *C* hit) then we can call `closeConn` just before we exit.

Reading environment variables

Next we are going to write a function that will read the environment variables and set up the `OAuth` objects we'll need in order to authenticate the requests. Add the following code in the `twitter.go` file:

```
var (
  authClient *oauth.Client
  creds *oauth.Credentials
)
```

```
func setupTwitterAuth() {
  var ts struct {
    ConsumerKey    string `env:"SP_TWITTER_KEY,required"`
    ConsumerSecret string `env:"SP_TWITTER_SECRET,required"`
    AccessToken    string `env:"SP_TWITTER_ACCESSTOKEN,required"`
    AccessSecret   string `env:"SP_TWITTER_ACCESSSECRET,required"`
  }
  if err := envdecode.Decode(&ts); err != nil {
    log.Fatalln(err)
  }
  creds = &oauth.Credentials{
    Token:  ts.AccessToken,
    Secret: ts.AccessSecret,
  }
  authClient = &oauth.Client{
    Credentials: oauth.Credentials{
      Token:  ts.ConsumerKey,
      Secret: ts.ConsumerSecret,
    },
  }
}
```

Here we define a `struct` type to store the environment variables that we need to authenticate with Twitter. Since we don't need to use the type elsewhere, we define it inline and creating a variable called `ts` of this anonymous type (that's why we have the somewhat unusual `var ts struct…` code). We then use Joe Shaw's elegant `envdecode` package to pull in those environment variables for us. You will need to run `go get github.com/joeshaw/envdecode` and also import the `log` package. Our program will try to load appropriate values for all the fields marked `required`, and return an error if it fails to do so, which reminds people that the program won't work without Twitter credentials.

The strings inside the back ticks alongside each field in `struct` are called tags, and are available through a reflection interface, which is how `envdecode` knows which variables to look for. Tyler Bunnell and I added the required argument to this package, which indicates that it is an error for any of the environment variables to be missing (or empty).

Once we have the keys, we use them to create `oauth.Credentials` and an `oauth.Client` object from Gary Burd's `go-oauth` package, which will allow us to authorize requests with Twitter.

Now that we have the ability to control the underlying connection and authorize requests, we are ready to write the code that will actually build the authorized request, and return the response. In `twitter.go`, add the following code:

```
var (
  authSetupOnce sync.Once
  httpClient    *http.Client
)
func makeRequest(req *http.Request, params url.Values) (*http.
Response, error) {
  authSetupOnce.Do(func() {
    setupTwitterAuth()
    httpClient = &http.Client{
      Transport: &http.Transport{
        Dial: dial,
      },
    }
  })
  formEnc := params.Encode()
  req.Header.Set("Content-Type", "application/x-www-form-
urlencoded")
  req.Header.Set("Content-Length", strconv.Itoa(len(formEnc)))
  req.Header.Set("Authorization",
authClient.AuthorizationHeader(creds, "POST", req.URL, params))
  return httpClient.Do(req)
}
```

We use `sync.Once` to ensure our initialization code only gets run once despite the number of times we call `makeRequest`. After calling the `setupTwitterAuth` method, we create a new `http.Client` using an `http.Transport` that uses our custom `dial` method. We then set the appropriate headers needed for authorization with Twitter by encoding the specified `params` object that will contain the options we are querying for.

Reading from MongoDB

In order to load the polls, and therefore the options to search Twitter for, we need to connect to and query MongoDB. In `main.go`, add the two functions `dialdb` and `closedb`:

```
var db *mgo.Session
func dialdb() error {
  var err error
  log.Println("dialing mongodb: localhost")
  db, err = mgo.Dial("localhost")
  return err
}
func closedb() {
  db.Close()
  log.Println("closed database connection")
}
```

These two functions will connect to and disconnect from the locally running MongoDB instance using the `mgo` package, and store `mgo.Session` (the database connection object) in a global variable called `db`.

[As an additional assignment, see if you can find an elegant way to make the location of the MongoDB instance configurable so that you don't need to run it locally.]

Assuming MongoDB is running and our code is able to connect, we need to load the poll objects and extract all the options from the documents, which we will then use to search Twitter. Add the following `Options` function to `main.go`:

```
type poll struct {
  Options []string
}
func loadOptions() ([]string, error) {
  var options []string
  iter := db.DB("ballots").C("polls").Find(nil).Iter()
  var p poll
  for iter.Next(&p) {
    options = append(options, p.Options...)
  }
  iter.Close()
  return options, iter.Err()
}
```

Our poll document contains more than just `Options`, but our program doesn't care about anything else, so there's no need for us to bloat our `poll` struct. We use the `db` variable to access the `polls` collection from the `ballots` database, and call the `mgo` package's fluent `Find` method, passing `nil` (meaning no filtering).

[A fluent interface (first coined by Eric Evans and Martin Fowler) refers to an API design that aims to make the code more readable by allowing you to chain together method calls. This is achieved by each method returning the context object itself, so that another method can be called directly afterwards. For example, `mgo` allows you to write queries such as this:

```
query := col.Find(q).Sort("field").Limit(10).Skip(10)
```
]

We then get an iterator by calling the `Iter` method, which allows us to access each poll one by one. This is a very memory-efficient way of reading the poll data, because it only ever uses a single `poll` object. If we were to use the `All` method instead, the amount of memory we'd use would depend on the number of polls we had in our database, which would be out of our control.

When we have a poll, we use the `append` method to build up the options slice. Of course, with millions of polls in the database, this slice too would grow large and unwieldy. For that kind of scale, we would probably run multiple `twittervotes` programs, each dedicated to a portion of the poll data. A simple way to do this would be to break polls into groups based on the letters the titles begin with, such as group A-N and O-Z. A somewhat more sophisticated approach would be to add a field to the `poll` document grouping it up in a more controlled manner, perhaps based on the stats for the other groups so that we are able to balance the load across many `twittervotes` instances.

The `append` built-in function is actually a `variadic` function, which means you can pass multiple elements for it to append. If you have a slice of the correct type, you can add `...` to the end, which simulates the passing of each item of the slice as a different argument.

Finally, we close the iterator and clean up any used memory before returning the options and any errors that occurred while iterating (by calling the `Err` method on the `mgo.Iter` object).

Reading from Twitter

Now we are able to load the options and make authorized requests to the Twitter API. We are thus ready to write the code that initiates the connection, and continuously reads from the stream until either we call our `closeConn` method, or Twitter closes the connection for one reason or another. The structure contained in the stream is a complex one containing all kinds of information about the tweet—who made it and when, and even what links or mentions of users occur in the body (see Twitter's API documentation for more details). However, we are only interested in the tweet text itself so you need not worry about all the other noise; add the following structure to `twitter.go`:

```
type tweet struct {
  Text string
}
```

[This may feel incomplete, but think about how clear it makes our intentions to other programmers who might see our code: a tweet has some text, and that is all we care about.]

Using this new structure, in `twitter.go`, add the following `readFromTwitter` function that takes a send-only channel called `votes`; this is how this function will inform the rest of our program that it has noticed a vote on twitter:

```
func readFromTwitter(votes chan<- string) {
  options, err := loadOptions()
  if err != nil {
    log.Println("failed to load options:", err)
    return
  }
  u, err := url.Parse("https://stream.twitter.com/1.1/statuses/filter.
json")
  if err != nil {
    log.Println("creating filter request failed:", err)
    return
  }
  query := make(url.Values)
  query.Set("track", strings.Join(options, ","))
  req, err := http.NewRequest("POST", u.String(), strings.
NewReader(query.Encode()))
  if err != nil {
    log.Println("creating filter request failed:", err)
    return
  }
  resp, err := makeRequest(req, query)
  if err != nil {
    log.Println("making request failed:", err)
    return
  }
  reader := resp.Body
  decoder := json.NewDecoder(reader)
  for {
    var tweet tweet
    if err := decoder.Decode(&tweet); err != nil {
      break
    }
    for _, option := range options {
      if strings.Contains(
        strings.ToLower(tweet.Text),
        strings.ToLower(option),
      ) {
        log.Println("vote:", option)
        votes <- option
      }
    }
  }
}
```

In the preceding code, after loading the options from all the polls data (by calling the `loadOptions` function), we use `url.Parse` to create a `url.URL` object describing the appropriate endpoint on Twitter. We build a `url.Values` object called `query`, and set the options as a comma-separated list. As per the API, we make a new `POST` request using the encoded `url.Values` object as the body, and pass it to `makeRequest` along with the query object itself. All being well, we make a new `json.Decoder` from the body of the request, and keep reading inside an infinite `for` loop by calling the `Decode` method. If there is an error (probably due to the connection being closed), we simply break the loop and exit the function. If there is a tweet to read, it will be decoded into the `tweet` variable, which will give us access to the `Text` property (the 140 characters of the tweet itself). We then iterate over all possible options, and if the tweet has mentioned it, we send it on the `votes` channel. This technique also allows a tweet to contain many votes at the same time, something you may or may not decide to change based on the rules of the election.

The `votes` channel is **send-only** (which means we cannot receive on it), since it is of the type `chan<- string`. Think of the little "arrow" telling us which way messages will flow: either into the channel or out of it. This is a great way to express intent—it's clear that we never intend to read votes using our `readFromTwitter` function; rather we will only send them on that channel.

Terminating the program whenever `Decode` returns an error doesn't provide a very robust solution. This is because the Twitter API documentation states that the connection will drop from time to time, and clients should consider this when consuming the services. And remember, we are going to terminate the connection periodically too, so we need to think about a way to reconnect once the connection is dropped.

Signal channels

A great use of channels in Go is to signal events between code running in different goroutines. We are going to see a real-world example of this when we write our next function.

The purpose of the function is to start a goroutine that continually calls the `readFromTwitter` function (with the specified `votes` channel to receive the votes on), until we signal that we want it to stop. And once it has stopped, we want to be notified through another signal channel. The return of the function will be a channel of `struct{}`; a signal channel.

Signal channels have some interesting properties that are worth taking a closer look at. Firstly, the type sent down the channels is an empty `struct{}`, instances of which actually take up zero bytes, since it has no fields. So `struct{}{}` is a great memory-efficient option for signaling events. Some people use `bool` types, which is also fine, although `true` and `false` both take up a byte of memory.

> Head over to `http://play.golang.org` and try this out for yourself.
>
> The size of a `bool` is 1:
>
> ```
> fmt.Println(reflect.TypeOf(true).Size())
> = 1
> ```
>
> Whereas the size of `struct{}{}` is `0`:
>
> ```
> fmt.Println(reflect.TypeOf(struct{}{}).Size())
> = 0
> ```

The signal channels also have a buffer size of 1, which means that execution will not block until something reads the signal from the channel.

We are going to employ two signal channels in our code, one that we pass into our function that tells our goroutine that it should stop, and another (provided by the function) that signals once stopping is complete.

In `twitter.go`, add the following function:

```
func startTwitterStream(stopchan <-chan struct{}, votes chan<- string)
<-chan struct{} {
  stoppedchan := make(chan struct{}, 1)
  go func() {
    defer func() {
      stoppedchan <- struct{}{}
    }()
    for {
      select {
      case <-stopchan:
        log.Println("stopping Twitter...")
        return
      default:
```

```
                log.Println("Querying Twitter...")
                readFromTwitter(votes)
                log.Println("  (waiting)")
                time.Sleep(10 * time.Second) // wait before reconnecting
            }
        }
    }()
    return stoppedchan
}
```

In the preceding code, the first argument `stopchan` is a channel of type `<-chan struct{}`, a **receive-only** signal channel. It is this channel that, outside the code, will signal on, which will tell our goroutine to stop. Remember that it's receive-only inside this function, the actual channel itself will be capable of sending. The second argument is the `votes` channel on which votes will be sent. The return type of our function is also a signal channel of type `<-chan struct{}`; a receive-only channel that we will use to indicate that we have stopped.

These channels are necessary because our function triggers its own goroutine, and immediately returns, so without this, calling code would have no idea if the spawned code were still running or not.

The first thing we do in the `startTwitterStream` function is make our `stoppedchan`, and defer the sending of a `struct{}{}` to indicate that we have finished when our function exits. Notice that `stoppedchan` is a normal channel so even though it is returned as a receive-only, we will be able to send on it from within this function.

We then start an infinite `for` loop in which we select from one of two channels. The first is the `stopchan` (the first argument), which would indicate that it was time to stop, and return (thus triggering the deferred signaling on `stoppedchan`). If that hasn't happened, we will call `readFromTwitter` (passing in the `votes` channel), which will go and load the options from the database and open the connection to Twitter.

When the Twitter connection dies, our code will return here where we sleep for ten seconds using the `time.Sleep` function. This is to give the Twitter API a rest in case it closed the connection due to overuse. Once we've rested, we re-enter the loop and check again on the `stopchan` channel to see if the calling code wants us to stop or not.

To make this flow clear, we are logging out key statements that will not only help us debug our code, but also let us peek into the inner workings of this somewhat complicated mechanism.

Publishing to NSQ

Once our code is successfully noticing votes on Twitter and sending them down the `votes` channel, we need a way to publish them into an NSQ topic; after all, this is the point of the `twittervotes` program.

We will write a function called `publishVotes` that will take the `votes` channel, this time of type `<-chan string` (a receive-only channel) and publish each string that is received from it.

> In our previous functions, the `votes` channel was of type `chan<- string`, but this time it's of the type `<-chan string`. You might think this is a mistake, or even that it means we cannot use the same channel for both but you would be wrong. The channel we create later will be made with `make(chan string)`, neither receive or only send, and can act in both cases. The reason for using the `<-` operator on a channel in arguments is to make clear the intent of what the channel will be used for; or in the case where it is the return type, to prevent users from accidentally sending on channels intended for receiving or vice versa. The compiler will actually produce an error if they use such a channel incorrectly.

Once the `votes` channel is closed (this is how external code will tell our function to stop working), we will stop publishing and send a signal down the returned stop signal channel.

Add the `publishVotes` function to `main.go`:

```
func publishVotes(votes <-chan string) <-chan struct{} {
  stopchan := make(chan struct{}, 1)
  pub, _ := nsq.NewProducer("localhost:4150", nsq.NewConfig())
  go func() {
    for vote := range votes {
      pub.Publish("votes", []byte(vote)) // publish vote
    }
    log.Println("Publisher: Stopping")
    pub.Stop()
    log.Println("Publisher: Stopped")
    stopchan <- struct{}{}
  }()
  return stopchan
}
```

Again the first thing we do is to create the `stopchan`, which we later return, this time not deferring the signaling but doing it inline by sending a `struct{}{}` down `stopchan`.

[The difference is to show alternative options: within one codebase you should pick a style you like and stick with it, until a standard emerges within the community; in which case we should all go with that.]

We then create an NSQ producer by calling `NewProducer` and connecting to the default NSQ port on `localhost`, using a default configuration. We start a goroutine, which uses another great built-in feature of the Go language that lets us continually pull values from a channel (in our case the `votes` channel) just by doing a normal `for…range` operation on it. Whenever the channel has no values, execution will be blocked until a value comes down the line. If the `votes` channel is closed, the `for` loop will exit.

[To learn more about the power of channels in Go, it is highly recommended that you seek out blog posts and videos by John Graham-Cumming, in particular one entitled *A Channel Compendium* that he presented at Gophercon 2014 and which contains a brief history of channels, including their origin. (Interestingly, John was also the guy who successfully petitioned the British Government to officially apologize for its treatment of Alan Turing.)]

When the loop exits (after the `votes` channel is closed) the publisher is stopped, following which the `stopchan` signal is sent.

Gracefully starting and stopping

When our program is terminated, we want to do a few things before actually exiting; namely closing our connection to Twitter and stopping the NSQ publisher (which actually deregisters its interest in the queue). To achieve this, we have to override the default *Ctrl* + *C* behavior.

[The upcoming code blocks all go inside the main function; they are broken up so we can discuss each section before continuing.]

Add the following code inside the `main` function:

```
var stoplock sync.Mutex
stop := false
stopChan := make(chan struct{}, 1)
```

```
signalChan := make(chan os.Signal, 1)
go func() {
  <-signalChan
  stoplock.Lock()
  stop = true
  stoplock.Unlock()
  log.Println("Stopping...")
  stopChan <- struct{}{}
  closeConn()
}()
signal.Notify(signalChan, syscall.SIGINT, syscall.SIGTERM)
```

Here we create a stop `bool` with an associated `sync.Mutex` so that we can access it from many goroutines at the same time. We then create two more signal channels, `stopChan` and `signalChan`, and use `signal.Notify` to ask Go to send the signal down the `signalChan` when someone tries to halt the program (either with the `SIGINT` interrupt, or the `SIGTERM` termination POSIX signals). The `stopChan` is how we indicate that we want our processes to terminate, and we pass it as an argument to `startTwitterStream` later.

We then run a goroutine that blocks waiting for the signal by trying to read from `signalChan`; this is what the `<-` operator does in this case (it's trying to read from the channel). Since we don't care about the type of signal, we don't bother capturing the object returned on the channel. Once a signal is received, we set `stop` to `true`, and close the connection. Only when one of the specified signals is sent will the rest of the goroutine code run, which is how we are able to perform teardown code before exiting the program.

Add the following piece of code (inside the main function) to open and defer the closing of the database connection:

```
if err := dialdb(); err != nil {
  log.Fatalln("failed to dial MongoDB:", err)
}
defer closedb()
```

Since the `readFromTwitter` method reloads the options from the database each time, and because we want to keep our program updated without having to restart it, we are going to introduce one final goroutine. This goroutine will simply call `closeConn` every minute, causing the connection to die, and cause `readFromTwitter` to be called over again. Insert the following code at the bottom of the `main` function to start all of these processes, and then wait for them to gracefully stop:

```
// start things
votes := make(chan string) // chan for votes
```

```
publisherStoppedChan := publishVotes(votes)
twitterStoppedChan := startTwitterStream(stopChan, votes)
go func() {
  for {
    time.Sleep(1 * time.Minute)
    closeConn()
    stoplock.Lock()
    if stop {
      stoplock.Unlock()
      break
    }
    stoplock.Unlock()
  }
}()
<-twitterStoppedChan
close(votes)
<-publisherStoppedChan
```

First we make the `votes` channel that we have been talking about throughout this section, which is a simple channel of string. Notice that it is neither a send (`chan<-`) or receive (`<-chan`) channel; in fact, making such channels makes little sense. We then call `publishVotes`, passing in the `votes` channel for it to receive from, and capturing the returned stop signal channel as `publisherStoppedChan`. Similarly, we call `startTwitterStream` passing in our `stopChan` from the beginning of the `main` function, and the `votes` channel for it to send to, while capturing the resulting stop signal channel as `twitterStoppedChan`.

We then start our refresher goroutine, which immediately enters an infinite `for` loop before sleeping for a minute and closing the connection via the call to `closeConn`. If the stop `bool` has been set to true (in that previous goroutine), we will `break` the loop and exit, otherwise we will loop around and wait another minute before closing the connection again. The use of the `stoplock` is important because we have two goroutines that might try to access the stop variable at the same time but we want to avoid collisions.

Once the goroutine has started, we then block on the `twitterStoppedChan` by attempting to read from it. When successful (which means the signal was sent on the `stopChan`), we close the `votes` channel which will cause the publisher's `for…range` loop to exit, and the publisher itself to stop, after which the signal will be sent on the `publisherStoppedChan`, which we wait for before exiting.

Testing

To make sure our program works, we need to do two things: first we need to create a poll in the database, and second, we need to peer inside the messaging queue to see if the messages are indeed being generated by `twittervotes`.

In a terminal, run the `mongo` command to open a database shell that allows us to interact with MongoDB. Then enter the following commands to add a test poll:

```
> use ballots
switched to db ballots
> db.polls.insert({"title":"Test poll","options":["happy","sad","fail","w
in"]})
```

The preceding commands add a new item to the `polls` collection in the `ballots` database. We are using some common words for options that are likely to be mentioned by people on Twitter so that we can observe real tweets being translated into messages. You might notice that our poll object is missing the `results` field; this is fine since we are dealing with unstructured data where documents do not have to adhere to a strict schema. The `counter` program we are going to write in the next section will add and maintain the `results` data for us later.

Press *Ctrl* + *C* to exit the MongoDB shell and type the following command:

```
nsq_tail --topic="votes" --lookupd-http-address=localhost:4161
```

The `nsq_tail` tool connects to the specified messaging queue topic and outputs any messages that it notices. This is where we will validate that our `twittervotes` program is sending messages.

In a separate terminal window, let's build and run the `twittervotes` program:

```
go build -o twittervotes
./twittervotes
```

Now switch back to the window running `nsq_tail` and notice that messages are indeed being generated in response to live Twitter activity.

[If you aren't seeing much activity, try looking up trending hashtags on Twitter and adding another poll containing those options.]

Counting votes

The second program we are going to implement is the `counter` tool, which will be responsible for watching out for votes in NSQ, counting them, and keeping MongoDB up to date with the latest numbers.

Create a new folder called `counter` alongside `twittervotes`, and add the following code to a new `main.go` file:

```
package main
import (
  "flag"
  "fmt"
  "os"
)
var fatalErr error
func fatal(e error) {
  fmt.Println(e)
  flag.PrintDefaults()
  fatalErr = e
}
func main() {
  defer func() {
    if fatalErr != nil {
      os.Exit(1)
    }
  }()
}
```

Normally when we encounter an error in our code, we use a call like `log.Fatal` or `os.Exit`, which immediately terminates the program. Exiting the program with a non-zero exit code is important, because it is our way of telling the operating system that something went wrong, and we didn't complete our task successfully. The problem with the normal approach is that any deferred functions we have scheduled (and therefore any tear down code we need to run), won't get a chance to execute.

The pattern employed in the preceding code snippet lets us call the `fatal` function to record that an error occurred. Note that only when our main function exits will the deferred function run, which in turn calls `os.Exit(1)` to exit the program with an exit code of `1`. Because the deferred statements are run in LIFO (last in, first out) order, the first function we defer will be the last function to be executed, which is why the first thing we do in the `main` function is to defer the exiting code. This allows us to be sure that other functions we defer will be called *before* the program exits. We'll use this feature to ensure our database connection gets closed regardless of any errors.

Connecting to the database

The best time to think about cleaning up resources, such as database connections, is immediately after you have successfully obtained the resource; Go's `defer` keyword makes this easy. At the bottom of the main function, add the following code:

```
log.Println("Connecting to database...")
db, err := mgo.Dial("localhost")
if err != nil {
  fatal(err)
  return
}
defer func() {
  log.Println("Closing database connection...")
  db.Close()
}()
pollData := db.DB("ballots").C("polls")
```

This code uses the familiar `mgo.Dial` method to open a session to the locally running MongoDB instance and immediately defers a function that closes the session. We can be sure that this code will run before our previously deferred statement containing the exit code (because deferred functions are run in the reverse order in which they were called). Therefore, whatever happens in our program, we know that the database session will definitely and properly close.

> The log statements are optional, but will help us see what's going on when we run and exit our program.

At the end of the snippet, we use the `mgo` fluent API to keep a reference of the `ballots.polls` data collection in the `pollData` variable, which we will use later to make queries.

Consuming messages in NSQ

In order to count the votes, we need to consume the messages on the `votes` topic in NSQ, and we'll need a place to store them. Add the following variables to the `main` function:

```
var counts map[string]int
var countsLock sync.Mutex
```

A map and a lock (sync.Mutex) is a common combination in Go, because we will have multiple goroutines trying to access the same map and we need to avoid corrupting it by trying to modify or read it at the same time.

Add the following code to the main function:

```
log.Println("Connecting to nsq...")
q, err := nsq.NewConsumer("votes", "counter", nsq.NewConfig())
if err != nil {
  fatal(err)
  return
}
```

The NewConsumer function allows us to set up an object that will listen on the votes NSQ topic, so when twittervotes publishes a vote on that topic, we can handle it in this program. If NewConsumer returns an error, we'll use our fatal function to record it and return.

Next we are going to add the code that handles messages (votes) from NSQ:

```
q.AddHandler(nsq.HandlerFunc(func(m *nsq.Message) error {
  countsLock.Lock()
  defer countsLock.Unlock()
  if counts == nil {
    counts = make(map[string]int)
  }
  vote := string(m.Body)
  counts[vote]++
  return nil
}))
```

We call the AddHandler method on nsq.Consumer and pass it a function that will be called for every message received on the votes topic.

When a vote comes in, the first thing we do is lock the countsLock mutex. Next we defer the unlocking of the mutex for when the function exits. This allows us to be sure that while NewConsumer is running, we are the only ones allowed to modify the map; others will have to wait until our function exits before they can use it. Calls to the Lock method block execution while the lock is in place, and it only continues when the lock is released by a call to Unlock. This is why it's vital that every Lock call has an Unlock counterpart, otherwise we will deadlock our program.

Every time we receive a vote, we check if counts is nil and make a new map if it is, because once the database has been updated with the latest results, we want to reset everything and start at zero. Finally we increase the int value by one for the given key, and return nil indicating no errors.

Although we have created our NSQ consumer, and added our handler function, we still need to connect to the NSQ service, which we will do by adding the following code:

```
if err := q.ConnectToNSQLookupd("localhost:4161"); err != nil {
  fatal(err)
  return
}
```

It is important to note that we are actually connecting to the HTTP port of the `nsqlookupd` instance, rather than NSQ instances; this abstraction means that our program doesn't need to know *where* the messages are coming from in order to consume them. If we fail to connect to the server (for instance if we forget to start it), we'll get an error, which we report to our fatal function before immediately returning.

Keeping the database updated

Our code will listen out for votes, and keep a map of the results in memory, but that information is so far trapped inside our program. Next, we need to add the code that will periodically push the results to the database:

```
log.Println("Waiting for votes on nsq...")
var updater *time.Timer
updater = time.AfterFunc(updateDuration, func() {
  countsLock.Lock()
  defer countsLock.Unlock()
  if len(counts) == 0 {
    log.Println("No new votes, skipping database update")
  } else {
    log.Println("Updating database...")
    log.Println(counts)
    ok := true
    for option, count := range counts {
      sel := bson.M{"options": bson.M{"$in": []string{option}}}
      up := bson.M{"$inc": bson.M{"results." + option: count}}
      if _, err := pollData.UpdateAll(sel, up); err != nil {
        log.Println("failed to update:", err)
        ok = false
      }
    }
    if ok {
      log.Println("Finished updating database...")
```

```
          counts = nil // reset counts
        }
      }
      updater.Reset(updateDuration)
    })
```

The `time.AfterFunc` function calls the function after the specified duration in a goroutine of its own. At the end we call `Reset`, which starts the process again; this allows us to schedule our update code to run at regular intervals.

When our update function runs, the first thing we do is lock the `countsLock`, and defer its unlocking. We then check to see if there are any values in the counts map. If there aren't, we just log that we're skipping the update and wait for next time.

If there are some votes, we iterate over the `counts` map pulling out the option and number of votes (since the last update), and use some MongoDB magic to update the results.

MongoDB stores BSON (short for Binary JSON) documents internally, which are easier to traverse than normal JSON documents, and is why the `mgo` package comes with `mgo/bson` encoding package. When using `mgo`, we will often use `bson` types, such as the `bson.M` map to describe concepts for MongoDB.

We first create the selector for our update operation using the `bson.M` shortcut type, which is similar to creating `map[string]interface{}` types. The selector we create will look something like this:

```
    {
      "options": {
        "$in": ["happy"]
      }
    }
```

In MongoDB, the preceding BSON specifies that we want to select polls where `"happy"` is one of the items in the `options` array.

Next, we use the same technique to generate the update operation, which looks something like this:

```
    {
      "$inc": {
        "results.happy": 3
      }
    }
```

In MongoDB, the preceding BSON specifies that we want to increase the `results.happy` field by 3. If there is no `results` map in the poll, one will be created, and if there is no `happy` key inside `results`, `0` will be assumed.

We then call the `UpdateAll` method on our `pollsData` query to issue the command to the database, which will in turn update every poll that matches the selector (contrast this to the `Update` method, which will update only one). If something goes wrong, we report it and set the `ok` Boolean to false. If all goes well, we set the `counts` map to nil, since we want to reset the counter.

We are going to specify the `updateDuration` as a constant at the top of the file, which will make it easy for us to change when we are testing our program. Add the following code above the `main` function:

```
const updateDuration = 1 * time.Second
```

Responding to Ctrl + C

The last thing to do before our program is ready is to make sure our `main` function waits for operations to complete before exiting, like we did in our `twittervotes` program. Add the following code at the end of the `main` function:

```
termChan := make(chan os.Signal, 1)
signal.Notify(termChan, syscall.SIGINT, syscall.SIGTERM, syscall.
SIGHUP)
for {
  select {
  case <-termChan:
    updater.Stop()
    q.Stop()
  case <-q.StopChan:
    // finished
    return
  }
}
```

Here we have employed a slightly different tactic than before. We trap the termination event, which will cause a signal to go down `termChan` when we hit *Ctrl* + *C*. Next we start an infinite loop, inside which we use Go's `select` structure to allow us to run code if we receive something on either `termChan`, or the `StopChan` of the consumer.

In fact, we will only ever get a `termChan` signal first in response to a `Ctrl+C`-press, at which point we stop the `updater` timer and ask the consumer to stop listening for votes. Execution then re-enters the loop and will block until the consumer reports that it has indeed stopped by signaling on its `StopChan`. When that happens, we're done and we exit, at which point our deferred statement runs, which, if you remember, tidies up the database session.

Running our solution

It's time to see our code in action. Be sure to have `nsqlookupd`, `nsqd`, and `mongod` running in separate terminal windows with:

```
nsqlookupd
nsqd --lookupd-tcp-address=127.0.0.1:4160
mongod --dbpath ./db
```

If you haven't already done so, make sure the `twittervotes` program is running too. Then in the `counter` folder, build and run our counting program:

```
go build -o counter
./counter
```

You should see periodic output describing what work `counter` is doing, such as:

```
No new votes, skipping database update
Updating database...
map[win:2 happy:2 fail:1]
Finished updating database...
No new votes, skipping database update
Updating database...
map[win:3]
Finished updating database...
```

The output will of course vary since we are actually responding to real live activity on Twitter.

We can see that our program is receiving vote data from NSQ, and reports to be updating the database with the results. We can confirm this by opening the MongoDB shell and querying the poll data to see if the `results` map is being updated. In another terminal window, open the MongoDB shell:

```
mongo
```

Ask it to use the ballots database:

```
> use ballots
switched to db ballots
```

Use the find method with no arguments to get all polls (add the `pretty` method to the end to get nicely formatted JSON):

```
> db.polls.find().pretty()
{
       "_id" : ObjectId("53e2a3afffbff195c2e09a02"),
       "options" : [
              "happy","sad","fail","win"
       ],
       "results" : {
              "fail" : 159, "win" : 711,
              "happy" : 233, "sad" : 166,
       },
       "title" : "Test poll"
}
```

The `results` map is indeed being updated, and at any point in time contains the total number of votes for each option.

Summary

In this chapter we covered a lot of ground. We learned different techniques for gracefully shutting down programs using signaling channels, which is especially important when our code has some work to do before it can exit. We saw that deferring the reporting of fatal errors at the start of our program can give our other deferred functions a chance to execute before the process ends.

We also discovered how easy it is to interact with MongoDB using the `mgo` package, and how to use BSON types when describing concepts for the database. The `bson.M` alternative to `map[string]interface{}` helps us keep our code more concise, while still providing all the flexibility we need when working with unstructured or schemaless data.

We learned about message queues and how they allow us to break apart the components of a system into isolated and specialized micro-services. We started an instance of NSQ by first running the lookup daemon `nsqlookupd`, before running a single `nsqd` instance and connecting them together via a TCP interface. We were then able to publish votes to the queue in `twittervotes`, and connect to the lookup daemon to run a handler function for every vote sent in our `counter` program.

While our solution is actually performing a pretty simple task, the architecture we have put together in this chapter is capable of doing some pretty great things.

- We eliminated the need for our `twittervotes` and `counter` programs to run on the same machine—as long as they can both connect to the appropriate NSQ, they will function as expected regardless of where they are running.
- We can distribute our MongoDB and NSQ nodes across many physical machines which would mean our system is capable of gigantic scale—whenever resources start running low, we can add new boxes to cope with the demand.
- When we add other applications that need to query and read the results from polls, we can be sure that our database services are highly available and capable of delivering.
- We can spread our database across geographical expanses replicating data for backup so we don't lose anything when disaster strikes.
- We can build a multi-node, fault tolerant NSQ environment, which means when our `twittervotes` program learns of interesting tweets, there will always be somewhere to send the data.
- We could write many more programs that generate votes from different sources; the only requirement is that they know how to put messages into NSQ.
- In the next chapter, we will build a RESTful data service of our own, through which we will expose the functionality of our social polling application. We will also build a web interface that lets users create their own polls, and visualize the results.

6
Exposing Data and Functionality through a RESTful Data Web Service API

In the previous chapter, we built a service that reads tweets from Twitter, counts the hashtag votes, and stores the results in a MongoDB database. We also used the MongoDB shell to add polls and see the poll results. This approach is fine if we are the only ones using our solution, but it would be madness if we released our project and expected users to connect directly to our MongoDB instance in order to use the service we built.

Therefore, in this chapter, we are going to build a RESTful data service through which the data and functionality will be exposed. We will also put together a simple website that consumes the new API. Users may then either use our website to create and monitor polls or build their own application on top of the web services we release.

[The code in this chapter depends on the code in *Chapter 5*, *Building Distributed Systems and Working with Flexible Data*, so it is recommended that you complete that chapter first, especially since it covers setting up the environment that the code in this chapter runs on.]

Specifically, you will learn:

- How wrapping `http.HandlerFunc` types can give us a simple but powerful pipeline of execution for our HTTP requests
- How to safely share data between HTTP handlers

- Best practices for writing handlers responsible for exposing data
- Where small abstractions can allow us to write the simplest possible implementations now, but leave room to improve them later without changing the interface
- How adding simple helper functions and types to our project will prevent us from (or at least defer) adding dependencies on external packages

RESTful API design

For an API to be considered RESTful, it must adhere to a few principles that stay true to the original concepts behind the Web, and are already known to most developers. Such an approach allows us to make sure we aren't building anything strange or unusual into our API while also giving our users a head start towards consuming it, since they are already familiar with its concepts.

Some of the most important RESTful design concepts are:

- HTTP methods describe the kind of action to take, for example, `GET` methods will only ever *read* data, while `POST` requests will *create* something
- Data is expressed as a collection of resources
- Actions are expressed as changes to data
- URLs are used to refer to specific data
- HTTP headers are used to describe the kind of representation coming into and going out of the server

For an in-depth overview of these and other details of RESTful designs, see the Wikipedia article at `http://en.wikipedia.org/wiki/Representational_state_transfer`.

The following table shows the HTTP methods and URLs that represent the actions that we will support in our API, along with a brief description and an example use case of how we intend the call to be used:

Request	**Description**	**Use case**
`GET /polls/`	Read all polls	Show a list of polls to the users
`GET /polls/{id}`	Read the poll	Show details or results of a specific poll
`POST /polls/`	Create a poll	Create a new poll
`DELETE /polls/{id}`	Delete a poll	Delete a specific poll

The `{id}` placeholder represents where in the path the unique ID for a poll will go.

Sharing data between handlers

If we want to keep our handlers as pure as the `http.Handler` interface from the Go standard library, while still extracting common functionality into our own methods, we need a way of sharing data between handlers. The `HandlerFunc` signature that follows tells us that we are only allowed to pass in an `http.ResponseWriter` object and an `http.Request` object, and nothing else:

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

This means that we cannot create and manage database session objects in one place and pass them into our handlers, which is ideally what we want to do.

Instead, we are going to implement an in-memory map of per-request data, and provide an easy way for handlers to access it. Alongside the `twittervotes` and `counter` folders, create a new folder called `api` and create a new file called `vars.go` inside it. Add the following code to the file:

```
package main
import (
  "net/http"
  "sync"
)
var vars map[*http.Request]map[string]interface{}
var varsLock sync.RWMutex
```

Here we declare a `vars` map that has a pointer to an `http.Request` type as its key, and another map as the value. We will store the map of variables keyed with the request instances that the variables belong to. The `varsLock` mutex is important, as our handlers will all be trying to access and change the `vars` map at the same time as handling many concurrent HTTP requests, and we need to ensure that they do this safely.

Next we are going to add the `OpenVars` function that allows us to prepare the `vars` map to hold variables for a particular request:

```
func OpenVars(r *http.Request) {
  varsLock.Lock()
  if vars == nil {
    vars = map[*http.Request]map[string]interface{}{}
  }
  vars[r] = map[string]interface{}{}
  varsLock.Unlock()
}
```

This function first locks the mutex so that we can safely modify the map, before ensuring that `vars` contains a non-nil map, which would otherwise cause a panic when we try to access its data. Finally, it assigns a new empty `map` value using the specified `http.Request` pointer as the key, before unlocking the mutex and therefore freeing other handlers to interact with it.

Once we have finished handling the request, we need a way to clean up the memory that we are using here; otherwise the memory footprint of our code would continuously increase (also known as a memory leak). We do this by adding a `CloseVars` function:

```
func CloseVars(r *http.Request) {
  varsLock.Lock()
  delete(vars, r)
  varsLock.Unlock()
}
```

This function safely deletes the entry in the `vars` map for the request. As long as we call `OpenVars` before we try to interact with the variables, and `CloseVars` when we have finished, we will be free to safely store and retrieve data for each request. However, we don't want our handler code to have to worry about locking and unlocking the map whenever it needs to get or set some data, so let's add two helper functions, `GetVar` and `SetVar`:

```
func GetVar(r *http.Request, key string) interface{} {
  varsLock.RLock()
  value := vars[r][key]
  varsLock.RUnlock()
  return value
}
func SetVar(r *http.Request, key string, value interface{}) {
  varsLock.Lock()
  vars[r][key] = value
  varsLock.Unlock()
}
```

The `GetVar` function will make it easy for us to get a variable from the map for the specified request, and `SetVar` allows us to set one. Notice that the `GetVar` function calls `RLock` and `RUnlock` rather than `Lock` and `Unlock`; this is because we're using `sync.RWMutex`, which means it's safe for many reads to occur at the same time, as long as a write isn't happening. This is good for performance on items that are safe to concurrently read from. With a normal mutex, `Lock` would block execution—waiting for the thing that has locked it to unlock it—while `RLock` will not.

Wrapping handler functions

One of the most valuable patterns to learn when building web services and websites in Go is one we already utilized in *Chapter 2*, *Adding Authentication*, where we decorated `http.Handler` types by wrapping them with other `http.Handler` types. For our RESTful API, we are going to apply this same technique to `http.HandlerFunc` functions, to deliver an extremely powerful way of modularizing our code without breaking the standard `func(w http.ResponseWriter, r *http.Request)` interface.

API key

Most web APIs require clients to register an API key for their application, which they are asked to send along with every request. Such keys have many purposes, ranging from simply identifying which app the requests are coming from to addressing authorization concerns in situations where some apps are only able to do limited things based on what a user has allowed. While we don't actually need to implement API keys for our application, we are going to ask clients to provide one, which will allow us to add an implementation later while keeping the interface constant.

Add the essential `main.go` file inside your `api` folder:

```
package main
func main(){}
```

Next we are going to add our first `HandlerFunc` wrapper function called `withAPIKey` to the bottom of `main.go`:

```
func withAPIKey(fn http.HandlerFunc) http.HandlerFunc {
  return func(w http.ResponseWriter, r *http.Request) {
    if !isValidAPIKey(r.URL.Query().Get("key")) {
      respondErr(w, r, http.StatusUnauthorized, "invalid API key")
      return
    }
    fn(w, r)
  }
}
```

As you can see, our `withAPIKey` function both takes an `http.HandlerFunc` type as an argument and returns one; this is what we mean by wrapping in this context. The `withAPIKey` function relies on a number of other functions that we are yet to write, but you can clearly see what's going on. Our function immediately returns a new `http.HandlerFunc` type that performs a check for the query parameter `key` by calling `isValidAPIKey`. If the key is deemed invalid (by the return of `false`), we respond with an `invalid API key` error. To use this wrapper, we simply pass an `http.HandlerFunc` type into this function to enable the `key` parameter check. Since it returns an `http.HandlerFunc` type too, the result can then be passed into other wrappers or given directly to the `http.HandleFunc` function to actually register it as the handler for a particular path pattern.

Let's add our `isValidAPIKey` function next:

```
func isValidAPIKey(key string) bool {
  return key == "abc123"
}
```

For now, we are simply going to hardcode the API key as `abc123`; anything else will return `false` and therefore be considered invalid. Later we could modify this function to consult a configuration file or database to check the authenticity of a key without affecting how we use the `isValidAPIKey` method, or indeed the `withAPIKey` wrapper.

Database session

Now that we can be sure a request has a valid API key, we must consider how handlers will connect to the database. One option is to have each handler dial its own connection, but this isn't very **DRY** (**Don't Repeat Yourself**), and leaves room for potentially erroneous code, such as code that forgets to close a database session once it is finished with it. Instead, we will create another `HandlerFunc` wrapper that manages the database session for us. In `main.go`, add the following function:

```
func withData(d *mgo.Session, f http.HandlerFunc) http.HandlerFunc {
  return func(w http.ResponseWriter, r *http.Request) {
    thisDb := d.Copy()
    defer thisDb.Close()
```

```
      SetVar(r, "db", thisDb.DB("ballots"))
      f(w, r)
    }
  }
```

The withData function takes a MongoDB session representation using the mgo package, and another handler as per the pattern. The returned http.HandlerFunc type will copy the database session, defer the closing of that copy, and set a reference to the ballots database as the db variable using our SetVar helper, before finally calling the next HandlerFunc. This means that any handlers that get executed after this one will have access to a managed database session via the GetVar function. Once the handlers have finished executing, the deferred closing of the session will occur, which will clean up any memory used by the request without the individual handlers having to worry about it.

Per request variables

Our pattern allows us to very easily perform common tasks on behalf of our actual handlers. Notice that one of the handlers is calling OpenVars and CloseVars so that GetVar and SetVar may be used without individual handlers having to concern themselves with setting things up and tearing them down. The function will return an http.HandlerFunc that first calls OpenVars for the request, defers the calling of CloseVars, and calls the specified handler function. Any handlers wrapped with withVars will be able to use GetVar and SetVar.

Add the following code to main.go:

```
  func withVars(fn http.HandlerFunc) http.HandlerFunc {
    return func(w http.ResponseWriter, r *http.Request) {
      OpenVars(r)
      defer CloseVars(r)
      fn(w, r)
    }
  }
```

There are lots of other problems that can be addressed using this pattern; and whenever you find yourself duplicating common tasks inside handlers, it's worth considering whether a handler wrapper function could help simplify code.

Cross-browser resource sharing

The same-origin security policy mandates that AJAX requests in web browsers be only allowed for services hosted on the same domain, which would make our API fairly limited since we won't be necessarily hosting all of the websites that use our web service. The CORS technique circumnavigates the same-origin policy, allowing us to build a service capable of serving websites hosted on other domains. To do this, we simply have to set the `Access-Control-Allow-Origin` header in response to `*`. While we're at it—since we're using the `Location` header in our create poll call—we'll allow that header to be accessible by the client too, which can be done by listing it in the `Access-Control-Expose-Headers` header. Add the following code to `main.go`:

```
func withCORS(fn http.HandlerFunc) http.HandlerFunc {
  return func(w http.ResponseWriter, r *http.Request) {
    w.Header().Set("Access-Control-Allow-Origin", "*")
    w.Header().Set("Access-Control-Expose-Headers", "Location")
    fn(w, r)
  }
}
```

This is the simplest wrapper function yet; it just sets the appropriate header on the `ResponseWriter` type and calls the specified `http.HandlerFunc` type.

> In this chapter, we are handling CORS explicitly so we can understand exactly what is going on; for real production code, you should consider employing an open source solution such as `https://github.com/fasterness/cors`.

Responding

A big part of any API is responding to requests with a combination of status codes, data, errors, and sometimes headers—the `net/http` package makes all of this very easy to do. One option we have, which remains the best option for tiny projects or even the early stages of big projects, is to just build the response code directly inside the handler. As the number of handlers grows, however, we would end up duplicating a lot of code and sprinkling representation decisions all over our project. A more scalable approach is to abstract the response code into helper functions.

For the first version of our API, we are going to speak only JSON, but we want the flexibility to add other representations later if we need to.

Create a new file called `respond.go`, and add the following code:

```
func decodeBody(r *http.Request, v interface{}) error {
  defer r.Body.Close()
  return json.NewDecoder(r.Body).Decode(v)
}
func encodeBody(w http.ResponseWriter, r *http.Request, v
interface{}) error {
  return json.NewEncoder(w).Encode(v)
}
```

These two functions abstract the decoding and encoding of data from and to the `Request` and `ResponseWriter` objects respectively. The decoder also closes the request body, which is recommended. Although we haven't added much functionality here, it means that we do not need to mention JSON anywhere else in our code, and if we decide to add support for other representations or switch to a binary protocol instead, we need only touch these two functions.

Next we are going to add a few more helpers that will make responding even easier. In `respond.go`, add the following code:

```
func respond(w http.ResponseWriter, r *http.Request,
  status int, data interface{},
) {
  w.WriteHeader(status)
  if data != nil {
    encodeBody(w, r, data)
  }
}
```

This function makes it easy to write the status code and some data to the `ResponseWriter` object using our `encodeBody` helper.

Handling errors is another important aspect that is worth abstracting. Add the following `respondErr` helper:

```
func respondErr(w http.ResponseWriter, r *http.Request,
  status int, args ...interface{},
) {
  respond(w, r, status, map[string]interface{}{
    "error": map[string]interface{}{
      "message": fmt.Sprint(args...),
    },
  })
}
```

This method gives us an interface similar to the `respond` function, but the data written will be enveloped in an `error` object, to make it clear that something went wrong. Finally, we can add an HTTP error-specific helper that will generate the correct message for us by using the `http.StatusText` function from the Go standard library:

```
func respondHTTPErr(w http.ResponseWriter, r *http.Request,
  status int,
) {
  respondErr(w, r, status, http.StatusText(status))
}
```

Notice that these functions are all dogfooding, which means they use each other (as in, eating your own dog food), which is important since we want actual responding to only happen in one place, for if (or more likely, when) we need to make changes.

Understanding the request

The `http.Request` object gives us access to every piece of information we might need about the underlying HTTP request, and therefore it is worth glancing through the `net/http` documentation to really get a feel for its power. Examples include, but are not limited to:

- URL, path and query string
- HTTP method
- Cookies
- Files
- Form values
- Referrer and user agent of requester
- Basic authentication details
- Request body
- Header information

There are a few things it doesn't address, which we need to either solve ourselves or look to an external package to help us with. URL path parsing is one such example—while we can access a path (such as `/people/1/books/2`) as a string via the `http.Request` type's `URL.Path` field, there is no easy way to pull out the data encoded in the path such as the people ID of `1`, or the books ID of `2`.

A few projects do a good job of addressing this problem, such as Goweb or Gorillz's `mux` package. They let you map path patterns that contain placeholders for values that they then pull out of the original string and make available to your code. For example, you can map a pattern of `/users/{userID}/comments/{commentID}`, which will map paths such as `/users/1/comments/2`. In your handler code, you can then get the values by the names placed inside the curly braces, rather than having to parse the path yourself.

Since our needs are simple, we are going to knock together a simple path-parsing utility; we can always use a different package later if we have to, but that would mean adding a dependency to our project.

Create a new file called `path.go`, and insert the following code:

```
package main
import (
  "strings"
)
const PathSeparator = "/"
type Path struct {
  Path string
  ID   string
}
func NewPath(p string) *Path {
  var id string
  p = strings.Trim(p, PathSeparator)
  s := strings.Split(p, PathSeparator)
  if len(s) > 1 {
    id = s[len(s)-1]
    p = strings.Join(s[:len(s)-1], PathSeparator)
  }
  return &Path{Path: p, ID: id}
}
func (p *Path) HasID() bool {
  return len(p.ID) > 0
}
```

This simple parser provides a `NewPath` function that parses the specified path string and returns a new instance of the `Path` type. Leading and trailing slashes are trimmed (using `strings.Trim`) and the remaining path is split (using `strings.Split`) by the `PathSeparator` constant that is just a forward slash. If there is more than one segment (`len(s) > 1`), the last one is considered to be the ID. We re-slice the slice of strings to select the last item for the ID using `s[len(s)-1]`, and the rest of the items for the remainder of the path using `s[:len(s)-1]`. On the same lines, we also re-join the path segments with the `PathSeparator` constant to form a single string containing the path without the ID.

This supports any `collection/id` pair, which is all we need for our API. The following table shows the state of the `Path` type for the given original path string:

Original path string	**Path**	**ID**	**HasID**
`/`	`/`	`nil`	`false`
`/people/`	`people`	`nil`	`false`
`/people/1/`	`people`	`1`	`true`

A simple main function to serve our API

A web service is nothing more than a simple Go program that binds to a specific HTTP address and port and serves requests, so we get to use all our command-line tool-writing knowledge and techniques.

[We also want to ensure that our `main` function is as simple and modest as possible, which is always a goal of coding, especially in Go.]

Before we write our `main` function, let's look at a few design goals of our API program:

- We should be able to specify the HTTP address and port to which our API listens and the address of the MongoDB instances without having to recompile the program (through command-line flags)
- We want the program to gracefully shut down when we terminate it, allowing the in-flight requests (requests that are still being processed when the termination signal is sent to our program) to complete
- We want the program to log out status updates and report errors properly

Atop the `main.go` file, replace the `main` function placeholder with the following code:

```
func main() {
  var (
    addr  = flag.String("addr", ":8080", "endpoint address")
    mongo = flag.String("mongo", "localhost", "mongodb address")
  )
  flag.Parse()
  log.Println("Dialing mongo", *mongo)
  db, err := mgo.Dial(*mongo)
  if err != nil {
    log.Fatalln("failed to connect to mongo:", err)
  }
  defer db.Close()
  mux := http.NewServeMux()
  mux.HandleFunc("/polls/", withCORS(withVars(withData(db,
withAPIKey(handlePolls)))))
  log.Println("Starting web server on", *addr)
  graceful.Run(*addr, 1*time.Second, mux)
  log.Println("Stopping...")
}
```

This function is the entirety of our API `main` function, and even as our API grows, there is just a little bloat we would need to add to this.

The first thing we do is to specify two command-line flags, `addr` and `mongo`, with some sensible defaults, and to ask the `flag` package to parse them. We then attempt to dial the MongoDB database at the specified address. If we are unsuccessful, we abort with a call to `log.Fatalln`. Assuming the database is running and we are able to connect, we store the reference in the `db` variable before deferring the closing of the connection. This ensures our program properly disconnects and tidies up after itself when it ends.

We then create a new `http.ServeMux` object, which is a request multiplexer provided by the Go standard library, and register a single handler for all requests that begin with the path `/polls/`.

Finally, we make use of Tyler Bunnell's excellent `Graceful` package, which can be found at `https://github.com/stretchr/graceful` to start the server. This package allows us to specify `time.Duration` when running any `http.Handler` (such as our `ServeMux` handler), which will allow any in-flight requests some time to complete before the function exits. The `Run` function will block until the program is terminated (for example, when someone presses *Ctrl* + *C*).

Using handler function wrappers

It is when we call `HandleFunc` on the `ServeMux` handler that we are making use of our handler function wrappers, with the line:

```
withCORS(withVars(withData(db, withAPIKey(handlePolls)))))
```

Since each function takes an `http.HandlerFunc` type as an argument and also returns one, we are able to chain the execution just by nesting the function calls as we have done previously. So when a request comes in with a path prefix of `/polls/`, the program will take the following execution path:

1. `withCORS` is called, which sets the appropriate header.
2. `withVars` is called, which calls `OpenVars` and defers `CloseVars` for the request.
3. `withData` is then called, which copies the database session provided as the first argument and defers the closing of that session.
4. `withAPIKey` is called next, which checks the request for an API key and aborts if it's invalid, or else calls the next handler function.
5. `handlePolls` is then called, which has access to variables and a database session, and which may use the helper functions in `respond.go` to write a response to the client.
6. Execution goes back to `withAPIKey` that just exits.
7. Execution goes back to `withData` that exits, therefore calling the deferred session `Close` function and clearing up the database session.
8. Execution goes back to `withVars` that exits, therefore calling `CloseVars` and tidying that up too.
9. Execution finally goes back to `withCORS` that just exits.

The order that we nest the wrapper functions in is important, because `withData` puts the database session for each request in that request's variables map using `SetVar`. So `withVars` must be outside `withData`. If this isn't respected, the code will likely panic and you may want to add a check so that the panic is more meaningful to other developers.

Handling endpoints

The final piece of the puzzle is the `handlePolls` function that will use the helpers to understand the incoming request and access the database, and generate a meaningful response that will be sent back to the client. We also need to model the poll data that we were working with in the previous chapter.

Create a new file called `polls.go`, and add the following code:

```
package main
import "gopkg.in/mgo.v2/bson"
type poll struct {
  ID      bson.ObjectId  `bson:"_id" json:"id"`
  Title   string         `json":"title""`
  Options []string       `json:"options"`
  Results map[string]int `json:"results,omitempty"`
}
```

Here we define a structure called `poll` that has three fields that in turn describe the polls being created and maintained by the code we wrote in the previous chapter. Each field also has a tag (two in the `ID` case), which allows us to provide some extra metadata.

Using tags to add metadata to structs

Tags are strings that follow a field definition within a `struct` type on the same line of code. We use the back tick character to denote literal strings, which means we are free to use double quotes within the tag string itself. The `reflect` package allows us to pull out the value associated with any key; in our case, both `bson` and `json` are examples of keys, and they are each key/value-pair-separated by a space character. Both the `encoding/json` and `gopkg.in/mgo.v2/bson` packages allow you to use tags to specify the field name that will be used with encoding and decoding (along with some other properties), rather than having it infer the values from the name of the fields themselves. We are using BSON to talk with the MongoDB database and JSON to talk to the client, so we can actually specify different views of the same `struct` type. For example, consider the ID field:

```
ID bson.ObjectId `bson:"_id" json:"id"`
```

The name of the field in Go is `ID`, the JSON field is `id`, and the BSON field is `_id`, which is the special identifier field used in MongoDB.

Many operations with a single handler

Because our simple path-parsing solution cares only about the path, we have to do some extra work when looking at the kind of RESTful operation the client is making. Specifically, we need to consider the HTTP method so we know how to handle the request. For example, a `GET` call to our `/polls/` path should read polls, where a `POST` call would create a new one. Some frameworks solve this problem for you, by allowing you to map handlers based on more than the path, such as the HTTP method or the presence of specific headers in the request. Since our case is ultra simple, we are going to use a simple `switch` case. In `polls.go`, add the `handlePolls` function:

```
func handlePolls(w http.ResponseWriter, r *http.Request) {
  switch r.Method {
  case "GET":
    handlePollsGet(w, r)
    return
  case "POST":
    handlePollsPost(w, r)
    return
  case "DELETE":
    handlePollsDelete(w, r)
    return
  }
  // not found
  respondHTTPErr(w, r, http.StatusNotFound)
}
```

We switch on the HTTP method and branch our code depending on whether it is `GET`, `POST`, or `DELETE`. If the HTTP method is something else, we just respond with a `404 http.StatusNotFound` error. To make this code compile, you can add the following function stubs underneath the `handlePolls` handler:

```
func handlePollsGet(w http.ResponseWriter, r *http.Request) {
  respondErr(w, r, http.StatusInternalServerError, errors.New("not
implemented"))
}
func handlePollsPost(w http.ResponseWriter, r *http.Request) {
  respondErr(w, r, http.StatusInternalServerError, errors.New("not
implemented"))
}
func handlePollsDelete(w http.ResponseWriter, r *http.Request) {
  respondErr(w, r, http.StatusInternalServerError, errors.New("not
implemented"))
}
```

[In this section, we learned how to manually parse elements of the requests (the HTTP method) and make decisions in code. This is great for simple cases, but it's worth looking at packages such as Goweb or Gorilla's `mux` package for some more powerful ways of solving these problems. Nevertheless, keeping external dependencies to a minimum is a core philosophy of writing good and contained Go code.]

Reading polls

Now it's time to implement the functionality of our web service. Inside the `GET` case, add the following code:

```
func handlePollsGet(w http.ResponseWriter, r *http.Request) {
  db := GetVar(r, "db").(*mgo.Database)
  c := db.C("polls")
  var q *mgo.Query
  p := NewPath(r.URL.Path)
  if p.HasID() {
    // get specific poll
    q = c.FindId(bson.ObjectIdHex(p.ID))
  } else {
    // get all polls
    q = c.Find(nil)
  }
  var result []*poll
  if err := q.All(&result); err != nil {
    respondErr(w, r, http.StatusInternalServerError, err)
    return
  }
  respond(w, r, http.StatusOK, &result)
}
```

The very first thing we do in each of our subhandler functions is to use `GetVar` to get the `mgo.Database` object that will allow us to interact with MongoDB. Since this handler was nested inside both `withVars` and `withData`, we know that the database will be available by the time execution reaches our handler. We then use `mgo` to create an object referring to the `polls` collection in the database—if you remember, this is where our polls live.

We then build up an `mgo.Query` object by parsing the path. If an ID is present, we use the `FindId` method on the `polls` collection, otherwise we pass `nil` to the `Find` method, which indicates that we want to select all the polls. We are converting the ID from a string to a `bson.ObjectId` type with the `ObjectIdHex` method so that we can refer to the polls with their numerical (hex) identifiers.

Since the `All` method expects to generate a collection of poll objects, we define the result as `[]*poll`, or a slice of pointers to poll types. Calling the `All` method on the query will cause `mgo` to use its connection to MongoDB to read all the polls and populate the `result` object.

For small scale projects, such as a small number of polls, this approach is fine, but as the number of polls grow, we would need to consider paging the results or even iterating over them using the `Iter` method on the query, so we do not try to load too much data into memory.

Now that we have added some functionality, let's try out our API for the first time. If you are using the same MongoDB instance that we set up in the previous chapter, you should already have some data in the `polls` collection; to see our API working properly, you should ensure there are at least two polls in the database.

If you need to add other polls to the database, in a terminal, run the `mongo` command to open a database shell that will allow you to interact with MongoDB. Then enter the following commands to add some test polls:

```
> use ballots
switched to db ballots
> db.polls.insert({"title":"Test
poll","options":["one","two","three"]})
> db.polls.insert({"title":"Test poll
two","options":["four","five","six"]})
```

In a terminal, navigate to your `api` folder, and build and run the project:

```
go build -o api
./api
```

Now make a `GET` request to the `/polls/` endpoint by navigating in your browser to `http://localhost:8080/polls/?key=abc123`; remember to include the trailing slash. The result will be an array of polls in JSON format.

Copy and paste one of the IDs from the polls list, and insert it before the `?` character in the browser to access the data for a specific poll; for example, `http://localhost:8080/polls/5415b060a02cd4adb487c3ae?key=abc123`. Notice that instead of returning all the polls, it only returns one.

Test the API key functionality by removing or changing the key parameter to see what the error looks like.

You might have also noticed that although we are only returning a single poll, this poll value is still nested inside an array. This is a deliberate design decision made for two reasons: the first and most important reason is that nesting makes it easier for users of the API to write code to consume the data. If users are always expecting a JSON array, they can write strong types that describe that expectation, rather than having one type for single polls and another for collections of polls. As an API designer, this is your decision to make. The second reason we left the object nested in an array is that it makes the API code simpler, allowing us to just change the `mgo.Query` object and to leave the rest of the code the same.

Creating a poll

Clients should be able to make a `POST` request to `/polls/` to create a poll. Let's add the following code inside the `POST` case:

```
func handlePollsPost(w http.ResponseWriter, r *http.Request) {
  db := GetVar(r, "db").(*mgo.Database)
  c := db.C("polls")
  var p poll
  if err := decodeBody(r, &p); err != nil {
    respondErr(w, r, http.StatusBadRequest, "failed to read poll
from request", err)
    return
  }
  p.ID = bson.NewObjectId()
  if err := c.Insert(p); err != nil {
    respondErr(w, r, http.StatusInternalServerError, "failed to
insert poll", err)
    return
  }
  w.Header().Set("Location", "polls/"+p.ID.Hex())
  respond(w, r, http.StatusCreated, nil)
}
```

Here we first attempt to decode the body of the request that, according to RESTful principles, should contain a representation of the poll object the client wants to create. If an error occurs, we use the `respondErr` helper to write the error to the user, and immediately return the function. We then generate a new unique ID for the poll, and use the `mgo` package's `Insert` method to send it into the database. As per HTTP standards, we then set the `Location` header of the response and respond with a `201 http.StatusCreated` message, pointing to the URL from which the newly created poll maybe accessed.

Deleting a poll

The final piece of functionality we are going to include in our API is the capability to delete polls. By making a request with the `DELETE` HTTP method to the URL of a poll (such as `/polls/5415b060a02cd4adb487c3ae`), we want to be able to remove the poll from the database and return a `200 Success` response:

```
func handlePollsDelete(w http.ResponseWriter, r *http.Request) {
  db := GetVar(r, "db").(*mgo.Database)
  c := db.C("polls")
  p := NewPath(r.URL.Path)
  if !p.HasID() {
    respondErr(w, r, http.StatusMethodNotAllowed, "Cannot delete
all polls.")
    return
  }
  if err := c.RemoveId(bson.ObjectIdHex(p.ID)); err != nil {
    respondErr(w, r, http.StatusInternalServerError, "failed to
delete poll", err)
    return
  }
  respond(w, r, http.StatusOK, nil) // ok
}
```

Similar to the `GET` case, we parse the path, but this time we respond with an error if the path does not contain an ID. For now, we don't want people to be able to delete all polls with one request, and so use the suitable `StatusMethodNotAllowed` code. Then, using the same collection we used in the previous cases, we call `RemoveId`, passing in the ID in the path after converting it into a `bson.ObjectId` type. Assuming things go well, we respond with an `http.StatusOK` message, with no body.

CORS support

In order for our `DELETE` capability to work over CORS, we must do a little extra work to support the way CORS browsers handle some HTTP methods such as `DELETE`. A CORS browser will actually send a pre-flight request (with an HTTP method of `OPTIONS`) asking for permission to make a `DELETE` request (listed in the `Access-Control-Request-Method` request header), and the API must respond appropriately in order for the request to work. Add another case in the `switch` statement for `OPTIONS`:

```
case "OPTIONS":
  w.Header().Add("Access-Control-Allow-Methods", "DELETE")
  respond(w, r, http.StatusOK, nil)
  return
```

If the browser asks for permission to send a `DELETE` request, the API will respond by setting the `Access-Control-Allow-Methods` header to `DELETE`, thus overriding the default `*` value that we set in our `withCORS` wrapper handler. In the real world, the value for the `Access-Control-Allow-Methods` header will change in response to the request made, but since `DELETE` is the only case we are supporting, we can hardcode it for now.

> The details of CORS are out of the scope of this book, but it is recommended that you research the particulars online if you intend to build truly accessible web services and APIs. Head over to `http://enable-cors.org/` to get started.

Testing our API using curl

`curl` is a command-line tool that allows us to make HTTP requests to our service so that we can access it as though we were a real app or client consuming the service.

> Windows users do not have access to `curl` by default, and will need to seek an alternative. Check out `http://curl.haxx.se/dlwiz/?type=bin` or search the Web for "Windows `curl` alternative".

In a terminal, let's read all the polls in the database through our API. Navigate to your `api` folder and build and run the project, and also ensure MongoDB is running:

```
go build –o api
./api
```

We then perform the following steps:

1. Enter the following `curl` command that uses the `-X` flag to denote we want to make a `GET` request to the specified URL:
   ```
   curl -X GET http://localhost:8080/polls/?key=abc123
   ```
2. The output is printed after you hit *Enter*:
   ```
   [{"id":"541727b08ea48e5e5d5bb189","title":"Best
   Beatle?","options":["john","paul","george","ringo"]},{"id":"54
   1728728ea48e5e5d5bb18a","title":"Favorite
   language?","options":["go","java","javascript","ruby"]}]
   ```
3. While it isn't pretty, you can see that the API returns the polls from your database. Issue the following command to create a new poll:
   ```
   curl --data '{"title":"test","options":["one","two","three"]}'
   -X POST http://localhost:8080/polls/?key=abc123
   ```

4. Get the list again to see the new poll included:

   ```
   curl -X GET http://localhost:8080/polls/?key=abc123
   ```

5. Copy and paste one of the IDs, and adjust the URL to refer specifically to that poll:

   ```
   curl -X GET
   http://localhost:8080/polls/541727b08ea48e5e5d5bb189?key=abc12
   3
   ```

   ```
   [{"id":"541727b08ea48e5e5d5bb189",","title":"Best
   Beatle?","options":["john","paul","george","ringo"]}]
   ```

6. Now we see only the selected poll, `Best Beatle`. Let's make a `DELETE` request to remove the poll:

   ```
   curl -X DELETE
   http://localhost:8080/polls/541727b08ea48e5e5d5bb189?key=abc12
   3
   ```

7. Now when we get all the polls again, we'll see that the `Best Beatle` poll has gone:

   ```
   curl -X GET http://localhost:8080/polls/?key=abc123
   ```

   ```
   [{"id":"541728728ea48e5e5d5bb18a","title":"Favorite
   language?","options":["go","java","javascript","ruby"]}]
   ```

So now that we know that our API is working as expected, it's time to build something that consumes the API properly.

A web client that consumes the API

We are going to put together an ultra-simple web client that consumes the capabilities and data exposed through our API, allowing users to interact with the polling system we built in the previous chapter and earlier in this chapter. Our client will be made up of three web pages:

- An `index.html` page that shows all the polls
- A `view.html` page that shows the results of a specific poll
- A `new.html` page that allows users to create new polls

Create a new folder called `web` alongside the `api` folder, and add the following content to the `main.go` file:

```
package main
```

```
import (
  "flag"
  "log"
  "net/http"
)
func main() {
  var addr = flag.String("addr", ":8081", "website address")
  flag.Parse()
  mux := http.NewServeMux()
  mux.Handle("/", http.StripPrefix("/",
    http.FileServer(http.Dir("public"))))
  log.Println("Serving website at:", *addr)
  http.ListenAndServe(*addr, mux)
}
```

These few lines of Go code really highlight the beauty of the language and the Go standard library. They represent a complete, highly scalable, static website hosting program. The program takes an `addr` flag and uses the familiar `http.ServeMux` type to serve static files from a folder called `public`.

Building the next few pages—while we're building the UI—consists of writing a lot of HTML and JavaScript code. Since this is not Go code, if you'd rather not type it all out, feel free to head over to the GitHub repository for this book and copy and paste it from `https://github.com/matryer/goblueprints`.

An index page showing a list of polls

Create the `public` folder inside `web` and add the `index.html` file after writing the following HTML code in it:

```
<!DOCTYPE html>
<html>
<head>
  <title>Polls</title>
  <link rel="stylesheet"
    href="//maxcdn.bootstrapcdn.com/bootstrap/3.2.0/css/
    bootstrap.min.css">
</head>
<body>
</body>
</html>
```

We will use Bootstrap again to make our simple UI look nice, but we need to add two additional sections to the `body` tag of the HTML page. First, add the DOM elements that will display the list of polls:

```
<div class="container">
  <div class="col-md-4"></div>
  <div class="col-md-4">
    <h1>Polls</h1>
    <ul id="polls"></ul>
    <a href="new.html" class="btn btn-primary">Create new poll</a>
  </div>
  <div class="col-md-4"></div>
</div>
```

Here we are using Bootstrap's grid system to center-align our content that is made up of a list of polls and a link to `new.html`, where users can create new polls.

Next, add the following `script` tags and JavaScript underneath the previous code:

```
<script
src="//ajax.googleapis.com/ajax/libs/jquery/2.1.1/jquery.min.js"><
/script>
<script
src="//maxcdn.bootstrapcdn.com/bootstrap/3.2.0/js/bootstrap.min.js
"></script>
<script>
  $(function(){
    var update = function(){
      $.get("http://localhost:8080/polls/?key=abc123", null, null,
"json")
        .done(function(polls){
          $("#polls").empty();
          for (var p in polls) {
            var poll = polls[p];
            $("#polls").append(
              $("<li>").append(
                $("<a>")
                  .attr("href", "view.html?poll=polls/" + poll.id)
                  .text(poll.title)
              )
            )
          }
        }
      );
```

```
        window.setTimeout(update, 10000);
      }
      update();
    });
  </script>
```

We are using jQuery's `$.get` function to make an AJAX request to our web service. We are also hardcoding the API URL. In practice, you might decide against this, but you should at least use a domain name to abstract it. Once the polls have loaded, we use jQuery to build up a list containing hyperlinks to the `view.html` page, passing the ID of the poll as a query parameter.

A page to create a new poll

To allow users to create a new poll, create a file called `new.html` inside the `public` folder, and add the following HTML code to the file:

```
<!DOCTYPE html>
<html>
<head>
  <title>Create Poll</title>
  <link rel="stylesheet"
    href="//maxcdn.bootstrapcdn.com/bootstrap/3.2.0/css/
    bootstrap.min.css">
</head>
<body>
  <script
src="//ajax.googleapis.com/ajax/libs/jquery/2.1.1/jquery.min.js"><
/script>
  <script
src="//maxcdn.bootstrapcdn.com/bootstrap/3.2.0/js/bootstrap.min.js
"></script>
</body>
</html>
```

We are going to add the elements for an HTML form that will capture the information we need when creating a new poll, namely the title of the poll and the options. Add the following code inside the `body` tags:

```
<div class="container">
  <div class="col-md-4"></div>
  <form id="poll" role="form" class="col-md-4">
    <h2>Create Poll</h2>
    <div class="form-group">
      <label for="title">Title</label>
```

```
        <input type="text" class="form-control" id="title"
placeholder="Title">
      </div>
      <div class="form-group">
        <label for="options">Options</label>
        <input type="text" class="form-control" id="options"
placeholder="Options">
        <p class="help-block">Comma separated</p>
      </div>
      <button type="submit" class="btn btn-primary">Create
Poll</button> or <a href="/">cancel</a>
    </form>
    <div class="col-md-4"></div>
</div>
```

Since our API speaks JSON, we need to do a bit of work to turn the HTML form into a JSON-encoded string, and also break the comma-separated options string into an array of options. Add the following `script` tag:

```
<script>
  $(function(){
    var form = $("form#poll");
    form.submit(function(e){
      e.preventDefault();
      var title = form.find("input[id='title']").val();
      var options = form.find("input[id='options']").val();
      options = options.split(",");
      for (var opt in options) {
        options[opt] = options[opt].trim();
      }
      $.post("http://localhost:8080/polls/?key=abc123",
        JSON.stringify({
          title: title, options: options
        })
      ).done(function(d, s, r){
        location.href = "view.html?poll=" +
r.getResponseHeader("Location");
      });
    });
  });
</script>
```

Here we add a listener to the `submit` event of our form, and use jQuery's `val` method to collect the input values. We split the options with a comma, and trim the spaces away before using the `$.post` method to make the `POST` request to the appropriate API endpoint. `JSON.stringify` allows us to turn the data object into a JSON string, and we use that string as the body of the request, as expected by the API. On success, we pull out the `Location` header and redirect the user to the `view.html` page, passing a reference to the newly created poll as the parameter.

A page to show details of the poll

The final page of our app we need to complete is the `view.html` page where users can see the details and live results of the poll. Create a new file called `view.html` inside the `public` folder, and add the following HTML code to it:

```
<!DOCTYPE html>
<html>
<head>
  <title>View Poll</title>
  <link rel="stylesheet"
href="//maxcdn.bootstrapcdn.com/bootstrap/3.2.0/css/bootstrap.min.
css">
</head>
<body>
  <div class="container">
    <div class="col-md-4"></div>
    <div class="col-md-4">
      <h1 data-field="title">...</h1>
      <ul id="options"></ul>
      <div id="chart"></div>
      <div>
        <button class="btn btn-sm" id="delete">Delete this
poll</button>
      </div>
    </div>
    <div class="col-md-4"></div>
  </div>
</body>
</html>
```

This page is mostly similar to the other pages; it contains elements for presenting the title of the poll, the options, and a pie chart. We will be mashing up Google's Visualization API with our API to present the results. Underneath the final `div` tag in `view.html` (and above the closing `body` tag), add the following `script` tags:

```
<script src="//www.google.com/jsapi"></script>
<script
src="//ajax.googleapis.com/ajax/libs/jquery/2.1.1/jquery.min.js"><
/script>
<script
src="//maxcdn.bootstrapcdn.com/bootstrap/3.2.0/js/bootstrap.min.js
"></script>
<script>
google.load('visualization', '1.0', {'packages':['corechart']});
google.setOnLoadCallback(function(){
  $(function(){
    var chart;
    var poll = location.href.split("poll=")[1];
    var update = function(){
      $.get("http://localhost:8080/"+poll+"?key=abc123", null,
null, "json")
        .done(function(polls){
          var poll = polls[0];
          $('[data-field="title"]').text(poll.title);
          $("#options").empty();
          for (var o in poll.results) {
            $("#options").append(
              $("<li>").append(
                $("<small>").addClass("label label-
default").text(poll.results[o]),
                " ", o
              )
            )
          }
          if (poll.results) {
            var data = new google.visualization.DataTable();
            data.addColumn("string","Option");
            data.addColumn("number","Votes");
            for (var o in poll.results) {
              data.addRow([o, poll.results[o]])
            }
            if (!chart) {
              chart = new
google.visualization.PieChart(document.getElementById('chart'));
            }
```

```
              chart.draw(data, {is3D: true});
            }
          }
        );
        window.setTimeout(update, 1000);
      };
      update();
      $("#delete").click(function(){
        if (confirm("Sure?")) {
          $.ajax({
            url:"http://localhost:8080/"+poll+"?key=abc123",
            type:"DELETE"
          })
          .done(function(){
            location.href = "/";
          })
        }
      });
    });
  });
  </script>
```

We include the dependencies we will need to power our page, jQuery and Bootstrap, and also the Google JavaScript API. The code loads the appropriate visualization libraries from Google, and waits for the DOM elements to load before extracting the poll ID from the URL by splitting it on `poll=`. We then create a variable called `update` that represents a function responsible for generating the view of the page. This approach is taken to make it easy for us to use `window.setTimeout` to issue regular calls to update the view. Inside the `update` function, we use `$.get` to make a `GET` request to our `/polls/{id}` endpoint, replacing `{id}` with the actual ID we extracted from the URL earlier. Once the poll has loaded, we update the title on the page and iterate over the options to add them to the list. If there are results (remember in the previous chapter, the `results` map was only added to the data as votes start being counted), we create a new `google.visualization.PieChart` object and build a `google.visualization.DataTable` object containing the results. Calling `draw` on the chart causes it to render the data, and thus update the chart with the latest numbers. We then use `setTimeout` to tell our code to call `update` again in another second.

Finally, we bind to the `click` event of the `delete` button we added to our page, and after asking the user if they are sure, make a `DELETE` request to the polls URL and then redirect them back to the home page. It is this request that will actually cause the `OPTIONS` request to be made first, asking for permission, which is why we added explicit support for it in our `handlePolls` function earlier.

Running the solution

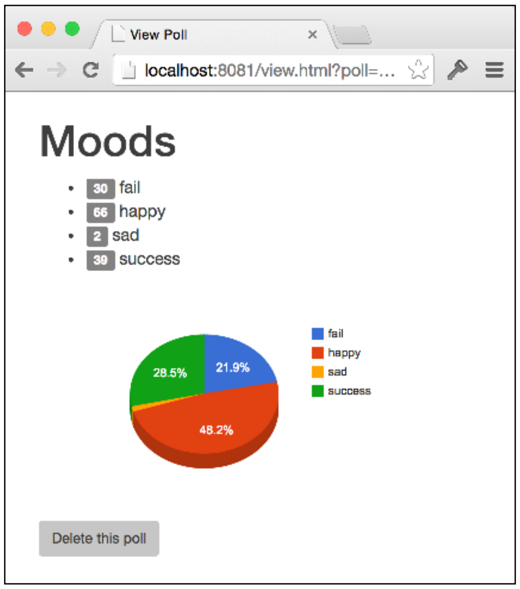
We have built many components over the last two chapters, and it is now time to see them all working together. This section contains everything you need to get all the items running, assuming you have the environment set up properly as described at the beginning of the previous chapter. This section assumes you have a single folder that contains four subfolders: `api`, `counter`, `twittervotes`, and `web`.

Assuming nothing is running, take the following steps (each step in its own terminal window):

1. In the top-level folder, start the `nsqlookupd` daemon:
   ```
   nsqlookupd
   ```
2. In the same directory, start the `nsqd` daemon:
   ```
   nsqd --lookupd-tcp-address=localhost:4160
   ```
3. Start the MongoDB daemon:
   ```
   mongod
   ```
4. Navigate to the `counter` folder and build and run it:
   ```
   cd counter
   go build -o counter
   ./counter
   ```
5. Navigate to the `twittervotes` folder and build and run it. Be sure that you have the appropriate environment variables set, otherwise you will see errors when you run the program:
   ```
   cd ../twittervotes
   go build -o twittervotes
   ./twittervotes
   ```
6. Navigate to the `api` folder and build and run it:
   ```
   cd ../api
   go build -o api
   ./api
   ```
7. Navigate to the `web` folder and build and run it:
   ```
   cd ../web
   go build -o web
   ./web
   ```

Now that everything is running, open a browser and head to `http://localhost:8081/`. Using the user interface, create a poll called `Moods` and input the options as `happy,sad,fail,and success`. These are common enough words that we are likely to see some relevant activity on Twitter.

Once you have created your poll, you will be taken to the view page where you will start to see the results coming in. Wait for a few seconds, and enjoy the fruits of your hard work as the UI updates in real time showing live, real-time results.



Moods

- 30 fail
- 66 happy
- 2 sad
- 39 success

Summary

In this chapter, we exposed the data for our social polling solution through a highly scalable RESTful API and built a simple website that consumes the API to provide an intuitive way for users to interact with it. The website consists of static content only, with no server-side processing (since the API does the heavy lifting for us). This allows us to host the website very cheaply on static hosting sites such as `bitballoon.com`, or to distribute the files to content delivery networks.

Within our API service, we learned how to share data between handlers without breaking or obfuscating the handler pattern from the standard library. We also saw how writing wrapped handler functions allows us to build a pipeline of functionality in a very simple and intuitive way.

We wrote some basic encoding and decoding functions that—while only simply wrapping their counterparts from the `encoding/json` package for now—could be improved later to support a range of different data representations without changing the internal interface to our code. We wrote a few simple helper functions that make responding to data requests easy, while providing the same kind of abstraction that would allow us to evolve our API later.

We saw how, for simple cases, switching on to HTTP methods is an elegant way to support many functions for a single endpoint. We also saw how, with a few extra lines of code, we are able to build in support for CORS to allow applications running on different domains to interact with our services—without the need for hacks like JSONP.

The code in this chapter, combined with the work we did in the previous chapter, provides a real-world, production-ready solution that implements the following flow:

1. The user clicks on the **Create Poll** button on the website, and enters the title and options for a poll.
2. The JavaScript running in the browser encodes the data as a JSON string and sends it in the body of a `POST` request to our API.
3. The API receives the request, and after validating the API key, setting up a database session, and storing it in our variables map, calls the `handlePolls` function that processes the request and stores the new poll in the MongoDB database.
4. The API redirects the user to the `view.html` page for the newly created poll.
5. Meanwhile, the `twittervotes` program loads all polls from the database, including the new one, and opens a connection to Twitter filtering on the hashtags that represent options from the polls.
6. As votes come in, `twittervotes` pushes them to NSQ.
7. The `counter` program is listening in on the appropriate channel and notices the votes coming in, counting each one and periodically making updates to the database.
8. The user sees the results displayed (and refreshed) on the `view.html` page as the website continually makes `GET` requests to the API endpoint for the selected poll.

In the next chapter, we will evolve our API and web skills to build a brand new start-up app called Meander. We'll see how we can write a full, static web server in just a few lines of Go code, and explore an interesting way of representing enumerators in a language that doesn't officially support them!

7
Random Recommendations Web Service

The concept behind the project that we will build in this chapter is a simple one: we want users to be able to generate random recommendations for things to do in specific geographical locations based on a predefined set of journey types that we will expose through the API. We will give our project the codename Meander.

Often on projects in the real world, you are responsible for the full stack; somebody else builds the website, a different person still might write the iOS app, and maybe an outsourced company builds the desktop version. On more successful API projects, you might not even know who the consumers of your API are, especially if it's a public API.

In this chapter, we will simulate this reality by designing and agreeing a minimal API design with a fictional partner up front before going on to implement the API. Once we have finished our side of the project, we will download a user interface built by our teammates to see the two work together to produce the final application.

In this chapter, you will:

- Learn to express the general goals of a project using short and simple Agile user stories
- Discover that you can agree a meeting point in a project by agreeing on the design of an API, which allows many people to work in parallel
- See how early versions of code can actually have data fixtures written in code and compiled into the program, allowing us to change the implementation later without touching the interface
- Learn a strategy that allows structs (and other types) to represent a public version of themselves for cases when we want to hide or transform internal representations

- Learn to use embedded structs to represent nested data, while keeping the interface of our types simple
- Learn to use `http.Get` to make external API requests, specifically to the Google Places API, with no code bloat
- Learn to effectively implement enumerators in Go, even though they aren't really a language feature
- Experience a real-world example of TDD
- See how the `math/rand` package makes it easy to select an item from a slice at random
- Learn an easy way to grab data from the URL parameters of the `http.Request` type

Project overview

Following Agile methodologies, let's write two user stories that describe the functionality of our project. User stories shouldn't be comprehensive documents describing the entire set of features of an application; rather small cards are perfect for not only describing what the user is trying to do, but why. Also, we should do this without trying to design the whole system up front or delve too deep into implementation details.

First we need a story about seeing the different journey types from which our users may select:

As a	traveler
I want	to see the different types of journeys I can get recommendations for
So that	I can decide what kind of evening to take my partner on

Secondly, we need a story about providing random recommendations for a selected journey type:

As a	traveler
I want	to see a random recommendation for my selected journey type
So that	I know where to go, and what the evening will entail

These two stories represent the two core capabilities that our API needs to provide, and actually ends up representing two endpoints.

In order to discover places around specified locations, we are going to make use of the Google Places API, which allows us to search for listings of businesses with given types, such as `bar`, `café`, or `movie_theater`. We will then use Go's `math/rand` package to pick from those places at random, building up a complete journey for our users.

> The Google Places API supports many business types; see `https://developers.google.com/places/documentation/supported_types` for the complete list.

Project design specifics

In order to turn our stories into an interactive application, we are going to provide two JSON endpoints; one to deliver the kinds of journeys users will be able to select in the application, and another to actually generate the random recommendations for the selected journey type.

```
GET /journeys
```

The above call should return a list such as the following:

```
[
  {
    name: "Romantic",
    journey: "park|bar|movie_theater|restaurant|florist"
  },
  {
    name: "Shopping",
    journey: "department_store|clothing_store|jewelry_store"
  }
]
```

The `name` field is a human-readable label for the type of recommendations the app generates, and the `journey` field is a pipe-separated list of supported journey types. It is the journey value that we will pass, as a URL parameter, into our other endpoint, which generates the actual recommendations:

```
GET /recommendations?
     lat=1&lng=2&journey=bar|cafe&radius=10&cost=$...$$$$$
```

This endpoint is responsible for querying the Google Places API and generating the recommendations before returning an array of place objects. We will use the parameters in the URL to control the kind of query to make as per the HTTP specification. The `lat` and `lng` parameters, representing latitude and longitude, respectively, tell our API where in the world we want recommendations from, and the `radius` parameter represents the distance in meters around the point in which we are interested in. The `cost` value is a human-readable way of representing the price range for places that the API returns. It is made up of two values: a lower and upper range separated by three dots. The number of dollar characters represents the price level, with `$` being the most affordable and `$$$$$` being the most expensive. Using this pattern, a value of `$...$$` would represent very low cost recommendations, where `$$$$...$$$$$` would represent a pretty expensive experience.

[Some programmers might insist the cost range is represented by numerical values, but since our API is going to be consumed by people, why not make things a little more interesting?]

An example payload for this call might look something like this:

```
[
  {
    icon: "http://maps.gstatic.com/mapfiles/place_api/icons/cafe-
71.png",
    lat: 51.519583, lng: -0.146251,
    vicinity: "63 New Cavendish St, London",
    name: "Asia House",
    photos: [{
      url:
"https://maps.googleapis.com/maps/api/place/photo?maxwidth=400&pho
toreference=CnRnAAAAyLRN"
     }]
  }, ...
]
```

The array returned contains a place object representing a random recommendation for each segment in the journey, in the appropriate order. The preceding example is a café in London. The data fields are fairly self-explanatory; the `lat` and `lng` fields represent the location of the place (they're short for latitude and longitude), the `name` and `vicinity` fields tell us what and where the business is, and the `photos` array gives us a list of relevant photographs from Google's servers. The `vicinity` and `icon` fields will help us deliver a richer experience to our users.

Representing data in code

We are first going to expose the journeys that users can select from, so create a new folder called `meander` in `GOPATH`, and add the following `journeys.go` code:

```
package meander
type j struct {
  Name       string
  PlaceTypes []string
}
var Journeys = []interface{}{
  &j{Name: "Romantic", PlaceTypes: []string{"park", "bar",
"movie_theater", "restaurant", "florist", "taxi_stand"}},
  &j{Name: "Shopping", PlaceTypes: []string{"department_store",
"cafe", "clothing_store", "jewelry_store", "shoe_store"}},
  &j{Name: "Night Out", PlaceTypes: []string{"bar", "casino",
"food", "bar", "night_club", "bar", "bar", "hospital"}},
  &j{Name: "Culture", PlaceTypes: []string{"museum", "cafe",
"cemetery", "library", "art_gallery"}},
  &j{Name: "Pamper", PlaceTypes: []string{"hair_care",
"beauty_salon", "cafe", "spa"}},
}
```

Here we define an internal type called `j` inside the `meander` package, which we then use to describe the journeys by creating instances of them inside the `Journeys` slice. This approach is an ultra-simple way of representing data in the code, without building in a dependency on an external data store.

> As an additional assignment, why not see if you can keep `golint` happy throughout this process? Every time you add some code, run `golint` for the packages and satisfy any suggestions that emerge. It cares a lot about exported items having no documentation, so adding simple comments in the correct format will keep it happy. To learn more about `golint`, see `https://github.com/golang/lint`.

Of course, this would likely evolve into just that later, maybe even with the ability for users to create and share their own journeys. Since we are exposing our data via an API, we are free to change the internal implementation without affecting the interface, so this approach is great for a version 1.0.

> We are using a slice of type `[]interface{}` because we will later implement a general way of exposing public data regardless of actual types.

A romantic journey consists of a visit first to a park, then a bar, a movie theater, then a restaurant, before a visit to a florist, and finally a taxi ride home; you get the general idea. Feel free to get creative and add others by consulting the supported types in the Google Places API.

You might have noticed that since we are containing our code inside a package called `meander` (rather than `main`), our code can never be run as a tool like the other APIs we have written so far. Create a new folder called `cmd` inside `meander`; this will house the actual command-line tool that exposes the `meander` package's capabilities via an HTTP endpoint.

Inside the `cmd` folder, add the following code to the `main.go` file:

```
package main
func main() {
  runtime.GOMAXPROCS(runtime.NumCPU())
  //meander.APIKey = "TODO"
  http.HandleFunc("/journeys", func(w http.ResponseWriter, r
*http.Request) {
    respond(w, r, meander.Journeys)
  })
  http.ListenAndServe(":8080", http.DefaultServeMux)
}
func respond(w http.ResponseWriter, r *http.Request, data
[]interface{}) error {
  return json.NewEncoder(w).Encode(data)
}
```

You will recognize this as a simple API endpoint program, mapping to the `/journeys` endpoint.

You'll have to import the `encoding/json`, `net/http`, and `runtime` packages, along with the `meander` package you created earlier.

The `runtime.GOMAXPROCS` call sets the maximum number of CPUs that our program can use, and we tell it to use them all. We then set the value of `APIKey` in the `meander` package (which is commented out for now, since we have yet to implement it) before calling the familiar `HandleFunc` function on the `net/http` package to bind our endpoint, which then just responds with the `meander.Journeys` variable. We borrow the abstract responding concept from the previous chapter by providing a `respond` function that encodes the specified data to the `http.ResponseWriter` type.

Let's run our API program by navigating to the `cmd` folder in a terminal and using `go run`. We don't need to build this into an executable file at this stage since it's just a single file:

```
go run main.go
```

Hit the `http://localhost:8080/journeys` endpoint, and notice that our `Journeys` data payload is served, which looks like this:

```
[{
  Name: "Romantic",
  PlaceTypes: [
    "park",
    "bar",
    "movie_theater",
    "restaurant",
    "florist",
    "taxi_stand"
  ]
}]
```

This is perfectly acceptable, but there is one major flaw: it exposes internals about our implementation. If we changed the `PlaceTypes` field name to `Types`, our API would change and it's important that we avoid this.

Projects evolve and change over time, especially successful ones, and as developers we should do what we can to protect our customers from the impact of the evolution. Abstracting interfaces is a great way to do this, as is taking ownership of the public-facing view of our data objects.

Public views of Go structs

In order to control the public view of structs in Go, we need to invent a way to allow individual `journey` types to tell us how they want to be exposed. In the `meander` folder, create a new file called `public.go`, and add the following code:

```
package meander
type Facade interface {
  Public() interface{}
}
func Public(o interface{}) interface{} {
  if p, ok := o.(Facade); ok {
    return p.Public()
  }
  return o
}
```

The `Facade` interface exposes a single `Public` method, which will return the public view of a struct. The `Public` function takes any object and checks to see whether it implements the `Facade` interface (does it have a `Public() interface{}` method?); and if it is implemented, calls the method and returns the result—otherwise it just returns the original object untouched. This allows us to pass anything through the `Public` function before writing the result to the `ResponseWriter` object, allowing individual structs to control their public appearance.

Let's implement a `Public` method for our `j` type by adding the following code to `journeys.go`:

```
func (j *j) Public() interface{} {
  return map[string]interface{}{
    "name":    j.Name,
    "journey": strings.Join(j.PlaceTypes, "|"),
  }
}
```

The public view of our `j` type joins the `PlaceTypes` field into a single string separated by the pipe character, as per our API design.

Head back to `cmd/main.go` and replace the `respond` method with one that makes use of our new `Public` function:

```
func respond(w http.ResponseWriter, r *http.Request, data []
interface{}) error {
  publicData := make([]interface{}, len(data))
  for i, d := range data {
    publicData[i] = meander.Public(d)
  }
  return json.NewEncoder(w).Encode(publicData)
}
```

Here we iterate over the data slice calling the `meander.Public` function for each item, building the results into a new slice of the same size. In the case of our `j` type, its `Public` method will be called to serve the public view of the data, rather than the default view. In a terminal, navigate to the `cmd` folder again and run `go run main.go` before hitting `http://localhost:8080/journeys` again. Notice that the same data has now changed to a new structure:

```
[{
  journey: "park|bar|movie_theater|restaurant|florist|taxi_stand",
  name: "Romantic"
}, ...]
```

Generating random recommendations

In order to obtain the places from which our code will randomly build up recommendations, we need to query the Google Places API. In the `meander` folder, add the following `query.go` file:

```
package meander
type Place struct {
  *googleGeometry `json:"geometry"`
  Name            string         `json:"name"`
  Icon            string         `json:"icon"`
  Photos          []*googlePhoto `json:"photos"`
  Vicinity        string         `json:"vicinity"`
}
type googleResponse struct {
  Results []*Place `json:"results"`
}
type googleGeometry struct {
  *googleLocation `json:"location"`
}
type googleLocation struct {
  Lat float64 `json:"lat"`
  Lng float64 `json:"lng"`
}
type googlePhoto struct {
  PhotoRef string `json:"photo_reference"`
  URL      string `json:"url"`
}
```

This code defines the structures we will need to parse the JSON response from the Google Places API into usable objects.

> Head over to the Google Places API documentation for an example of the response we are expecting. See `http://developers.google.com/places/documentation/search`.

Most of the preceding code will be obvious, but it's worth noticing that the `Place` type embeds the `googleGeometry` type, which allows us to represent the nested data as per the API, while essentially flattening it in our code. We do the same with `googleLocation` inside `googleGeometry`, which means that we will be able to access the `Lat` and `Lng` values directly on a `Place` object, even though they're technically nested in other structures.

Because we want to control how a `Place` object appears publically, let's give this type the following `Public` method:

```
func (p *Place) Public() interface{} {
  return map[string]interface{}{
    "name":     p.Name,
    "icon":     p.Icon,
    "photos":   p.Photos,
    "vicinity": p.Vicinity,
    "lat":      p.Lat,
    "lng":      p.Lng,
  }
}
```

Remember to run `golint` on this code to see which comments need to be added to the exported items.

Google Places API key

Like with most APIs, we will need an API key in order to access the remote services. Head over to the Google APIs Console, sign in with a Google account, and create a key for the Google Places API. For more detailed instructions, see the documentation on Google's developer website.

Once you have your key, let's make a variable inside the `meander` package that can hold it. At the top of `query.go`, add the following definition:

```
var APIKey string
```

Now nip back into `main.go`, remove the double slash `//` from the `APIKey` line, and replace the `TODO` value with the actual key provided by the Google APIs console.

Enumerators in Go

To handle the various cost ranges for our API, it makes sense to use an enumerator (or **enum**) to denote the various values and to handle conversions to and from string representations. Go doesn't explicitly provide enumerators, but there is a neat way of implementing them, which we will explore in this section.

A simple flexible checklist for writing enumerators in Go is:

- Define a new type, based on a primitive integer type
- Use that type whenever you need users to specify one of the appropriate values
- Use the `iota` keyword to set the values in a `const` block, disregarding the first zero value
- Implement a map of sensible string representations to the values of your enumerator
- Implement a `String` method on the type that returns the appropriate string representation from the map
- Implement a `ParseType` function that converts from a string to your type using the map

Now we will write an enumerator to represent the cost levels in our API. Create a new file called `cost_level.go` inside the `meander` folder and add the following code:

```
package meander
type Cost int8
const (
  _ Cost = iota
  Cost1
  Cost2
  Cost3
  Cost4
  Cost5
)
```

Here we define the type of our enumerator, which we have called `Cost`, and since we only need to represent a few values, we have based it on an `int8` range. For enumerators where we need larger values, you are free to use any of the integer types that work with `iota`. The `Cost` type is now a real type in its own right, and we can use it wherever we need to represent one of the supported values—for example, we can specify a `Cost` type as an argument in functions, or use it as the type for a field in a struct.

We then define a list of constants of that type, and use the `iota` keyword to indicate that we want incrementing values for the constants. By disregarding the first `iota` value (which is always zero), we indicate that one of the specified constants must be explicitly used, rather than the zero value.

To provide a string representation of our enumerator, we need only add a `String` method to the `Cost` type. This is a useful exercise even if you don't need to use the strings in your code, because whenever you use the print calls from the Go standard library (such as `fmt.Println`), the numerical values will be used by default. Often those values are meaningless and will require you to look them up, and even count the lines to determine the numerical value for each item.

> For more information about the `String()` method in Go, see the `Stringer` and `GoStringer` interfaces in the `fmt` package at `http://golang.org/pkg/fmt/#Stringer`.

Test-driven enumerator

To be sure that our enumerator code is working correctly, we are going to write unit tests that make some assertions about expected behavior.

Alongside `cost_level.go`, add a new file called `cost_level_test.go`, and add the following unit test:

```
package meander_test
import (
  "testing"
  "github.com/cheekybits/is"
  "path/to/meander"
)
func TestCostValues(t *testing.T) {
  is := is.New(t)
  is.Equal(int(meander.Cost1), 1)
  is.Equal(int(meander.Cost2), 2)
  is.Equal(int(meander.Cost3), 3)
  is.Equal(int(meander.Cost4), 4)
  is.Equal(int(meander.Cost5), 5)
}
```

You will need to run `go get` to get the CheekyBits' `is` package (from `github.com/cheekybits/is`).

> The `is` package is an alternative testing helper package, but this one is ultra-simple and deliberately bare-bones. You get to pick your favorite when you write your own projects.

Normally, we wouldn't worry about the actual integer value of constants in our enumerator, but since the Google Places API uses numerical values to represent the same thing, we need to care about the values.

You might have noticed something strange about this test file that breaks from convention. Although it is inside the `meander` folder, it is not a part of the `meander` package; rather it's in `meander_test`.

In Go, this is an error in every case except for tests. Because we are putting our test code into its own package, it means that we no longer have access to the internals of the `meander` package—notice how we have to use the package prefix. This may seem like a disadvantage, but in fact it allows us to be sure that we are testing the package as though we were a real user of it. We may only call exported methods and only have visibility into exported types; just like our users.

Run the tests by running `go test` in a terminal, and notice that it passes.

Let's add another test to make assertions about the string representations for each `Cost` constant. In `cost_level_test.go`, add the following unit test:

```
func TestCostString(t *testing.T) {
  is := is.New(t)
  is.Equal(meander.Cost1.String(), "$")
  is.Equal(meander.Cost2.String(), "$$")
  is.Equal(meander.Cost3.String(), "$$$")
  is.Equal(meander.Cost4.String(), "$$$$")
  is.Equal(meander.Cost5.String(), "$$$$$")
}
```

This test asserts that calling the `String` method for each constant yields the expected value. Running these tests will of course fail, because we haven't yet implemented the `String` method.

Underneath the `Cost` constants, add the following map and the `String` method:

```
var costStrings = map[string]Cost{
  "$":     Cost1,
  "$$":    Cost2,
  "$$$":   Cost3,
  "$$$$":  Cost4,
  "$$$$$": Cost5,
}
func (l Cost) String() string {
  for s, v := range costStrings {
    if l == v {
      return s
    }
  }
  return "invalid"
}
```

The `map[string]Cost` variable maps the cost values to the string representation, and the `String` method iterates over the map to return the appropriate value.

In our case, a simple return `strings.Repeat("$", int(l))` would work just as well (and wins because it's simpler code), but it often won't, therefore this section explores the general approach.

Now if we were to print out the `Cost3` value, we would actually see `$$$`, which is much more useful than numerical vales. However, since we do want to use these strings in our API, we are also going to add a `ParseCost` method.

In `cost_value_test.go`, add the following unit test:

```
    func TestParseCost(t *testing.T) {
      is := is.New(t)
      is.Equal(meander.Cost1, meander.ParseCost("$"))
      is.Equal(meander.Cost2, meander.ParseCost("$$"))
      is.Equal(meander.Cost3, meander.ParseCost("$$$"))
      is.Equal(meander.Cost4, meander.ParseCost("$$$$"))
      is.Equal(meander.Cost5, meander.ParseCost("$$$$$"))
    }
```

Here we assert that calling `ParseCost` will in fact yield the appropriate value depending on the input string.

In `cost_value.go`, add the following implementation code:

```
    func ParseCost(s string) Cost {
      return costStrings[s]
    }
```

Parsing a `Cost` string is very simple since this is how our map is laid out.

As we need to represent a range of cost values, let's imagine a `CostRange` type, and write the tests out for how we intend to use it. Add the following tests to `cost_value_test.go`:

```
func TestParseCostRange(t *testing.T) {
  is := is.New(t)
  var l *meander.CostRange
  l = meander.ParseCostRange("$$...$$$")
  is.Equal(l.From, meander.Cost2)
  is.Equal(l.To, meander.Cost3)
  l = meander.ParseCostRange("$...$$$$$")
  is.Equal(l.From, meander.Cost1)
  is.Equal(l.To, meander.Cost5)
}
func TestCostRangeString(t *testing.T) {
  is := is.New(t)
  is.Equal("$$...$$$$", (&meander.CostRange{
    From: meander.Cost2,
    To:   meander.Cost4,
  }).String())
}
```

We specify that passing in a string with two dollar characters first, followed by three dots and then three dollar characters should create a new `meander.CostRange` type that has `From` set to `meander.Cost2`, and `To` set to `meander.Cost3`. The second test does the reverse by testing that the `CostRange.String` method returns the appropriate value.

To make our tests pass, add the following `CostRange` type and associated `String` and `ParseString` functions:

```
type CostRange struct {
  From Cost
  To   Cost
}
func (r CostRange) String() string {
  return r.From.String() + "..." + r.To.String()
}
func ParseCostRange(s string) *CostRange {
  segs := strings.Split(s, "...")
```

```
    return &CostRange{
      From: ParseCost(segs[0]),
      To:   ParseCost(segs[1]),
    }
  }
```

This allows us to convert a string such as `$...$$$$$` to a structure that contains two `Cost` values; a `From` and `To` set and vice versa.

Querying the Google Places API

Now that we are capable of representing the results of the API, we need a way to represent and initiate the actual query. Add the following structure to `query.go`:

```
type Query struct {
  Lat          float64
  Lng          float64
  Journey      []string
  Radius       int
  CostRangeStr string
}
```

This structure contains all the information we will need to build up the query, all of which will actually come from the URL parameters in the requests from the client. Next, add the following `find` method, which will be responsible for making the actual request to Google's servers:

```
func (q *Query) find(types string) (*googleResponse, error) {
  u :=
"https://maps.googleapis.com/maps/api/place/nearbysearch/json"
  vals := make(url.Values)
  vals.Set("location", fmt.Sprintf("%g,%g", q.Lat, q.Lng))
  vals.Set("radius", fmt.Sprintf("%d", q.Radius))
  vals.Set("types", types)
  vals.Set("key", APIKey)
  if len(q.CostRangeStr) > 0 {
    r := ParseCostRange(q.CostRangeStr)
    vals.Set("minprice", fmt.Sprintf("%d", int(r.From)-1))
    vals.Set("maxprice", fmt.Sprintf("%d", int(r.To)-1))
  }
  res, err := http.Get(u + "?" + vals.Encode())
  if err != nil {
    return nil, err
  }
  defer res.Body.Close()
  var response googleResponse
  if err := json.NewDecoder(res.Body).Decode(&response); err !=
nil {
    return nil, err
  }
  return &response, nil
}
```

First we build the request URL as per the Google Places API specification, by appending the `url.Values` encoded string of the data for `lat`, `lng`, `radius`, and of course the `APIKey` values.

> The `url.Values` type is actually a `map[string][]string` type, which is why we use `make` rather than `new`.

The `types` value we specify as an argument represents the kind of business to look for. If there is a `CostRangeStr`, we parse it and set the `minprice` and `maxprice` values, before finally calling `http.Get` to actually make the request. If the request is successful, we defer the closing of the response body and use a `json.Decoder` method to decode the JSON that comes back from the API into our `googleResponse` type.

Building recommendations

Next we need to write a method that will allow us to make many calls to find, for the different steps in a journey. Underneath the `find` method, add the following `Run` method to the `Query` struct:

```
// Run runs the query concurrently, and returns the results.
func (q *Query) Run() []interface{} {
  rand.Seed(time.Now().UnixNano())
  var w sync.WaitGroup
  var l sync.Mutex
  places := make([]interface{}, len(q.Journey))
```

```
    for i, r := range q.Journey {
      w.Add(1)
      go func(types string, i int) {
        defer w.Done()
        response, err := q.find(types)
        if err != nil {
          log.Println("Failed to find places:", err)
          return
        }
        if len(response.Results) == 0 {
          log.Println("No places found for", types)
          return
        }
        for _, result := range response.Results {
          for _, photo := range result.Photos {
            photo.URL =
"https://maps.googleapis.com/maps/api/place/photo?" +
              "maxwidth=1000&photoreference=" + photo.PhotoRef +
"&key=" + APIKey
          }
        }
        randI := rand.Intn(len(response.Results))
        l.Lock()
        places[i] = response.Results[randI]
        l.Unlock()
      }(r, i)
    }
    w.Wait() // wait for everything to finish
    return places
  }
```

The first thing we do is set the random seed to the current time in nanoseconds past since January 1, 1970 UTC. This ensures that every time we call the `Run` method and use the `rand` package, the results will be different. If we didn't do this, our code would suggest the same recommendations every time, which defeats the object.

Since we need to make many requests to Google—and since we want to make sure this is as quick as possible—we are going to run all the queries at the same time by making concurrent calls to our `Query.find` method. So we next create a `sync.WaitGroup` method, and a map to hold the selected places along with a `sync.Mutex` method to allow many go routines to access the map concurrently.

We then iterate over each item in the `Journey` slice, which might be `bar`, `cafe`, `movie_theater`. For each item, we add `1` to the `WaitGroup` object, and call a goroutine. Inside the routine, we first defer the `w.Done` call informing the `WaitGroup` object that this request has completed, before calling our `find` method to make the actual request. Assuming no errors occurred, and it was indeed able to find some places, we iterate over the results and build up a usable URL for any photos that might be present. According to the Google Places API, we are given a `photoreference` key, which we can use in another API call to get the actual image. To save our clients from having to have knowledge of the Google Places API at all, we build the complete URL for them.

We then lock the map locker and with a call to `rand.Intn`, pick one of the options at random and insert it into the right position in the `places` slice, before unlocking the `sync.Mutex` method.

Finally, we wait for all goroutines to complete with a call to `w.Wait`, before returning the places.

Handlers that use query parameters

Now we need to wire up our `/recommendations` call, so head back to `main.go` in the `cmd` folder, and add the following code inside the `main` function:

```
http.HandleFunc("/recommendations", func(w http.ResponseWriter, r
*http.Request) {
  q := &meander.Query{
    Journey: strings.Split(r.URL.Query().Get("journey"), "|"),
  }
  q.Lat, _ = strconv.ParseFloat(r.URL.Query().Get("lat"), 64)
  q.Lng, _ = strconv.ParseFloat(r.URL.Query().Get("lng"), 64)
  q.Radius, _ = strconv.Atoi(r.URL.Query().Get("radius"))
  q.CostRangeStr = r.URL.Query().Get("cost")
  places := q.Run()
  respond(w, r, places)
})
```

This handler is responsible for preparing the `meander.Query` object and calling its `Run` method, before responding with the results. The `http.Request` type's URL value exposes the `Query` data that provides a `Get` method that, in turn, looks up a value for a given key.

The journey string is translated from the bar|cafe|movie_theater format to a slice of strings, by splitting on the pipe character. Then a few calls to functions in the strconv package turn the string latitude, longitude, and radius values into numerical types.

CORS

The final piece of the first version of our API will be to implement CORS as we did in the previous chapter. See if you can solve this problem yourself before reading on to the solution in the next section.

> If you are going to tackle this yourself, remember that your aim is to set the Access-Control-Allow-Origin response header to *. Also consider the http.HandlerFunc wrapping we did in the previous chapter. The best place for this code is probably in the cmd program, since that is what exposes the functionality through an HTTP endpoint.

In main.go, add the following cors function:

```
func cors(f http.HandlerFunc) http.HandlerFunc {
  return func(w http.ResponseWriter, r *http.Request) {
    w.Header().Set("Access-Control-Allow-Origin", "*")
    f(w, r)
  }
}
```

This familiar pattern takes in an http.HandlerFunc type and returns a new one that sets the appropriate header before calling the passed-in function. Now we can modify our code to make sure the cors function gets called for both of our endpoints. Update the appropriate lines in the main function:

```
func main() {
  runtime.GOMAXPROCS(runtime.NumCPU())
  meander.APIKey = "YOUR_API_KEY"
  http.HandleFunc("/journeys", cors(func(w http.ResponseWriter, r
*http.Request) {
    respond(w, r, meander.Journeys)
  }))
  http.HandleFunc("/recommendations", cors(func(w
http.ResponseWriter, r *http.Request) {
    q := &meander.Query{
      Journey: strings.Split(r.URL.Query().Get("journey"), "|"),
    }
```

```
      q.Lat, _ = strconv.ParseFloat(r.URL.Query().Get("lat"), 64)
      q.Lng, _ = strconv.ParseFloat(r.URL.Query().Get("lng"), 64)
      q.Radius, _ = strconv.Atoi(r.URL.Query().Get("radius"))
      q.CostRangeStr = r.URL.Query().Get("cost")
      places := q.Run()
      respond(w, r, places)
    }))
    http.ListenAndServe(":8080", http.DefaultServeMux)
  }
```

Now calls to our API will be allowed from any domain without a cross-origin error occurring.

Testing our API

Now that we are ready to test our API, head to a console and navigate to the `cmd` folder. Because our program imports the `meander` package, building the program will automatically build our `meander` package too.

Build and run the program:

```
go build -o meanderapi
./meanderapi
```

To see meaningful results from our API, let's take a minute to find your actual latitude and longitude. Head over to `http://mygeoposition.com/` and use the web tools to get the `x,y` values for a location you are familiar with.

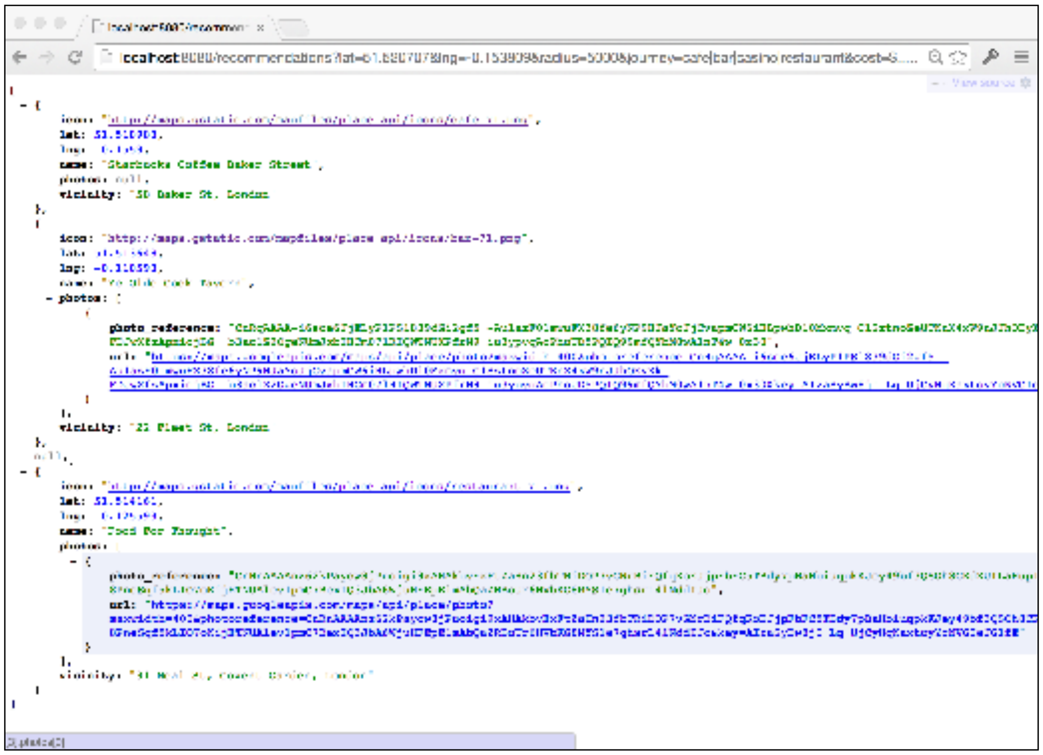
Or pick from these popular cities:

- London, England: `51.520707 x 0.153809`
- New York, USA: `40.7127840 x -74.0059410`
- Tokyo, Japan: `35.6894870 x 139.6917060`
- San Francisco, USA: `37.7749290 x -122.4194160`

Now open a web browser and access the `/recommendations` endpoint with some appropriate values for the fields:

```
http://localhost:8080/recommendations?
  lat=51.520707&lng=-0.153809&radius=5000&
  journey=cafe|bar|casino|restaurant&
  cost=$...$$$
```

The following screenshot shows what a sample recommendation around London might look like:



```
localhost:8080/recommendations?lat=51.520707&lng=-0.153809&radius=5000&journey=cafe|bar|casino|restaurant&cost=$...
[
 - {
      icon: "[illegible]",
      lat: 51.520703,
      lng: [illegible],
      name: "Starbucks Coffee Baker Street",
      photos: null,
      vicinity: "50 Baker St, London"
   },
   {
      icon: "http://maps.gstatic.com/mapfiles/place_api/icons/bar-71.png",
      lat: [illegible],
      lng: -0.110553,
      name: "[illegible]",
    - photos: [
         {
            photo_reference: "[illegible]",
            url: "[illegible]"
         }
      ],
      vicinity: "22 Fleet St, London"
   },
   null],
 - {
      icon: "[illegible]",
      lat: 51.514161,
      lng: [illegible],
      name: "Food For Thought",
      photos: [
       - {
            photo_reference: "[illegible]",
            url: "https://maps.googleapis.com/maps/api/place/photo?
            maxwidth=400&photoreference=[illegible]"
         }
      ],
      vicinity: "[illegible]"
   }
]
```

Feel free to play around with the values in the URL to see how powerful the simple API is by trying various journey strings, tweaking the locations, and trying different cost range value strings.

Web application

We are going to download a complete web application built to the same API specifications, and point it at our implementation to see it come to life before our eyes. Head over to `https://github.com/matryer/goblueprints/tree/master/chapter7/meanderweb` and download the `meanderweb` project into your `GOPATH`.

In a terminal, navigate to the `meanderweb` folder, and build and run it:

```
go build -o meanderweb
./meanderweb
```

This will start a website running on `localhost:8081`, which is hardcoded to look for the API running at `localhost:8080`. Because we added the CORS support, this won't be a problem despite them running on different domains.

Open a browser to `http://localhost:8081/` and interact with the application, while somebody else built the UI it would be pretty useless without the API that we built powering it.

Summary

In this chapter, we built an API that consumes and abstracts the Google Places API to provide a fun and interesting way of letting users plan their days and evenings.

We started by writing some simple and short user stories that described at a really high level what we wanted to achieve, without trying to design the implementation up front. In order to parallelize the project, we agreed the meeting point of the project as the API design, and we built towards it (as would our partners).

We embedded data directly in code, avoiding the need to investigate, design, and implement a data store in the early stages of a project. By caring instead about how that data is accessed (via the API endpoint), we allowed our future selves to completely change how and where the data is stored, without breaking any apps that have been written to our API.

We implemented the `Facade` interface, which allows our structs and other types to provide public representations of them, without revealing messy or sensitive details about our implementation.

Our foray into enumerators gave us a useful starting point to build enumerated types, even though there is no official support for them in the language. The `iota` keyword that we used lets us specify constants of our own numerical type, with incrementing values. The common `String` method that we implemented showed us how to make sure our enumerated types don't become obscure numbers in our logs. At the same time, we also saw a real-world example of TDD, and red/green programming where we wrote unit tests that first fail, but which we then go on to make pass by writing the implementation code.

8
Filesystem Backup

There are many solutions that provide filesystem backup capabilities. These include everything from apps such as Dropbox, Box, Carbonite to hardware solutions such as Apple's Time Machine, Seagate, or network-attached storage products, to name a few. Most consumer tools provide some key automatic functionality, along with an app or website for you to manage your policies and content. Often, especially for developers, these tools don't quite do the things we need them to. However, thanks to Go's standard library (that includes packages such as `ioutil` and `os`) we have everything we need to build a backup solution that behaves exactly as we need it to.

For our final project, we will build a simple filesystem backup for our source code projects that archive specified folders and save a snapshot of them every time we make a change. The change could be when we tweak a file and save it, or if we add new files and folders, or even if we delete a file. We want to be able to go back to any point in time to retrieve old files.

Specifically in this chapter, you will learn:

- How to structure projects that consist of packages and command-line tools
- A pragmatic approach to persisting simple data across tool executions
- How the `os` package allows you to interact with a filesystem
- How to run code in an infinite timed loop, while respecting *Ctrl* + *C*
- How to use `filepath.Walk` to iterate over files and folders
- How to quickly determine if the contents of a directory have changed
- How to use the `archive/zip` package to zip files
- How to build tools that care about a combination of command-line flags and normal arguments

Solution design

We will start by listing some high-level acceptance criteria for our solution and the approach we want to take:

- The solution should create a snapshot of our files at regular intervals, as we make changes to our source code projects
- We want to control the interval at which the directories are checked for changes
- Code projects are primarily text-based, so zipping the directories to generate archives will save a lot of space
- We will build this project quickly, while keeping a close watch over where we might want to make improvements later
- Any implementation decisions we make should be easily modified if we decide to change our implementation in the future
- We will build two command-line tools, the backend daemon that does the work, and a user interaction utility that will let us list, add, and remove paths from the backup service

Project structure

It is common in Go solutions to have, in a single project, both a package that allows other Go programmers to use your capabilities, and a command-line tool that allows end users to use your code.

A convention is emerging to structure the project by having the package in the main project folder, and the command-line tool inside a subfolder called `cmd`, or `cmds` if you have multiple commands. Because all packages (regardless of the directory tree) are equal in Go, you can import the main package from the subpackages, knowing you'll never need to import the commands from the main package. This may seem like an unnecessary abstraction, but is actually quite a common pattern and can be seen in the standard Go tool chain with examples such as `gofmt` and `goimports`.

For example, for our project we are going to write a package called `backup`, and two command-line tools: the daemon and the user interaction tool. We will structure our project in the following way:

```
/backup - package
/backup/cmds/backup – user interaction tool
/backup/cmds/backupd – worker daemon
```

Backup package

We are first going to write the `backup` package, of which we will become the first customer when we write the associated tools. The package will be responsible for deciding whether directories have changed and need backing up or not, as well as actually performing the backup procedure too.

Obvious interfaces?

The first thing to think about when embarking on a new Go program is whether any interfaces stand out to you. We don't want to over-abstract or waste too much time up front designing something that we know will change as we start to code, but that doesn't mean we shouldn't look for obvious concepts that are worth pulling out. Since our code will archive files, the `Archiver` interface pops out as a candidate.

Create a new folder inside your `GOPATH` called `backup`, and add the following `archiver.go` code:

```
package backup

type Archiver interface {
  Archive(src, dest string) error
}
```

An `Archiver` interface will specify a method called `Archive` that takes source and destination paths and returns an error. Implementations of this interface will be responsible for archiving the source folder, and storing it in the destination path.

Defining an interface up front is a nice way to get some concepts out of our heads and into code; it doesn't mean this interface can't change as we evolve our solution as long as we remember the power of simple interfaces. Also, remember that most of the I/O interfaces in the `io` package expose only a single method.

From the very beginning, we have made the case that while we are going to implement ZIP files as our archive format, we could easily swap this out later with another kind of `Archiver` format.

Implementing ZIP

Now that we have the interface for our `Archiver` types, we are going to implement one that uses the ZIP file format.

Add the following `struct` definition to `archiver.go`:

```
type zipper struct{}
```

We are not going to export this type, which might make you jump to the conclusion that users outside of the package won't be able to make use of it. In fact, we are going to provide them with an instance of the type for them to use, to save them from having to worry about creating and managing their own types.

Add the following exported implementation:

```
// Zip is an Archiver that zips and unzips files.
var ZIP Archiver = (*zipper)(nil)
```

This curious snippet of Go voodoo is actually a very interesting way of exposing the intent to the compiler, without using any memory (literally 0 bytes). We are defining a variable called `ZIP` of type `Archiver`, so from outside the package it's pretty clear that we can use that variable wherever `Archiver` is needed—if you want to zip things. Then we assign it with `nil` cast to the type `*zipper`. We know that `nil` takes no memory, but since it's cast to a `zipper` pointer, and given that our `zipper` struct has no fields, it's an appropriate way of solving a problem, which hides the complexity of code (and indeed the actual implementation) from outside users. There is no reason anybody outside of the package needs to know about our `zipper` type at all, which frees us up to change the internals without touching the externals at any time; the true power of interfaces.

Another handy side benefit to this trick is that the compiler will now be checking whether our zipper type properly implements the `Archiver` interface or not, so if you try to build this code you'll get a compiler error:

```
./archiver.go:10: cannot use (*zipper)(nil) (type *zipper) as type
Archiver in assignment:
  *zipper does not implement Archiver (missing Archive method)
```

We see that our `zipper` type does not implement the `Archive` method as mandated in the interface.

You can also use the `Archive` method in test code to ensure that your types implement the interfaces they should. If you don't need to use the variable, you can always throw it away by using an underscore and you'll still get the compiler help:

```
var _ Interface = (*Implementation)(nil)
```

To make the compiler happy, we are going to add the implementation of the `Archive` method for our `zipper` type.

Add the following code to `archiver.go`:

```
func (z *zipper) Archive(src, dest string) error {
  if err := os.MkdirAll(filepath.Dir(dest), 0777); err != nil {
    return err
  }
  out, err := os.Create(dest)
  if err != nil {
    return err
  }
  defer out.Close()
  w := zip.NewWriter(out)
  defer w.Close()
  return filepath.Walk(src, func(path string, info os.FileInfo,
err error) error {
    if info.IsDir() {
      return nil // skip
    }
    if err != nil {
      return err
    }
    in, err := os.Open(path)
    if err != nil {
      return err
    }
    defer in.Close()
    f, err := w.Create(path)
    if err != nil {
      return err
    }
    io.Copy(f, in)
    return nil
  })
}
```

You will have to also import the `archive/zip` package from the Go standard library. In our `Archive` method, we take the following steps to prepare writing to a ZIP file:

- Use `os.MkdirAll` to ensure the destination directory exists. The `0777` code represents the file permissions with which to create any missing directories.
- Use `os.Create` to create a new file as specified by the `dest` path.
- If the file is created without error, defer the closing of the file with `defer out.Close()`.
- Use `zip.NewWriter` to create a new `zip.Writer` type that will write to the file we just created, and defer the closing of the writer.

Once we have a `zip.Writer` type ready to go, we use the `filepath.Walk` function to iterate over the source directory `src`.

The `filepath.Walk` function takes two arguments: the root path, and a callback function `func` to be called for every item (files and folders) it encounters while iterating over the file system. The `filepath.Walk` function is recursive, so it will travel deep into subfolders too. The callback function itself takes three arguments: the full path of the file, the `os.FileInfo` object that describes the file or folder itself, and an error (it also returns an error in case something goes wrong). If any calls to the callback function result in an error being returned, the operation will be aborted and `filepath.Walk` returns that error. We simply pass that up to the caller of `Archive` and let them worry about it, since there's nothing more we can do.

For each item in the tree, our code takes the following steps:

- If the `info.IsDir` method tells us that the item is a folder, we just return `nil`, effectively skipping it. There is no reason to add folders to ZIP archives, because anyway the path of the files will encode that information for us.
- If an error is passed in (via the third argument), it means something went wrong when trying to access information about the file. This is uncommon, so we just return the error, which will eventually be passed out to the caller of `Archive`.
- Use `os.Open` to open the source file for reading, and if successful defer its closing.
- Call `Create` on the `ZipWriter` object to indicate that we want to create a new compressed file, and give it the full path of the file, which includes the directories it is nested inside.
- Use `io.Copy` to read all of the bytes from the source file, and write them through the `ZipWriter` object to the ZIP file we opened earlier.
- Return `nil` to indicate no errors.

This chapter will not cover unit testing or **Test-driven Development** (**TDD**) practices, but feel free to write a test to ensure that our implementation does what it is meant to do.

[Since we are writing a package, spend some time commenting the exported pieces so far. You can use `golint` to help you find any exported pieces you may have missed.]

Has the filesystem changed?

One of the biggest problems our backup system has is deciding whether a folder has changed or not in a cross-platform, predictable, and reliable way. A few things spring to mind when we think about this problem: should we just check the last modified date on the top-level folder? Should we use system notifications to be informed whenever a file we care about changes? There are problems with both of these approaches, and it turns out it's not a trivial problem to solve.

We are instead going to generate an MD5 hash made up of all of the information that we care about when considering whether something has changed or not.

Looking at the `os.FileInfo` type, we can see that we can find out a lot of information about a file:

```
type FileInfo interface {
  Name() string       // base name of the file
  Size() int64        // length in bytes for regular files;
                         system-dependent for others
  Mode() FileMode     // file mode bits
  ModTime() time.Time // modification time
  IsDir() bool        // abbreviation for Mode().IsDir()
  Sys() interface{}   // underlying data source (can return nil)
}
```

To ensure we are aware of a variety of changes to any file in a folder, the hash will be made up of the filename and path (so if they rename a file, the hash will be different), size (if a file changes size, it's obviously different), last modified date, whether the item is a file or folder, and file mode bits. Even though we won't be archiving the folders, we still care about their names and the tree structure of the folder.

Create a new file called `dirhash.go` and add the following function:

```
package backup
import (
  "crypto/md5"
```

```
    "fmt"
    "io"
    "os"
    "path/filepath"
  )
  func DirHash(path string) (string, error) {
    hash := md5.New()
    err := filepath.Walk(path, func(path string, info os.FileInfo, err
  error) error {
      if err != nil {
        return err
      }
      io.WriteString(hash, path)
      fmt.Fprintf(hash, "%v", info.IsDir())
      fmt.Fprintf(hash, "%v", info.ModTime())
      fmt.Fprintf(hash, "%v", info.Mode())
      fmt.Fprintf(hash, "%v", info.Name())
      fmt.Fprintf(hash, "%v", info.Size())
      return nil
    })
    if err != nil {
      return "", err
    }
    return fmt.Sprintf("%x", hash.Sum(nil)), nil
  }
```

We first create a new `hash.Hash` that knows how to calculate MD5s, before using `filepath.Walk` to iterate over all of the files and folders inside the specified path directory. For each item, assuming there are no errors, we write the differential information to the hash generator using `io.WriteString`, which lets us write a string to an `io.Writer`, and `fmt.Fprintf`, which does the same but exposes formatting capabilities at the same time, allowing us to generate the default value format for each item using the `%v` format verb.

Once each file has been processed, and assuming no errors occurred, we then use `fmt.Sprintf` to generate the result string. The `Sum` method on a `hash.Hash` calculates the final hash value with the specified values appended. In our case, we do not want to append anything since we've already added all of the information we care about, so we just pass `nil`. The `%x` format verb indicates that we want the value to be represented in hex (base 16) with lowercase letters. This is the usual way of representing an MD5 hash.

Checking for changes and initiating a backup

Now that we have the ability to hash a folder, and to perform a backup, we are going to put the two together in a new type called `Monitor`. The `Monitor` type will have a map of paths with their associated hashes, a reference to any `Archiver` type (of course, we'll use `backup.ZIP` for now), and a destination string representing where to put the archives.

Create a new file called `monitor.go` and add the following definition:

```
type Monitor struct {
  Paths       map[string]string
  Archiver    Archiver
  Destination string
}
```

In order to trigger a check for changes, we are going to add the following `Now` method:

```
func (m *Monitor) Now() (int, error) {
  var counter int
  for path, lastHash := range m.Paths {
    newHash, err := DirHash(path)
    if err != nil {
      return 0, err
    }
    if newHash != lastHash {
      err := m.act(path)
      if err != nil {
        return counter, err
      }
      m.Paths[path] = newHash // update the hash
      counter++
    }
  }
  return counter, nil
}
```

The `Now` method iterates over every path in the map and generates the latest hash of that folder. If the hash does not match the hash from the map (generated the last time it checked), then it is considered to have changed, and needs backing up again. We do this with a call to the as yet unwritten `act` method, before then updating the hash in the map with this new hash.

To give our users a high-level indication of what happened when they called `Now`, we are also maintaining a counter which we increment every time we back up a folder. We will use this later to keep our end users up-to-date on what the system is doing without bombarding them with information.

```
m.act undefined (type *Monitor has no field or method act)
```

The compiler is helping us again and reminding us that we have yet to add the `act` method:

```
func (m *Monitor) act(path string) error {
  dirname := filepath.Base(path)
  filename := fmt.Sprintf("%d.zip", time.Now().UnixNano())
  return m.Archiver.Archive(path, filepath.Join(m.Destination,
dirname, filename))
}
```

Because we have done the heavy lifting in our ZIP `Archiver` type, all we have to do here is generate a filename, decide where the archive will go, and call the `Archive` method.

> If the `Archive` method returns an error, the `act` method and then the `Now` method will each return it. This mechanism of passing errors up the chain is very common in Go and allows you to either handle cases where you can do something useful to recover, or else defer the problem to somebody else.

The `act` method in the preceding code uses `time.Now().UnixNano()` to generate a timestamp filename and hardcodes the `.zip` extension.

Hardcoding is OK for a short while

Hardcoding the file extension like we have is OK in the beginning, but if you think about it we have blended concerns a little here. If we change the `Archiver` implementation to use RAR or a compression format of our making, the `.zip` extension would no longer be appropriate.

> Before reading on, think about what steps you might take to avoid hardcoding. Where does the filename extension decision live? What changes would you need to make in order to avoid hardcoding properly?

The right place for the filename extensions decision is probably in the `Archiver` interface, since it knows the kind of archiving it will be doing. So we could add an `Ext()` string method and access that from our `act` method. But we can add a little extra power with not much extra work by instead allowing `Archiver` authors to specify the entire filename format, rather than just the extension.

Back in `archiver.go`, update the `Archiver` interface definition:

```
type Archiver interface {
  DestFmt() string
  Archive(src, dest string) error
}
```

Our `zipper` type needs to now implement this:

```
func (z *zipper) DestFmt() string {
  return "%d.zip"
}
```

Now that we can ask our `act` method to get the whole format string from the `Archiver` interface, update the `act` method:

```
func (m *Monitor) act(path string) error {
  dirname := filepath.Base(path)
  filename := fmt.Sprintf(m.Archiver.DestFmt(),
time.Now().UnixNano())
  return m.Archiver.Archive(path, filepath.Join(m.Destination,
dirname, filename))
}
```

The user command-line tool

The first of two tools we will build allows the user to add, list, and remove paths for the backup daemon tool (which we will write later). You could expose a web interface, or even use the binding packages for desktop user interface integration, but we are going to keep things simple and build ourselves a command-line tool.

Create a new folder called `cmds` inside the `backup` folder and create another `backup` folder inside that.

It's good practice to name the folder of the command and the command binary itself the same.

Inside our new `backup` folder, add the following code to `main.go`:

```
func main() {
  var fatalErr error
  defer func() {
    if fatalErr != nil {
      flag.PrintDefaults()
      log.Fatalln(fatalErr)
    }
  }()
  var (
    dbpath = flag.String("db", "./backupdata", "path to database
directory")
  )
  flag.Parse()
  args := flag.Args()
  if len(args) < 1 {
    fatalErr = errors.New("invalid usage; must specify command")
    return
  }
}
```

We first define our `fatalErr` variable and defer the function that checks to ensure that value is `nil`. If it is not, it will print the error along with flag defaults and exit with a non-zero status code. We then define a flag called `db` that expects the path to the `filedb` database directory, before parsing the flags and getting the remaining arguments and ensuring there is at least one.

Persisting small data

In order to keep track of the paths, and the hashes that we generate, we will need some kind of data storage mechanism that ideally works even when we stop and start our programs. We have lots of choices here: everything from a text file to a full horizontally scalable database solution. The Go ethos of simplicity tells us that building-in a database dependency to our little backup program would not be a great idea; rather we should ask what is the simplest way we can solve this problem?

The `github.com/matryer/filedb` package is an experimental solution for just this kind of problem. It lets you interact with the filesystem as though it were a very simple schemaless database. It takes its design lead from packages such as `mgo`, and can be used in the cases where data querying needs are very simple. In `filedb`, a database is a folder, and a collection is a file where each line represents a different record. Of course, this could all change as the `filedb` project evolves, but the interface hopefully won't.

Add the following code to the end of the `main` function:

```
db, err := filedb.Dial(*dbpath)
if err != nil {
  fatalErr = err
  return
}
defer db.Close()
col, err := db.C("paths")
if err != nil {
  fatalErr = err
  return
}
```

Here we use the `filedb.Dial` function to connect with the `filedb` database. In actuality, nothing much happens here except specifying where the database is, since there are no real database servers to connect to (although this might change in the future, which is why such provisions exist in the interface). If that was successful, we defer the closing of the database. Closing the database does actually do something, since files may be open that need to be cleaned up.

Following the `mgo` pattern, next we specify a collection using the `C` method and keep a reference to it in the `col` variable. If at any point an error occurs, we assign it to the `fatalErr` variable and return.

To store data, we are going to define a type called `path`, which will store the full path and the last hash value, and use JSON encoding to store this in our `filedb` database. Add the following `struct` definition above the `main` function:

```
type path struct {
  Path string
  Hash string
}
```

Parsing arguments

When we call `flag.Args` (as opposed to `os.Args`), we receive a slice of arguments excluding the flags. This allows us to mix flag arguments and non-flag arguments in the same tool.

We want our tool to be able to be used in the following ways:

- To add a path:
  `backup -db=/path/to/db add {path} [paths...]`

- To remove a path:

  ```
  backup -db=/path/to/db remove {path} [paths...]
  ```

- To list all paths:

  ```
  backup -db=/path/to/db list
  ```

To achieve this, since we have already dealt with flags, we must check the first (non-flag) argument.

Add the following code to the `main` function:

```
switch strings.ToLower(args[0]) {
case "list":
case "add":
case "remove":
}
```

Here we simply switch on the first argument, after setting it to lowercase (if the user types `backup LIST`, we still want it to work).

Listing the paths

To list the paths in the database, we are going to use a `ForEach` method on the path's `col` variable. Add the following code to the list case:

```
var path path
col.ForEach(func(i int, data []byte) bool {
  err := json.Unmarshal(data, &path)
  if err != nil {
    fatalErr = err
    return false
  }
  fmt.Printf("= %s\n", path)
  return false
})
```

We pass in a callback function to `ForEach` that will be called for every item in that collection. We then `Unmarshal` it from JSON, into our `path` type, and just print it out using `fmt.Printf`. We return `false` as per the `filedb` interface, which tells us that returning `true` would stop iterating and that we want to make sure we list them all.

String representations for your own types

If you print structs in Go in this way, using the `%s` format verbs, you can get some messy results that are difficult for users to read. If, however, the type implements a `String()` string method, that will be used instead and we can use this to control what gets printed. Below the path struct, add the following method:

```
func (p path) String() string {
  return fmt.Sprintf("%s [%s]", p.Path, p.Hash)
}
```

This tells the `path` type how it should represent itself as a string.

Adding paths

To add a path, or many paths, we are going to iterate over the remaining arguments and call the `InsertJSON` method for each one. Add the following code to the `add` case:

```
if len(args[1:]) == 0 {
  fatalErr = errors.New("must specify path to add")
  return
}
for _, p := range args[1:] {
  path := &path{Path: p, Hash: "Not yet archived"}
  if err := col.InsertJSON(path); err != nil {
    fatalErr = err
    return
  }
  fmt.Printf("+ %s\n", path)
}
```

If the user hasn't specified any additional arguments, like if they just called `backup add` without typing any paths, we will return a fatal error. Otherwise, we do the work and print out the path string (prefixed with a `+` symbol) to indicate that it was successfully added. By default, we'll set the hash to the `Not yet archived` string literal—this is an invalid hash but serves the dual purposes of letting the user know that it hasn't yet been archived, as well as indicating as such to our code (given that a hash of the folder will never equal that string).

Removing paths

To remove a path, or many paths, we use the `RemoveEach` method for the path's collection. Add the following code to the `remove` case:

```
var path path
col.RemoveEach(func(i int, data []byte) (bool, bool) {
  err := json.Unmarshal(data, &path)
  if err != nil {
    fatalErr = err
    return false, true
  }
  for _, p := range args[1:] {
    if path.Path == p {
      fmt.Printf("- %s\n", path)
      return true, false
    }
  }
  return false, false
})
```

The callback function we provide to `RemoveEach` expects us to return two bool types: the first one indicates whether the item should be removed or not, and the second one indicates whether we should stop iterating or not.

Using our new tool

We have completed our simple `backup` command-line tool. Let's see it in action. Create a folder called `backupdata` inside `backup/cmds/backup`; this will become the `filedb` database.

Build the tool in a terminal by navigating to the `main.go` file and running:

```
go build -o backup
```

If all is well, we can now add a path:

```
./backup -db=./backupdata add ./test ./test2
```

You should see the expected output:

```
+ ./test [Not yet archived]
+ ./test2 [Not yet archived]
```

Now let's add another path:

```
./backup -db=./backupdata add ./test3
```

You should now see the complete list:

```
./backup -db=./backupdata list
```

Our program should yield:

```
= ./test [Not yet archived]
= ./test2 [Not yet archived]
= ./test3 [Not yet archived]
```

Let's remove `test3` to make sure the remove functionality is working:

```
./backup -db=./backupdata remove ./test3
./backup -db=./backupdata list
```

This will take us back to:

```
+ ./test [Not yet archived]
+ ./test2 [Not yet archived]
```

We are now able to interact with the `filedb` database in a way that makes sense for our use case. Next we build the daemon program that will actually use our `backup` package to do the work.

The daemon backup tool

The `backup` tool, which we will call `backupd`, will be responsible for periodically checking the paths listed in the `filedb` database, hashing the folders to see whether anything has changed, and using the `backup` package to actually perform the archiving of folders that need it.

Create a new folder called `backupd` alongside the `backup/cmds/backup` folder, and let's jump right into handling the fatal errors and flags:

```
func main() {
  var fatalErr error
  defer func() {
    if fatalErr != nil {
      log.Fatalln(fatalErr)
    }
  }()
  var (
    interval = flag.Int("interval", 10, "interval between checks
(seconds)")
    archive  = flag.String("archive", "archive", "path to archive
location")
    dbpath   = flag.String("db", "./db", "path to filedb
database")
  )
  flag.Parse()
}
```

You must be quite used to seeing this kind of code by now. We defer the handling of fatal errors before specifying three flags: `interval`, `archive`, and `db`. The `interval` flag represents the number of seconds between checks to see whether folders have changed, the `archive` flag is the path to the archive location where ZIP files will go, and the `db` flag is the path to the same `filedb` database that the `backup` command is interacting with. The usual call to `flag.Parse` sets the variables up and validates whether we're ready to move on.

In order to check the hashes of the folders, we are going to need an instance of `Monitor` that we wrote earlier. Append the following code to the `main` function:

```
m := &backup.Monitor{
  Destination: *archive,
  Archiver:    backup.ZIP,
  Paths:       make(map[string]string),
}
```

Here we create a `backup.Monitor` method using the `archive` value as the `Destination` type. We'll use the `backup.ZIP` archiver and create a map ready for it to store the paths and hashes internally. At the start of the daemon, we want to load the paths from the database so that it doesn't archive unnecessarily as we stop and start things.

Add the following code to the `main` function:

```
db, err := filedb.Dial(*dbpath)
if err != nil {
  fatalErr = err
```

```
    return
  }
  defer db.Close()
  col, err := db.C("paths")
  if err != nil {
    fatalErr = err
    return
  }
```

You have seen this code before too; it dials the database and creates an object that allows us to interact with the `paths` collection. If anything fails, we set `fatalErr` and return.

Duplicated structures

Since we're going to use the same path structure as in our user command-line tool program, we need to include a definition of it for this program too. Insert the following structure above the `main` function:

```
type path struct {
  Path string
  Hash string
}
```

The object-oriented programmers out there are no doubt by now screaming at the pages demanding for this shared snippet to exist in one place only and not be duplicated in both programs. I urge you to resist this compulsion of early abstraction. These four lines of code hardly justify a new package and therefore dependency for our code, when they can just as easily exist in both programs with very little overhead. Consider also that we might want to add a `LastChecked` field to our `backupd` program so that we could add rules where each folder only gets archived at most once an hour. Our `backup` program doesn't care about this and will chug along perfectly happy with its view into what fields constitute a path.

Caching data

We can now query all existing paths and update the `Paths` map, which is a useful technique to increase the speed of a program, especially given slow or disconnected data stores. By loading the data into a cache (in our case, the `Paths` map), we can access it at lightening speeds without having to consult the files each time we need information.

Add the following code to the body of the `main` function:

```
var path path
col.ForEach(func(_ int, data []byte) bool {
  if err := json.Unmarshal(data, &path); err != nil {
    fatalErr = err
    return true
  }
  m.Paths[path.Path] = path.Hash
  return false // carry on
})
if fatalErr != nil {
  return
}
if len(m.Paths) < 1 {
  fatalErr = errors.New("no paths - use backup tool to add at
least one")
  return
}
```

Using the `ForEach` method again allows us to iterate over all the paths in the database. We `Unmarshal` the JSON bytes into the same path structure as we used in our other program and set the values in the `Paths` map. Assuming nothing goes wrong, we do a final check to make sure there is at least one path, and if not, return with an error.

One limitation to our program is that it will not dynamically add paths once it has started. The daemon would need to be restarted. If this bothers you, you could always build in a mechanism that updates the `Paths` map periodically.

Infinite loops

The next thing we need to do is to perform a check on the hashes right away to see whether anything needs archiving, before entering into an infinite timed loop where we check again at regular specified intervals.

An infinite loop sounds like a bad idea; in fact to some it sounds like a bug. However, since we're talking about an infinite loop within this program, and since infinite loops can be easily broken with a simple `break` command, they're not as dramatic as they might sound.

In Go, to write an infinite loop is as simple as:

```
for {}
```

The instructions inside the braces get executed over and over again, as quickly as the machine running the code can execute them. Again this sounds like a bad plan, unless you're careful about what you're asking it to do. In our case, we are immediately initiating a `select` case on the two channels that will block safely until one of the channels has something interesting to say.

Add the following code:

```
check(m, col)
signalChan := make(chan os.Signal, 1)
signal.Notify(signalChan, syscall.SIGINT, syscall.SIGTERM)
for {
  select {
  case <-time.After(time.Duration(*interval) * time.Second):
    check(m, col)
  case <-signalChan:
    // stop
    fmt.Println()
    log.Printf("Stopping...")
    goto stop
  }
}
stop:
```

Of course, as responsible programmers, we care about what happens when the user terminates our programs. So after a call to the `check` method, which doesn't yet exist, we make a signal channel and use `signal.Notify` to ask for the termination signal to be given to the channel, rather than handled automatically. In our infinite `for` loop, we select on two possibilities: either the `timer` channel sends a message or the termination signal channel sends a message. If it's the `timer` channel message, we call `check` again, otherwise we go about terminating the program.

The `time.After` function returns a channel that will send a signal (actually the current time) after the specified time has elapsed. The somewhat confusing `time.Duration(*interval) * time.Second` code simply indicates the amount of time to wait before the signal is sent; the first `*` character is a dereference operator since the `flag.Int` method represents a pointer to an int, and not the int itself. The second `*` character multiplies the interval value by `time.Second`, which gives a value equivalent to the specified interval in seconds. Casting the `*interval int` to `time.Duration` is required so that the compiler knows we are dealing with numbers.

We take a short trip down the memory lane in the preceding code snippet by using the `goto` statement to jump out of the switch and to block loops. We could do away with the `goto` statement altogether and just return when a termination signal is received, but the pattern discussed here allows us to run non-deferred code after the `for` loop, should we wish to.

Updating filedb records

All that is left is for us to implement the `check` function that should call the `Now` method on the `Monitor` type and update the database with new hashes if there are any.

Underneath the `main` function, add the following code:

```
func check(m *backup.Monitor, col *filedb.C) {
  log.Println("Checking...")
  counter, err := m.Now()
  if err != nil {
    log.Fatalln("failed to backup:", err)
  }
  if counter > 0 {
    log.Printf("  Archived %d directories\n", counter)
    // update hashes
    var path path
    col.SelectEach(func(_ int, data []byte) (bool, []byte, bool) {
      if err := json.Unmarshal(data, &path); err != nil {
        log.Println("failed to unmarshal data (skipping):", err)
        return true, data, false
      }
      path.Hash, _ = m.Paths[path.Path]
      newdata, err := json.Marshal(&path)
      if err != nil {
        log.Println("failed to marshal data (skipping):", err)
        return true, data, false
      }
      return true, newdata, false
    })
  } else {
    log.Println("  No changes")
  }
}
```

The `check` function first tells the user that a check is happening, before immediately calling `Now`. If the `Monitor` type did any work for us, which is to ask if it archived any files, we output them to the user and go on to update the database with the new values. The `SelectEach` method allows us to change each record in the collection if we so wish, by returning the replacement bytes. So we `Unmarshal` the bytes to get the path structure, update the hash value and return the marshaled bytes. This ensures that next time we start a `backupd` process, it will do so with the correct hash values.

Testing our solution

Let's see whether our two programs play nicely together and what affects the code inside our `backup` package. You may want to open two terminal windows for this, since we'll be running two programs.

We have already added some paths to the database, so let's use `backup` to see them:

```
./backup -db="./backupdata" list
```

You should see the two test folders; if you don't, refer back to the *Adding paths* section.

```
= ./test [Not yet archived]
= ./test2 [Not yet archived]
```

In another window, navigate to the `backupd` folder and create our two test folders called `test` and `test2`.

Build `backupd` using the usual method:

```
go build -o backupd
```

Assuming all is well, we can now start the backup process being sure to point the `db` path to the same path as we used for the `backup` program, and specify that we want to use a new folder called `archive` to store the ZIP files. For testing purposes, let's specify an interval of `5` seconds to save time:

```
./backupd -db="../backup/backupdata/" -archive="./archive" -
interval=5
```

Immediately, `backupd` should check the folders, calculate the hashes, notice that they are different (to `Not yet archived`), and initiate the archive process for both folders. It will print the output telling us this:

```
Checking...
Archived 2 directories
```

Open the newly created `archive` folder inside `backup/cmds/backupd` and notice it has created two subfolders: `test` and `test2`. Inside those are compressed archive versions of the empty folders. Feel free to unzip one and see; not very exciting so far.

Meanwhile, back in the terminal window, `backupd` has been checking the folders again for changes:

```
Checking...
  No changes
Checking...
  No changes
```

In your favorite text editor, create a new text file inside the `test2` folder containing the word `test`, and save it as `one.txt`. After a few seconds, you will see that `backupd` has noticed the new file and created another snapshot inside the `archive/test2` folder.

Of course, it has a different filename because the time is different, but if you unzip it you will notice that it has indeed created a compressed archive version of the folder.

Play around with the solution by taking the following actions:

- Change the contents of the `one.txt` file
- Add a file to the `test` folder too
- Delete a file

Summary

In this chapter, we successfully built a very powerful and flexible backup system for your code projects. You can see how simple it would be to extend or modify the behavior of these programs. The scope for potential problems that you could go on to solve is limitless.

Rather than having a local archive destination folder like we did in the previous section, imagine mounting a network storage device and using that instead. Suddenly, you have off-site (or at least off-machine) backups of those vital files. You could easily set a Dropbox folder as the archive destination, which would mean not only do you get access to the snapshots yourself, but also a copy is stored in the cloud and can even be shared with other users.

Extending the `Archiver` interface to support `Restore` operations (which would just use the `encoding/zip` package to unzip the files) allows you to build tools that can peer inside the archives and access the changes of individual files much like Time Machine allows you to do. Indexing the files gives you full search across the entire history of your code, much like GitHub does.

Since the filenames are timestamps, you could have backed up retiring old archives to less active storage mediums, or summarized the changes into a daily dump.

Obviously, backup software exists, is well tested, and used through the world and it may be a smart move to focus on solving problems that haven't yet been solved. But when it requires such little effort to write small programs to get things done, it is often worth doing because of the control it gives you. When you write the code, you can get exactly what you want without compromise, and it's down to each individual to make that call.

Specifically in this chapter, we explored how easy Go's standard library makes it to interact with the filesystem: opening files for reading, creating new files, and making directories. The `os` package mixed in with the powerful types from the `io` package, blended further with capabilities like `encoding/zip` and others, gives a clear example of how extremely simple Go interfaces can be composed to deliver very powerful results.

Module 3

Mastering Concurrency in Go

Discover and harness Go's powerful concurrency features to develop and build fast, scalable network systems

1
An Introduction to Concurrency in Go

While Go is both a great general purpose and low-level systems language, one of its primary strengths is the built-in concurrency model and tools. Many other languages have third-party libraries (or extensions), but inherent concurrency is something unique to modern languages, and it is a core feature of Go's design.

Though there's no doubt that Go excels at concurrency—as we'll see in this book—what it has that many other languages lack is a robust set of tools to test and build concurrent, parallel, and distributed code.

Enough talk about Go's marvelous concurrency features and tools, let's jump in.

Introducing goroutines

The primary method of handling concurrency is through a goroutine. Admittedly, our first piece of concurrent code (mentioned in the preface) didn't do a whole lot, simply spitting out alternating "hello"s and "world"s until the entire task was complete.

Here is that code once again:

```
package main

import (
  "fmt"
  "time"
)

type Job struct {
  i int
  max int
  text string
}

func outputText(j *Job) {
  for j.i < j.max {
    time.Sleep(1 * time.Millisecond)
    fmt.Println(j.text)
    j.i++
  }
}

func main() {
  hello := new(Job)
  world := new(Job)

  hello.text = "hello"
  hello.i = 0
  hello.max = 3

  world.text = "world"
  world.i = 0
  world.max = 5

  go outputText(hello)
  outputText(world)

}
```

Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at `http://www. packtpub.com`. If you purchased this book elsewhere, you can visit `http://www.packtpub.com/support` and register to have the files e-mailed directly to you.

But, if you think back to our real-world example of planning a surprise party for your grandmother, that's exactly how things often have to be managed with limited or finite resources. This asynchronous behavior is critical for some applications to run smoothly, although our example essentially ran in a vacuum.

You may have noticed one quirk in our early example: despite the fact that we called the `outputText()` function on the `hello` struct first, our output started with the `world` struct's text value. Why is that?

Being asynchronous, when a goroutine is invoked, it waits for the blocking code to complete before concurrency begins. You can test this by replacing the `outputText()` function call on the `world` struct with a goroutine, as shown in the following code:

```
go outputText(hello)
go outputText(world)
```

If you run this, you will get no output because the main function ends while the asynchronous goroutines are running. There are a couple of ways to stop this to see the output before the main function finishes execution and the program exits. The classic method simply asks for user input before execution, allowing you to directly control when the application finishes. You can also put an infinite loop at the end of your main function, as follows:

```
for {}
```

Better yet, Go also has a built-in mechanism for this, which is the `WaitGroup` type in the `sync` package.

If you add a `WaitGroup` struct to your code, it can delay execution of the main function until after all goroutines are complete. In simple terms, it lets you set a number of required iterations to get a completed response from the goroutines before allowing the application to continue. Let's look at a minor modification to our "Hello World" application in the following section.

A patient goroutine

From here, we'll implement a `WaitGroup` struct to ensure our goroutines run entirely before moving on with our application. In this case, when we say patient, it's in contrast to the way we've seen goroutines run outside of a parent method with our previous example. In the following code, we will implement our first `Waitgroup` struct:

```
package main

import (
  "fmt"
  "sync"
  "time"
)
```

```
type Job struct {
  i int
  max int
  text string
}

func outputText(j *Job, goGroup *sync.WaitGroup) {
  for j.i < j.max {
    time.Sleep(1 * time.Millisecond)
    fmt.Println(j.text)
    j.i++
  }
  goGroup.Done()
}

func main() {

  goGroup := new(sync.WaitGroup)
  fmt.Println("Starting")

  hello := new(Job)
  hello.text = "hello"
  hello.i = 0
  hello.max = 2

  world := new(Job)
  world.text = "world"
  world.i = 0
  world.max = 2

sync.WaitGroup.Add

go outputText(hello, goGroup)

go outputText(world, goGroup)

  goGroup.Add(2)
  goGroup.Wait()

}
```

Let's look at the changes in the following code:

```
  goGroup := new(sync.WaitGroup)
```

Here, we declared a `WaitGroup` struct named `goGroup`. This variable will receive note that our goroutine function has completed *x* number of times before allowing the program to exit. Here's an example of sending such an expectation in `WaitGroup`:

```
goGroup.Add(2)
```

The `Add()` method specifies how many `Done` messages `goGroup` should receive before satisfying its wait. Here, we specified `2` because we have two functions running asynchronously. If you had three goroutine members and still called two, you may see the output of the third. If you added a value more than two to `goGroup`, for example, `goGroup.Add(3)`, then `WaitGroup` would wait forever and deadlock.

With that in mind, you shouldn't manually set the number of goroutines that need to wait; this is ideally handled computationally or explicitly in a range. This is how we tell `WaitGroup` to wait:

```
goGroup.Wait()
```

Now, we wait. This code will fail for the same reason `goGroup.Add(3)` failed; the `goGroup` struct never receives messages that our goroutines are done. So, let's do this as shown in the following code snippet:

```
func outputText(j *Job, goGroup *sync.WaitGroup) {
  for j.i < j.max {
    time.Sleep(1 * time.Millisecond)
    fmt.Println(j.text)
    j.i++
  }
  goGroup.Done()
}
```

We've only made two changes to our `outputText()` function from the preface. First, we added a pointer to our `goGroup` as the second function argument. Then, when all our iterations were complete, we told `goGroup` that they are all done.

Implementing the defer control mechanism

While we're here, we should take a moment and talk about defer. Go has an elegant implementation of the defer control mechanism. If you've used defer (or something functionally similar) in other languages, this will seem familiar—it's a useful way of delaying the execution of a statement until the rest of the function is complete.

For the most part, this is syntactical sugar that allows you to see related operations together, even though they won't execute together. If you've ever written something similar to the following pseudocode, you'll know what I mean:

```
x = file.open('test.txt')
int longFunction() {
…
}
x.close();
```

You probably know the kind of pain that can come from large "distances" separating related bits of code. In Go, you can actually write the code similar to the following:

```
package main

import(
"os"
)

func main() {

  file, _ := os.Create("/defer.txt")

  defer file.Close()

  for {

    break

  }


}
```

There isn't any actual functional advantage to this other than making clearer, more readable code, but that's a pretty big plus in itself. Deferred calls are executed reverse of the order in which they are defined, or last-in-first-out. You should also take note that any data passed by reference may be in an unexpected state.

For example, refer to the following code snippet:

```
func main() {

  aValue := new(int)

  defer fmt.Println(*aValue)

  for i := 0; i < 100; i++ {
    *aValue++
  }

}
```

This will return `0`, and not `100`, as it is the default value for an integer.

> *Defer* is not the same as *deferred* (or futures/promises) in other languages. We'll talk about Go's implementations and alternatives to futures and promises in *Chapter 2, Understanding the Concurrency Model*.

Using Go's scheduler

With a lot of concurrent and parallel applications in other languages, the management of both soft and hard threads is handled at the operating system level. This is known to be inherently inefficient and expensive as the OS is responsible for context switching, among multiple processes. When an application or process can manage its own threads and scheduling, it results in faster runtime. The threads granted to our application and Go's scheduler have fewer OS attributes that need to be considered in context to switching, resulting in less overhead.

If you think about it, this is self-evident—the more you have to juggle, the slower it is to manage all of the balls. Go removes the natural inefficiency of this mechanism by using its own scheduler.

There's really only one quirk to this, one that you'll learn very early on: if you don't ever yield to the main thread, your goroutines will perform in unexpected ways (or won't perform at all).

Another way to look at this is to think that a goroutine must be blocked before concurrency is valid and can begin. Let's modify our example and include some file I/O to log to demonstrate this quirk, as shown in the following code:

```
package main

import (
  "fmt"
  "time"
  "io/ioutil"
)



type Job struct {
  i int
  max int
  text string
}

func outputText(j *Job) {
  fileName := j.text + ".txt"
  fileContents := ""
  for j.i < j.max {
    time.Sleep(1 * time.Millisecond)
    fileContents += j.text
    fmt.Println(j.text)
    j.i++
  }
  err := ioutil.WriteFile(fileName, []byte(fileContents), 0644)
  if (err != nil) {
    panic("Something went awry")
  }

}

func main() {

  hello := new(Job)
  hello.text = "hello"
  hello.i = 0
  hello.max = 3
```

```
    world := new(Job)
    world.text = "world"
    world.i = 0
    world.max = 5


    go outputText(hello)
    go outputText(world)

  }
```

In theory, all that has changed is that we're now using a file operation to log each operation to a distinct file (in this case, `hello.txt` and `world.txt`). However, if you run this, no files are created.

In our last example, we used a `sync.WaitSync` struct to force the main thread to delay execution until asynchronous tasks were complete. While this works (and elegantly), it doesn't really explain *why* our asynchronous tasks fail. As mentioned before, you can also utilize blocking code to prevent the main thread from completing before its asynchronous tasks.

Since the Go scheduler manages context switching, each goroutine must yield control back to the main thread to schedule all of these asynchronous tasks. There are two ways to do this manually. One method, and probably the ideal one, is the `WaitGroup` struct. Another is the `Gosched()` function in the runtime package.

The `Gosched()` function temporarily yields the processor and then returns to the current goroutine. Consider the following code as an example:

```
package main
import(
    _"runtime"
    "fmt"
)

func showNumber(num int) {
  fmt.Println(num)
}

func main() {
  iterations := 10

  for i := 0; i<=iterations; i++ {
    go showNumber(i)

  }
  //runtime.Gosched()
  fmt.Println("Goodbye!")

}
```

Run this with `runtime.Gosched()` commented out and the underscore before `"runtime"` removed, and you'll see only `Goodbye!`. This is because there's no guarantee as to how many goroutines, if any, will complete before the end of the `main()` function.

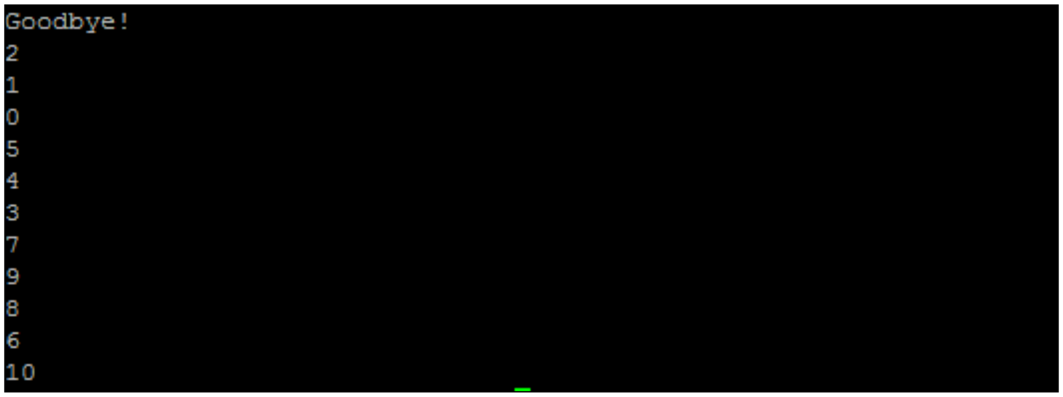
As we learned earlier, you can explicitly wait for a finite set number of goroutines before ending the execution of the application. However, `Gosched()` allows (in most cases) for the same basic functionality. Remove the comment before `runtime.Gosched()`, and you should get 0 through 10 printed before `Goodbye!`.

Just for fun, try running this code on a multicore server and modify your max processors using `runtime.GOMAXPROCS()`, as follows:

```
    func main() {

      runtime.GOMAXPROCS(2)
```

Also, push your `runtime.Gosched()` to the absolute end so that all goroutines have a chance to run before `main` ends.

Got something unexpected? That's not unexpected! You may end up with a totally jostled execution of your goroutines, as shown in the following screenshot:



```
Goodbye!
2
1
0
5
4
3
7
9
8
6
10
```

Although it's not entirely necessary to demonstrate how juggling your goroutines with multiple cores can be vexing, this is one of the simplest ways to show exactly why it's important to have communication between them (and the Go scheduler).

You can debug the parallelism of this using `GOMAXPROCS > 1`, enveloping your goroutine call with a timestamp display, as follows:

```
tstamp := strconv.FormatInt(time.Now().UnixNano(), 10)
fmt.Println(num, tstamp)
```

Remember to import the `time` and `strconv` parent packages here.

This will also be a good place to see concurrency and compare it to parallelism in action. First, add a one-second delay to the `showNumber()` function, as shown in the following code snippet:

```
func showNumber(num int) {
  tstamp := strconv.FormatInt(time.Now().UnixNano(), 10)
  fmt.Println(num,tstamp)
  time.Sleep(time.Millisecond * 10)
}
```

Then, remove the goroutine call before the `showNumber()` function with `GOMAXPROCS(0)`, as shown in the following code snippet:

```
runtime.GOMAXPROCS(0)
iterations := 10

for i := 0; i<=iterations; i++ {
  showNumber(i)
}
```

As expected, you get 0-10 with 10-millisecond delays between them followed by `Goodbye!` as an output. This is straight, serial computing.

Next, let's keep `GOMAXPROCS` at zero for a single thread, but restore the goroutine as follows:

```
go showNumber(i)
```

This is the same process as before, except for the fact that everything will execute within the same general timeframe, demonstrating the concurrent nature of execution. Now, go ahead and change your `GOMAXPROCS` to two and run again. As mentioned earlier, there is only one (or possibly two) timestamp, but the order has changed because everything is running simultaneously.

Goroutines aren't (necessarily) thread-based, but they feel like they are. When Go code is compiled, the goroutines are multiplexed across available threads. It's this very reason why Go's scheduler needs to know what's running, what needs to finish before the application's life ends, and so on. If the code has two threads to work with, that's what it will use.

Using system variables

So what if you want to know how many threads your code has made available to you?

Go has an environment variable returned from the runtime package function `GOMAXPROCS`. To find out what's available, you can write a quick application similar to the following code:

```
package main

import (
  "fmt"
  "runtime"
)

func listThreads() int {

  threads := runtime.GOMAXPROCS(0)
  return threads
}

func main() {
  runtime.GOMAXPROCS(2)
  fmt.Printf("%d thread(s) available to Go.", listThreads())

}
```

A simple Go build on this will yield the following output:

```
2 thread(s) available to Go.
```

The `0` parameter (or no parameter) delivered to `GOMAXPROCS` means no change is made. You can put another number in there, but as you might imagine, it will only return what is actually available to Go. You cannot exceed the available cores, but you can limit your application to use less than what's available.

The `GOMAXPROCS()` call itself returns an integer that represents the *previous* number of processors available. In this case, we first set it to two and then set it to zero (no change), returning two.

It's also worth noting that increasing `GOMAXPROCS` can sometimes *decrease* the performance of your application.

As there are context-switching penalties in larger applications and operating systems, increasing the number of threads used means goroutines can be shared among more than one, and the lightweight advantage of goroutines might be sacrificed.

If you have a multicore system, you can test this pretty easily with Go's internal benchmarking functionality. We'll take a closer look at this functionality in *Chapter 5, Locks, Blocks, and Better Channels,* and *Chapter 7, Performance and Scalability*.

The runtime package has a few other very useful environment variable return functions, such as `NumCPU`, `NumGoroutine`, `CPUProfile`, and `BlockProfile`. These aren't just handy to debug, they're also good to know how to best utilize your resources. This package also plays well with the reflect package, which deals with metaprogramming and program self-analysis. We'll touch on that in more detail later in *Chapter 9, Logging and Testing Concurrency in Go*, and *Chapter 10, Advanced Concurrency and Best Practices*.

Understanding goroutines versus coroutines

At this point, you may be thinking, "Ah, goroutines, I know these as coroutines." Well, yes and no.

A coroutine is a cooperative task control mechanism, but in its most simplistic sense, a coroutine is not concurrent. While coroutines and goroutines are utilized in similar ways, Go's focus on concurrency provides a lot more than just state control and yields. In the examples we've seen so far, we have what we can call *dumb* goroutines. Although they operate in the same time and address space, there's no real communication between the two. If you look at coroutines in other languages, you may find that they are often not necessarily concurrent or asynchronous, but rather they are step-based. They yield to `main()` and to each other, but two coroutines might not necessarily communicate between each other, relying on a centralized, explicitly written data management system.

The original coroutine

Coroutines were first described for COBOL by Melvin Conway. In his paper, *Design of a Separable Transition-Diagram Compiler*, he suggested that the purpose of a coroutine was to take a program broken apart into subtasks and allow them to operate independently, sharing only small pieces of data.

Goroutines can sometimes violate the basic tenets of Conway's coroutines. For example, Conway suggested that there should be only a unidirectional path of execution; in other words, A followed by B, then C, and then D, and so on, where each represents an application chunk in a coroutine. We know that goroutines can be run in parallel and can execute in a seemingly arbitrary order (at least without direction). To this point, our goroutines have not shared any information either; they've simply executed in a shared pattern.

Implementing channels

So far, we've dabbled in concurrent processes that are capable of doing a lot but not effectively communicating with each other. In other words, if you have two processes occupying the same processing time and sharing the same memory and data, you must have a way of knowing which process is in which place as part of a larger task.

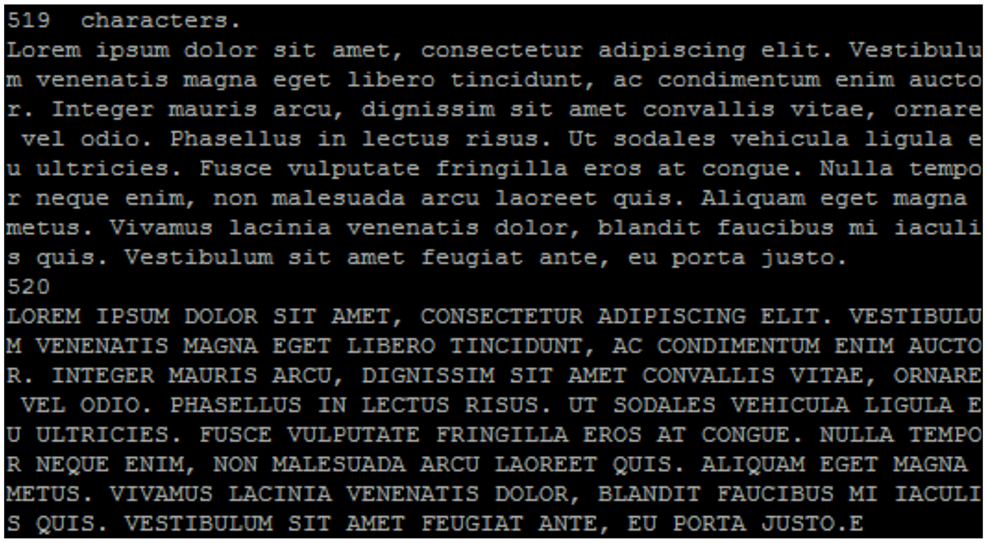
Take, for example, an application that must loop through one paragraph of Lorem Ipsum and capitalize each letter, then write the result to a file. Of course, we will not really need a concurrent application to do this (and in fact, it's an endemic function of almost any language that handles strings), but it's a quick way to demonstrate the potential limitations of isolated goroutines. Shortly, we'll turn this primitive example into something more practical, but for now, here's the beginning of our capitalization example:

```
package main

import (
  "fmt"
  "runtime"
  "strings"
)
```

```
var loremIpsum string
var finalIpsum string
var letterSentChan chan string

func deliverToFinal(letter string, finalIpsum *string) {
  *finalIpsum += letter
}

func capitalize(current *int, length int, letters []byte,
  finalIpsum *string) {
  for *current < length {
    thisLetter := strings.ToUpper(string(letters[*current]))

    deliverToFinal(thisLetter, finalIpsum)
    *current++
  }
}

func main() {

  runtime.GOMAXPROCS(2)

  index := new(int)
  *index = 0
  loremIpsum = "Lorem ipsum dolor sit amet, consectetur adipiscing
  elit. Vestibulum venenatis magna eget libero tincidunt, ac
  condimentum enim auctor. Integer mauris arcu, dignissim sit amet
  convallis vitae, ornare vel odio. Phasellus in lectus risus. Ut
  sodales vehicula ligula eu ultricies. Fusce vulputate fringilla
  eros at congue. Nulla tempor neque enim, non malesuada arcu
  laoreet quis. Aliquam eget magna metus. Vivamus lacinia
  venenatis dolor, blandit faucibus mi iaculis quis. Vestibulum
  sit amet feugiat ante, eu porta justo."

  letters := []byte(loremIpsum)
  length := len(letters)

  go capitalize(index, length, letters, &finalIpsum)
```

```
    go func() {
      go capitalize(index, length, letters, &finalIpsum)
    }()

    fmt.Println(length, " characters.")
    fmt.Println(loremIpsum)
    fmt.Println(*index)
    fmt.Println(finalIpsum)

  }
```

If we run this with some degree of parallelism here but no communication between our goroutines, we'll end up with a jumbled mess of text, as shown in the following screenshot:



```
519  characters.
Lorem ipsum dolor sit amet, consectetur adipiscing elit. Vestibulu
m venenatis magna eget libero tincidunt, ac condimentum enim aucto
r. Integer mauris arcu, dignissim sit amet convallis vitae, ornare
 vel odio. Phasellus in lectus risus. Ut sodales vehicula ligula e
u ultricies. Fusce vulputate fringilla eros at congue. Nulla tempo
r neque enim, non malesuada arcu laoreet quis. Aliquam eget magna
metus. Vivamus lacinia venenatis dolor, blandit faucibus mi iaculi
s quis. Vestibulum sit amet feugiat ante, eu porta justo.
520
LOREM IPSUM DOLOR SIT AMET, CONSECTETUR ADIPISCING ELIT. VESTIBULU
M VENENATIS MAGNA EGET LIBERO TINCIDUNT, AC CONDIMENTUM ENIM AUCTO
R. INTEGER MAURIS ARCU, DIGNISSIM SIT AMET CONVALLIS VITAE, ORNARE
 VEL ODIO. PHASELLUS IN LECTUS RISUS. UT SODALES VEHICULA LIGULA E
U ULTRICIES. FUSCE VULPUTATE FRINGILLA EROS AT CONGUE. NULLA TEMPO
R NEQUE ENIM, NON MALESUADA ARCU LAOREET QUIS. ALIQUAM EGET MAGNA
METUS. VIVAMUS LACINIA VENENATIS DOLOR, BLANDIT FAUCIBUS MI IACULI
S QUIS. VESTIBULUM SIT AMET FEUGIAT ANTE, EU PORTA JUSTO.E
```

Due to the demonstrated unpredictability of concurrent scheduling in Go, it may take many iterations to get this exact output. In fact, you may never get the exact output.

This won't do, obviously. So how do we best structure this application? The missing piece here is synchronization, but we could also do with a better design pattern.

Here's another way to break this problem down into pieces. Instead of having two processes handling the same thing in parallel, which is rife with risk, let's have one process that takes a letter from the `loremIpsum` string and capitalizes it, and then pass it onto another process to add it to our `finalIpsum` string.

You can envision this as two people sitting at two desks, each with a stack of letters. Person A is responsible to take a letter and capitalize it. He then passes the letter onto person B, who then adds it to the `finalIpsum` stack. To do this, we'll implement a channel in our code in an application tasked with taking text (in this case, the first line of Abraham Lincoln's Gettysburg address) and capitalizing each letter.

Channel-based sorting at the letter capitalization factory

Let's take the last example and do something (slightly) more purposeful by attempting to capitalize the preamble of Abraham Lincoln's Gettysburg address while mitigating the sometimes unpredictable effect of concurrency in Go, as shown in the following code:

```
package main

import(
  "fmt"
  "sync"
  "runtime"
  "strings"
)

var initialString string
var finalString string

var stringLength int

func addToFinalStack(letterChannel chan string, wg
  *sync.WaitGroup) {
  letter := <-letterChannel
  finalString += letter
  wg.Done()
}


func capitalize(letterChannel chan string, currentLetter string,
  wg *sync.WaitGroup) {

  thisLetter := strings.ToUpper(currentLetter)
```

```
    wg.Done()
    letterChannel <- thisLetter
  }


  func main() {

    runtime.GOMAXPROCS(2)
    var wg sync.WaitGroup

    initialString = "Four score and seven years ago our fathers
    brought forth on this continent, a new nation, conceived in
    Liberty, and dedicated to the proposition that all men are
    created equal."
    initialBytes := []byte(initialString)

    var letterChannel chan string = make(chan string)

    stringLength = len(initialBytes)



    for i := 0; i < stringLength; i++ {
      wg.Add(2)

      go capitalize(letterChannel, string(initialBytes[i]), &wg)
      go addToFinalStack(letterChannel, &wg)

      wg.Wait()
    }


    fmt.Println(finalString)

  }
```

You'll note that we even bumped this up to a duo-core process and ended up with the following output:

```
go run alpha-channel.go
FOUR SCORE AND SEVEN YEARS AGO OUR FATHERS BROUGHT FORTH ON THIS
  CONTINENT, A NEW NATION, CONCEIVED IN LIBERTY, AND DEDICATED TO THE
  PROPOSITION THAT ALL MEN ARE CREATED EQUAL.
```

The output is just as we expected. It's worth reiterating that this example is overkill of the most extreme kind, but we'll parlay this functionality into a usable practical application shortly.

So what's happening here? First, we reimplemented the `sync.WaitGroup` struct to allow all of our concurrent code to execute while keeping the main thread alive, as shown in the following code snippet:

```
var wg sync.WaitGroup
...
for i := 0; i < stringLength; i++ {
  wg.Add(2)

  go capitalize(letterChannel, string(initialBytes[i]), &wg)
  go addToFinalStack(letterChannel, &wg)

  wg.Wait()
}
```

We allow each goroutine to tell the `WaitGroup` struct that we're done with the step. As we have two goroutines, we queue two `Add()` methods to the `WaitGroup` struct. Each goroutine is responsible to announce that it's done.

Next, we created our first channel. We instantiate a channel with the following line of code:

```
  var letterChannel chan string = make(chan string)
```

This tells Go that we have a channel that will send and receive a string to various procedures/goroutines. This is essentially the manager of all of the goroutines. It is also responsible to send and receive data to goroutines and manage the order of execution. As we mentioned earlier, the ability of channels to operate with internal context switching and without reliance on multithreading permits them to operate very quickly.

There is a built-in limit to this functionality. If you design non-concurrent or blocking code, you will effectively remove concurrency from goroutines. We will talk more about this shortly.

We run two separate goroutines through `letterChannel`: `capitalize()` and `addToFinalStack()`. The first one simply takes a single byte from a byte array constructed from our string and capitalizes it. It then returns the byte to the channel as shown in the following line of code:

```
letterChannel <- thisLetter
```

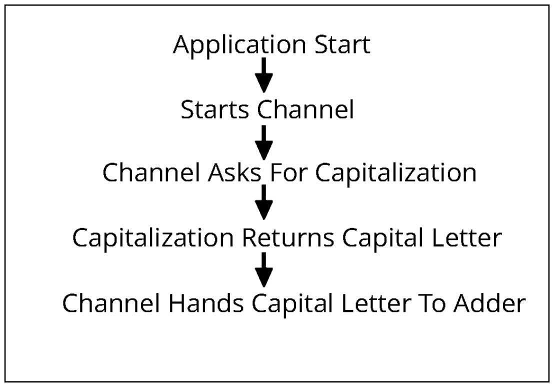
All communication across a channel happens in this fashion. The `<-` symbol syntactically tells us that data will be sent back to (or back through) a channel. It's never necessary to do anything with this data, but the most important thing to know is that a channel can be blocking, at least per thread, until it receives data back. You can test this by creating a channel and then doing absolutely nothing of value with it, as shown in the following code snippet:

```
package main

func doNothing()(string) {

  return "nothing"
}

func main() {

  var channel chan string = make(chan string)
  channel <- doNothing()

}
```

As nothing is sent along the channel and no goroutine is instantiated, this results in a deadlock. You can fix this easily by creating a goroutine and by bringing the channel into the global space by creating it outside of `main()`.

For the sake of clarity, our example here uses a local scope channel. Keeping these global whenever possible removes a lot of cruft, particularly if you have a lot of goroutines, as references to the channel can clutter up your code in a hurry.

For our example as a whole, you can look at it as is shown in the following figure:


Application Start
Starts Channel
Channel Asks For Capitalization
Capitalization Returns Capital Letter
Channel Hands Capital Letter To Adder

Cleaning up our goroutines

You may be wondering why we need a `WaitGroup` struct when using channels. After all, didn't we say that a channel gets blocked until it receives data? This is true, but it requires one other piece of syntax.

A nil or uninitialized channel will always get blocked. We will discuss the potential uses and pitfalls of this in *Chapter 7*, *Performance and Scalability*, and *Chapter 10*, *Advanced Concurrency and Best Practices*.

You have the ability to dictate how a channel blocks the application based on a second option to the `make` command by dictating the channel buffer.

Buffered or unbuffered channels

By default, channels are unbuffered, which means they will accept anything sent on them if there is a channel ready to receive. It also means that every channel call will block the execution of the application. By providing a buffer, the channel will only block the application when many returns have been sent.

A buffered channel is synchronous. To guarantee asynchronous performance, you'll want to experiment by providing a buffer length. We'll look at ways to ensure our execution falls as we expect in the next chapter.

[Go's channel system is based on **Communicating Sequential Processes** (**CSP**), a formal language to design concurrent patterns and multiprocessing. You will likely encounter CSP on its own when people describe goroutines and channels.]

Using the select statement

One of the issues with first implementing channels is that whereas goroutines were formerly the method of simplistic and concurrent execution of code, we now have a single-purpose channel that dictates application logic across the goroutines. Sure, the channel is the traffic manager, but it never knows when traffic is coming, when it's no longer coming, and when to go home, unless being explicitly told. It waits passively for communication and can cause problems if it never receives any.

Go has a select control mechanism, which works just as effectively as a `switch` statement does, but on channel communication instead of variable values. A `switch` statement modifies execution based on the value of a variable, and `select` reacts to actions and communication across a channel. You can use this to orchestrate and arrange the control flow of your application. The following code snippet is our traditional `switch`, familiar to Go users and common among other languages:

```
switch {

  case 'x':

  case 'y':

}
```

The following code snippet represents the `select` statement:

```
select {

  case <- channelA:

  case <- channelB:

}
```

In a `switch` statement, the right-hand expression represents a value; in `select`, it represents a receive operation on a channel. A `select` statement will block the application until some information is sent along the channel. If nothing is sent ever, the application deadlocks and you'll get an error to that effect.

If two receive operations are sent at the same time (or if two cases are otherwise met), Go will evaluate them in an unpredictable fashion.

So, how might this be useful? Let's look at a modified version of the letter capitalization application's main function:

```
package main

import(
  "fmt"
  "strings"
)
```

```
var initialString string
var initialBytes []byte
var stringLength int
var finalString string
var lettersProcessed int
var applicationStatus bool
var wg sync.WaitGroup

func getLetters(gQ chan string) {

  for i := range initialBytes {
    gQ <- string(initialBytes[i])

  }

}

func capitalizeLetters(gQ chan string, sQ chan string) {

  for {
    if lettersProcessed >= stringLength {
      applicationStatus = false
      break
    }
    select {
      case letter := <- gQ:
        capitalLetter := strings.ToUpper(letter)
        finalString += capitalLetter
        lettersProcessed++
    }
  }
}

func main() {

  applicationStatus = true;

  getQueue := make(chan string)
  stackQueue := make(chan string)
```

```
    initialString = "Four score and seven years ago our fathers
    brought forth on this continent, a new nation, conceived in
    Liberty, and dedicated to the proposition that all men are
    created equal."
    initialBytes = []byte(initialString)
    stringLength = len(initialString)
    lettersProcessed = 0

    fmt.Println("Let's start capitalizing")


    go getLetters(getQueue)
    capitalizeLetters(getQueue,stackQueue)

    close(getQueue)
    close(stackQueue)

    for {

      if applicationStatus == false {
        fmt.Println("Done")
        fmt.Println(finalString)
        break
      }

    }
  }
```

The primary difference here is we now have a channel that listens for data across two functions running concurrently, `getLetters` and `capitalizeLetters`. At the bottom, you'll see a `for{}` loop that keeps the main active until the `applicationStatus` variable is set to `false`. In the following code, we pass each of these bytes as a string through the Go channel:

```
  func getLetters(gQ chan string) {

    for i := range initialBytes {
      gQ <- string(initialBytes[i])

    }

  }
```

The `getLetters` function is our primary goroutine that fetches individual letters from the byte array constructed from Lincoln's line. As the function iterates through each byte, it sends the letter through the `getQueue` channel.

On the receiving end, we have `capitalizeLetters` that takes each letter as it's sent across the channel, capitalizes it, and appends to our `finalString` variable. Let's take a look at this:

```
func capitalizeLetters(gQ chan string, sQ chan string) {

  for {
    if lettersProcessed >= stringLength {
      applicationStatus = false
      break
    }
    select {
      case letter := <- gQ:
        capitalLetter := strings.ToUpper(letter)
        finalString += capitalLetter
        lettersProcessed++
    }
  }
}
```

It's critical that all channels are closed at some point or our application will hit a deadlock. If we never break the `for` loop here, our channel will be left waiting to receive from a concurrent process, and the program will deadlock. We manually check to see that we've capitalized all letters and only then break the loop.

Closures and goroutines

You may have noticed the anonymous goroutine in Lorem Ipsum:

```
  go func() {
    go capitalize(index, length, letters, &finalIpsum)
  }()
```

While it isn't always ideal, there are plenty of places where inline functions work best in creating a goroutine.

The easiest way to describe this is to say that a function isn't big or important enough to deserve a named function, but the truth is, it's more about readability. If you have dealt with lambdas in other languages, this probably doesn't need much explanation, but try to reserve these for quick inline functions.

In the earlier examples, the closure works largely as a wrapper to invoke a `select` statement or to create anonymous goroutines that will feed the `select` statement.

Since functions are first-class citizens in Go, not only can you utilize inline or anonymous functions directly in your code, but you can also pass them to and from other functions.

Here's an example that passes a function's result as a return value, keeping the state resolute outside of that returned function. In this, we'll return a function as a variable and iterate initial values on the returned function. The initial argument will accept a string that will be trimmed by word length with each successive call of the returned function.

```
import(
  "fmt"
  "strings"
)

func shortenString(message string) func() string {

  return func() string {
    messageSlice := strings.Split(message," ")
    wordLength := len(messageSlice)
    if wordLength < 1 {
      return "Nothingn Left!"
    }else {
      messageSlice = messageSlice[:(wordLength-1)]
      message = strings.Join(messageSlice, " ")
      return message
    }
  }
}

func main() {

  myString := shortenString("Welcome to concurrency in Go! ...")
```

```
    fmt.Println(myString())
    fmt.Println(myString())
    fmt.Println(myString())
    fmt.Println(myString())
    fmt.Println(myString())
    fmt.Println(myString())
  }
```

Once initialized and returned, we set the message variable, and each successive run of the returned method iterates on that value. This functionality allows us to eschew running a function multiple times on returned values or loop unnecessarily when we can very cleanly handle this with a closure as shown.

Building a web spider using goroutines and channels

Let's take the largely useless capitalization application and do something practical with it. Here, our goal is to build a rudimentary spider. In doing so, we'll accomplish the following tasks:

- Read five URLs
- Read those URLs and save the contents to a string
- Write that string to a file when all URLs have been scanned and read

These kinds of applications are written every day, and they're the ones that benefit the most from concurrency and non-blocking code.

It probably goes without saying, but this is not a particularly elegant web scraper. For starters, it only knows a few start points—the five URLs that we supply it. Also, it's neither recursive nor is it thread-safe in terms of data integrity.

That said, the following code works and demonstrates how we can use channels and the `select` statements:

```
package main

import(
  "fmt"
  "io/ioutil"
  "net/http"
  "time"
)
```

```
var applicationStatus bool
var urls []string
var urlsProcessed int
var foundUrls []string
var fullText string
var totalURLCount int
var wg sync.WaitGroup

var v1 int
```

First, we have our most basic global variables that we'll use for the application state. The `applicationStatus` variable tells us that our spider process has begun and `urls` is our slice of simple string URLs. The rest are idiomatic data storage variables and/or application flow mechanisms. The following code snippet is our function to read the URLs and pass them across the channel:

```
func readURLs(statusChannel chan int, textChannel chan string) {

  time.Sleep(time.Millisecond * 1)
  fmt.Println("Grabbing", len(urls), "urls")
  for i := 0; i < totalURLCount; i++ {

    fmt.Println("Url", i, urls[i])
    resp, _ := http.Get(urls[i])
    text, err := ioutil.ReadAll(resp.Body)

    textChannel <- string(text)

    if err != nil {
      fmt.Println("No HTML body")
    }

    statusChannel <- 0

  }

}
```

The `readURLs` function assumes `statusChannel` and `textChannel` for communication and loops through the `urls` variable slice, returning the text on `textChannel` and a simple ping on `statusChannel`. Next, let's look at the function that will append scraped text to the full text:

```
func addToScrapedText(textChannel chan string, processChannel chan
  bool) {
    for {
      select {
      case pC := <-processChannel:
        if pC == true {
          // hang on
        }
        if pC == false {

          close(textChannel)
          close(processChannel)
        }
      case tC := <-textChannel:
        fullText += tC

      }

    }

  }
```

We use the `addToScrapedText` function to accumulate processed text and add it to a master text string. We also close our two primary channels when we get a kill signal on our `processChannel`. Let's take a look at the `evaluateStatus()` function:

```
  func evaluateStatus(statusChannel chan int, textChannel chan
    string, processChannel chan bool) {

    for {
      select {
      case status := <-statusChannel:

        fmt.Print(urlsProcessed, totalURLCount)
        urlsProcessed++
        if status == 0 {

          fmt.Println("Got url")

        }
        if status == 1 {
          close(statusChannel)
        }
        if urlsProcessed == totalURLCount {
          fmt.Println("Read all top-level URLs")
          processChannel <- false
          applicationStatus = false

        }
      }

    }
  }
```

At this juncture, all that the `evaluateStatus` function does is determine what's happening in the overall scope of the application. When we send a `0` (our aforementioned ping) through this channel, we increment our `urlsProcessed` variable. When we send a `1`, it's a message that we can close the channel. Finally, let's look at the `main` function:

```
func main() {
  applicationStatus = true
  statusChannel := make(chan int)
  textChannel := make(chan string)
  processChannel := make(chan bool)
  totalURLCount = 0

  urls = append(urls, "http://www.mastergoco.com/index1.html")
  urls = append(urls, "http://www.mastergoco.com/index2.html")
  urls = append(urls, "http://www.mastergoco.com/index3.html")
  urls = append(urls, "http://www.mastergoco.com/index4.html")
  urls = append(urls, "http://www.mastergoco.com/index5.html")

  fmt.Println("Starting spider")

  urlsProcessed = 0
  totalURLCount = len(urls)

  go evaluateStatus(statusChannel, textChannel, processChannel)

  go readURLs(statusChannel, textChannel)

  go addToScrapedText(textChannel, processChannel)

  for {
    if applicationStatus == false {
      fmt.Println(fullText)
      fmt.Println("Done!")
```

```
          break
        }
        select {
        case sC := <-statusChannel:
          fmt.Println("Message on StatusChannel", sC)

        }
      }

    }
```

This is a basic extrapolation of our last function, the capitalization function. However, each piece here is responsible for some aspect of reading URLs or appending its respective content to a larger variable.

In the following code, we created a sort of master loop that lets you know when a URL has been grabbed on `statusChannel`:

```
for {
  if applicationStatus == false {
    fmt.Println(fullText)
    fmt.Println("Done!")
    break
  }
  select {
    case sC := <- statusChannel:
      fmt.Println("Message on StatusChannel",sC)

  }
}
```

Often, you'll see this wrapped in `go func()` as part of a `WaitGroup` struct, or not wrapped at all (depending on the type of feedback you require).

The control flow, in this case, is `evaluateStatus`, which works as a channel monitor that lets us know when data crosses each channel and ends execution when it's complete. The `readURLs` function immediately begins reading our URLs, extracting the underlying data and passing it on to `textChannel`. At this point, our `addToScrapedText` function takes each sent HTML file and appends it to the `fullText` variable. When `evaluateStatus` determines that all URLs have been read, it sets `applicationStatus` to `false`. At this point, the infinite loop at the bottom of `main()` quits.

As mentioned, a crawler cannot come more rudimentary than this, but seeing a real-world example of how goroutines can work in congress will set us up for safer and more complex examples in the coming chapters.

Summary

In this chapter, we learned how to go from simple goroutines and instantiating channels to extending the basic functionality of goroutines and allowing cross-channel, bidirectional communication within concurrent processes. We looked at new ways to create blocking code to prevent our main process from ending before our goroutines. Finally, we learned about using select statements to develop reactive channels that are silent unless data is sent along a channel.

In our rudimentary web spider example, we employed these concepts together to create a safe, lightweight process that could extract all links from an array of URLs, grab the content via HTTP, and store the resulting response.

In the next chapter, we'll go beneath the surface to see how Go's internal scheduling manages concurrency and start using channels to really utilize the power, thrift, and speed of concurrency in Go.

2 Understanding the Concurrency Model

Now that we have a sense of what Go is capable of and how to test drive some concurrency models, we need to look deeper into Go's most powerful features to understand how to best utilize various concurrent tools and models.

We played with some general and basic goroutines to see how we can run concurrent processes, but we need to see how Go manages scheduling in concurrency before we get to communication between channels.

Understanding the working of goroutines

By this point, you should be well-versed in what goroutines do, but it's worth understanding *how* they work internally in Go. Go handles concurrency with cooperative scheduling, which, as we mentioned in the previous chapter, is heavily dependent on some form of blocking code.

The most common alternative to cooperative scheduling is preemptive scheduling, wherein each subprocess is granted a space of time to complete and then its execution is paused for the next.

Without some form of yielding back to the main thread, execution runs into issues. This is because Go works with a single process, working as a conductor for an orchestra of goroutines. Each subprocess is responsible for announcing its own completion. As compared to other concurrency models, some of which allow for direct, named communication, this might pose a sticking point, particularly if you haven't worked with channels before.

You can probably see a potential for deadlocks given these facts. In this chapter, we'll discuss both the ways Go's design allows us to manage this and the methods to mitigate issues in applications wherein it fails.

Synchronous versus asynchronous goroutines

Understanding the concurrency model is sometimes an early pain point for programmers—not just for Go, but across languages that use different models as well. Part of this is due to operating in a *black box* (depending on your terminal preferences); a developer has to rely on logging or errors with data consistency to discern asynchronous and/or multiple core timing issues.

As the concepts of synchronous and asynchronous or concurrent and nonconcurrent tasks can sometimes be a bit abstract, we will have a bit of fun here in an effort to demonstrate all the concepts we've covered so far in a visual way.

There are, of course, a myriad of ways to address feedback and logging. You can write to files in `console/terminal/stdout…`, most of which are inherently linear in nature. There is no concise way to represent concurrency in a logfile. Given this and the fact that we are dealing with an emerging language with a focus on servers, let's take a different angle.

Instead of simply outputting to a file, we'll create a visual feedback that shows when a process starts and stops on a timeline.

Designing the web server plan

To show how approaches differ, we'll create a simple web server that loops through three trivial tasks and outputs their execution marks on an X-second timeline. We'll do this using a third-party library called `svgo` and the built-in `http` package for Go.

To start, let's grab the `svgo` library via `go get`:

```
go get github.com/ajstarks/svgo
```

If you try to install a package via the `go get` command and get an error about `$GOPATH` not being set, you need to set that environment variable. `GOPATH` is where Go will look to find installed import packages.

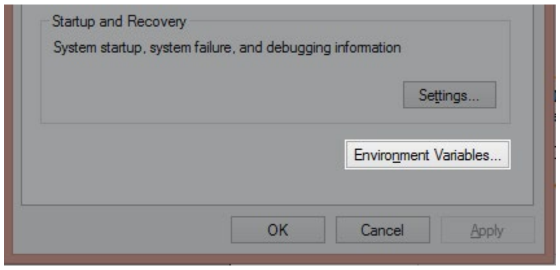
To set this in Linux (or Mac), type the following in bash (or Terminal):

```
export GOPATH=/usr/yourpathhere
```

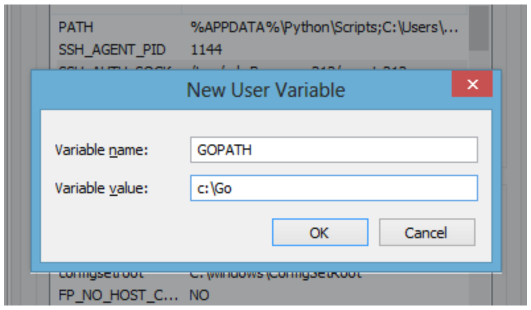
This path is up to you, so pick a place where you're most comfortable storing your Go packages.

To ensure it's globally accessible, install it where your Go binary is installed.

On Windows, you can right-click on **My Computer** and navigate to **Properties** | **Advanced system settings** | **Environment Variables...**, as shown in the following screenshot:



Here, you'll need to create a new variable called `GOPATH`. As with the Linux and Mac instructions, this can either be your Go language root directory or someplace else entirely. In this example, we've used `C:\Go`, as shown in the following screenshot:



PATH %APPDATA%\Python\Scripts;C:\Users\...
SSH_AGENT_PID 1144

New User Variable

Variable name: GOPATH

Variable value: c:\Go

FP_NO_HOST_C... NO

Note that after taking these steps, you may need to reopen the Terminal, Command Prompt, or bash sessions before the value is read as valid. On *nix systems, you can log in and log out to initiate this.

Now that we have installed gosvg, we can visually demonstrate how the asynchronous and synchronous processes will look side-by-side as well as with multiple processors.

More libraries

Why SVG? We didn't need to use SVG and a web server, of course, and if you'd rather see an image generated and open that separately, there are other alternatives to do so. There are some additional graphical libraries available for Go, which are as follows:

- **draw2d**: As the name suggests, this is a two-dimensional drawing library for doing vector-style and raster graphics, which can be found at `https://code.google.com/p/draw2d/`.
- **graphics-go**: This project involves some members of the Go team itself. It's fairly limited in scope. You can find more about it at `https://code.google.com/p/graphics-go/`.
- **go:ngine**: This is one of the few OpenGL implementations for Go. It can be overkill for this project, but if you find yourself in need of a three-dimensional graphics library, start at `http://go-ngine.com/`.
- **Go-SDL**: Another possible overkill method, this is an implementation of the wonderful multimedia library SDL. You can find more about it at `https://github.com/banthar/Go-SDL`.

Robust GUI toolkits are also available, but as they were designed as systems languages, it isn't really Go's forte.

Visualizing concurrency

Our first attempt at visualizing concurrency will have two simple goroutines running the `drawPoint` function in a loop with 100 iterations. After running this, you can visit `localhost:1900/visualize` and see what concurrent goroutines look like.

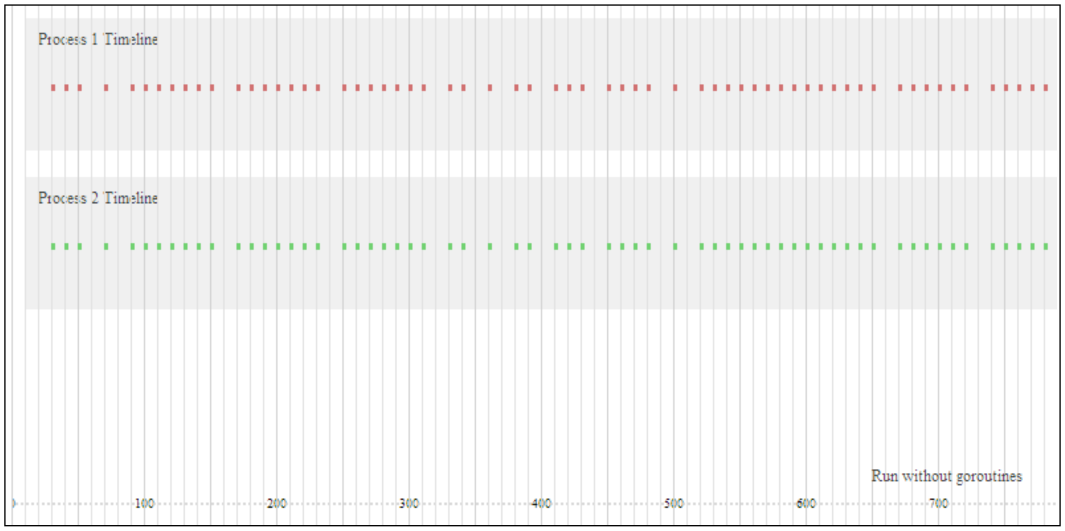
If you run into problems with port 1900 (either with your firewall or through a port conflict), feel free to change the value on line 99 in the `main()` function. You may also need to access it through `127.0.0.1` if your system doesn't resolve localhost.

Note that we're not using `WaitGroup` or anything to manage the end of the goroutines because all we want to see is a visual representation of our code running. You can also handle this with a specific blocking code or `runtime.Gosched()`, as shown:

```
package main

import (
    "github.com/ajstarks/svgo"
    "net/http"
    "fmt"
    "log"
    "time"
    "strconv"
)

var width = 800
var height = 400
var startTime = time.Now().UnixNano()


func drawPoint(osvg *svg.SVG, pnt int, process int) {
  sec := time.Now().UnixNano()
  diff := ( int64(sec) - int64(startTime) ) / 100000

  pointLocation := 0

  pointLocation = int(diff)
  pointLocationV := 0
  color := "#000000"
  switch {
    case process == 1:
      pointLocationV = 60
      color = "#cc6666"
    default:
      pointLocationV = 180
      color = "#66cc66"

  }
```

```
    osvg.Rect(pointLocation,pointLocationV,3,5,"fill:"+color+";stroke:
    none;")
    time.Sleep(150 * time.Millisecond)
  }

  func visualize(rw http.ResponseWriter, req *http.Request) {
    startTime = time.Now().UnixNano()
    fmt.Println("Request to /visualize")
    rw.Header().Set("Content-Type", "image/svg+xml")

    outputSVG := svg.New(rw)

    outputSVG.Start(width, height)
    outputSVG.Rect(10, 10, 780, 100, "fill:#eeeeee;stroke:none")
    outputSVG.Text(20, 30, "Process 1 Timeline", "text-
      anchor:start;font-size:12px;fill:#333333")
    outputSVG.Rect(10, 130, 780, 100, "fill:#eeeeee;stroke:none")
    outputSVG.Text(20, 150, "Process 2 Timeline", "text-
      anchor:start;font-size:12px;fill:#333333")

    for i:= 0; i < 801; i++ {
      timeText := strconv.FormatInt(int64(i),10)
      if i % 100 == 0 {
        outputSVG.Text(i,380,timeText,"text-anchor:middle;font-
          size:10px;fill:#000000")
      }else if i % 4 == 0 {
        outputSVG.Circle(i,377,1,"fill:#cccccc;stroke:none")
      }


      if i % 10 == 0 {
        outputSVG.Rect(i,0,1,400,"fill:#dddddd")
      }
      if i % 50 == 0 {
        outputSVG.Rect(i,0,1,400,"fill:#cccccc")
      }



    }

    for i := 0; i < 100; i++ {
      go drawPoint(outputSVG,i,1)
      drawPoint(outputSVG,i,2)
    }
    outputSVG.Text(650, 360, "Run without goroutines", "text-
      anchor:start;font-size:12px;fill:#333333")
    outputSVG.End()
}

func main() {
    http.Handle("/visualize", http.HandlerFunc(visualize))

        err := http.ListenAndServe(":1900", nil)
        if err != nil {
            log.Fatal("ListenAndServe:", err)
        }

}
```

When you go to `localhost:1900/visualize`, you should see something like the following screenshot:



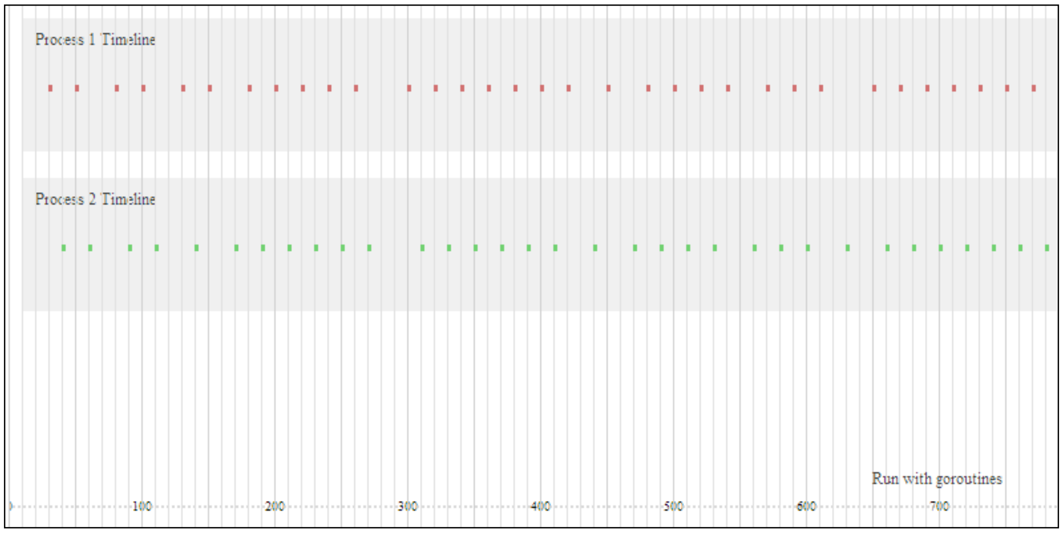
As you can see, everything is definitely running concurrently—our briefly sleeping goroutines hit on the timeline at the same moment. By simply forcing the goroutines to run in a serial fashion, you'll see a predictable change in this behavior. Remove the goroutine call on line 73, as shown:

```
        drawPoint(outputSVG,i,1)
        drawPoint(outputSVG,i,2)
```

To keep our demonstration clean, change line 77 to indicate that there are no goroutines as follows:

```
outputSVG.Text(650, 360, "Run with goroutines", "text-
  anchor:start;font-size:12px;fill:#333333")
```

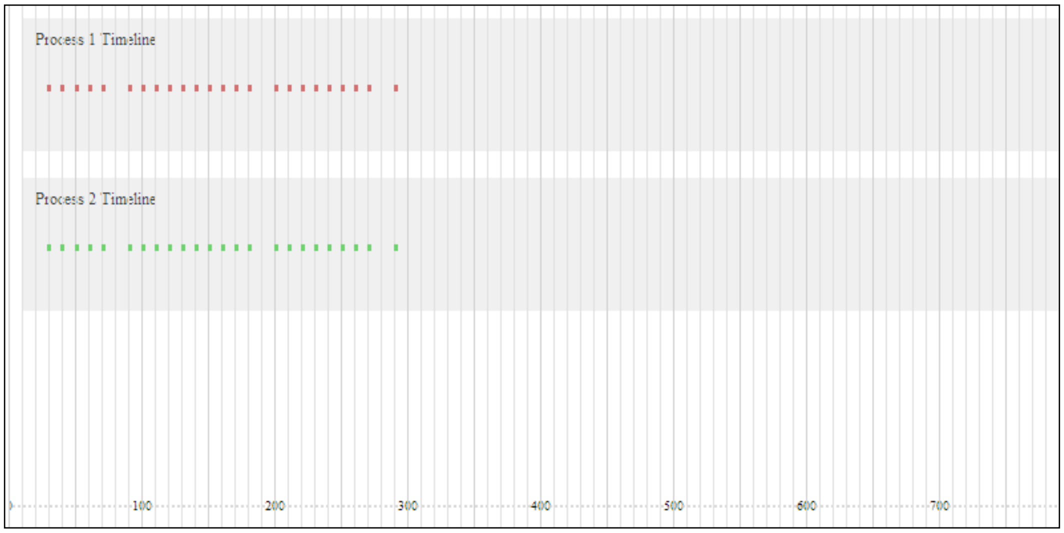
If we stop our server and restart with `go run`, we should see something like the following screenshot:



Now, each process waits for the previous process to complete before beginning. You can actually add this sort of feedback to any application if you run into problems with syncing data, channels, and processes.

If we so desired, we could add some channels and show communication across them as represented. Later, we will design a self-diagnosing server that gives real-time analytics about the state and status of the server, requests, and channels.

If we turn the goroutine back on and increase our maximum available processors, we'll see something similar to the following screenshot, which is not exactly the same as our first screenshot:



Your mileage will obviously vary depending on server speeds, the number of processors, and so on. But in this case, our change here resulted in a faster total execution time for our two processes with intermittent sleeps. This should come as no surprise, given we have essentially twice the bandwidth available to complete the two tasks.

RSS in action

Let's take the concept of **Rich Site Summary** / **Really Simple Syndication** (**RSS**) and inject some real potential delays to identify where we can best utilize goroutines in an effort to speed up execution and prevent blocking code. One common way to bring real-life, potentially blocking application elements into your code is to use something involving network transmission.

This is also a great place to look at timeouts and close channels to ensure that our program doesn't fall apart if something takes too long.

To accomplish both these requirements, we'll build a very basic RSS reader that will simply parse through and grab the contents of five RSS feeds. We'll read each of these as well as the provided links on each, and then we'll generate an SVG report of the process available via HTTP.

This is obviously an application best suited for a background task—you'll notice that each request can take a long time. However, for graphically representing a real-life process working with and without concurrency, it will work, especially with a single end user. We'll also log our steps to standard output, so be sure to take a look at your console as well.

For this example, we'll again use a third-party library, although it's entirely possible to parse RSS using Go's built-in XML package. Given the open-ended nature of XML and the specificity of RSS, we'll bypass them and use `go-pkg-rss` by Jim Teeuwen, available via the following `go get` command:

```
go get github.com/jteeuwen/go-pkg-rss
```

While this package is specifically intended as a replacement for the Google Reader product, which means that it does interval-based polling for new content within a set collection of sources, it also has a fairly neat and tidy RSS reading implementation. There are a few other RSS parsing libraries out there, though, so feel free to experiment.

An RSS reader with self diagnostics

Let's take a look at what we've learned so far, and use it to fetch and parse a set of RSS feeds concurrently while returning some visual feedback about the process in an internal web browser, as shown in the following code:

```
package main

import(
  "github.com/ajstarks/svgo"
  rss "github.com/jteeuwen/go-pkg-rss"
  "net/http"
  "log"
  "fmt"
  "strconv"
  "time"
  "os"
  "sync"
  "runtime"
)

type Feed struct {
  url string
  status int
  itemCount int
```

```
    complete bool
    itemsComplete bool
    index int
  }
```

Here is the basis of our feed's overall structure: we have a `url` variable that represents the feed's location, a `status` variable to indicate whether it's started, and a `complete` Boolean variable to indicate it's finished. The next piece is an individual `FeedItem`; here's how it can be laid out:

```
  type FeedItem struct {
    feedIndex int
    complete bool
    url string
  }
```

Meanwhile, we will not do much with individual items; at this point, we simply maintain a URL, whether it's complete or a `FeedItem` struct's index.

```
  var feeds []Feed
  var height int
  var width int
  var colors []string
  var startTime int64
  var timeout int
  var feedSpace int

  var wg sync.WaitGroup

  func grabFeed(feed *Feed, feedChan chan bool, osvg *svg.SVG) {


    startGrab := time.Now().Unix()
    startGrabSeconds := startGrab - startTime

    fmt.Println("Grabbing feed",feed.url,"
      at",startGrabSeconds,"second mark")

    if feed.status == 0 {
      fmt.Println("Feed not yet read")
      feed.status = 1

      startX := int(startGrabSeconds * 33);
      startY := feedSpace * (feed.index)

      fmt.Println(startY)
      wg.Add(1)
```

```
        rssFeed := rss.New(timeout, true, channelHandler,
          itemsHandler);

        if err := rssFeed.Fetch(feed.url, nil); err != nil {
          fmt.Fprintf(os.Stderr, "[e] %s: %s", feed.url, err)
          return
        } else {



          endSec := time.Now().Unix()
          endX := int( (endSec - startGrab) )
          if endX == 0 {
            endX = 1
          }
          fmt.Println("Read feed in",endX,"seconds")
          osvg.Rect(startX,startY,endX,feedSpace,"fill:
            #000000;opacity:.4")
          wg.Wait()

          endGrab := time.Now().Unix()
          endGrabSeconds := endGrab - startTime
          feedEndX := int(endGrabSeconds * 33);

          osvg.Rect(feedEndX,startY,1,feedSpace,"fill:#ff0000;opacity:.9")

          feedChan <- true
        }


      }else if feed.status == 1{
        fmt.Println("Feed already in progress")
      }



    }
```

The `grabFeed()` method directly controls the flow of grabbing any individual feed. It also bypasses potential concurrent duplication through the `WaitGroup` struct. Next, let's check out the `itemsHandler` function:

```
func channelHandler(feed *rss.Feed, newchannels []*rss.Channel) {

}


func itemsHandler(feed *rss.Feed, ch *rss.Channel, newitems
  []*rss.Item) {
    fmt.Println("Found",len(newitems),"items in",feed.Url)


    for i := range newitems {
      url := *newitems[i].Guid
      fmt.Println(url)

    }

    wg.Done()
  }
```

The `itemsHandler` function doesn't do much at this point, other than instantiating a new `FeedItem` struct—in the real world, we'd take this as the next step and retrieve the values of the items themselves. Our next step is to look at the process that grabs individual feeds and marks the time taken for each one, as follows:

```
  func getRSS(rw http.ResponseWriter, req *http.Request) {
    startTime = time.Now().Unix()
    rw.Header().Set("Content-Type", "image/svg+xml")
    outputSVG := svg.New(rw)
    outputSVG.Start(width, height)

    feedSpace = (height-20) / len(feeds)

    for i:= 0; i < 30000; i++ {
      timeText := strconv.FormatInt(int64(i/10),10)
      if i % 1000 == 0 {
        outputSVG.Text(i/30,390,timeText,"text-anchor:middle;font-
          size:10px;fill:#000000")
      }else if i % 4 == 0 {
        outputSVG.Circle(i,377,1,"fill:#cccccc;stroke:none")
      }


      if i % 10 == 0 {
        outputSVG.Rect(i,0,1,400,"fill:#dddddd")
      }
      if i % 50 == 0 {
        outputSVG.Rect(i,0,1,400,"fill:#cccccc")
      }


    }
    feedChan := make(chan bool, 3)

    for i := range feeds {

      outputSVG.Rect(0, (i*feedSpace), width, feedSpace,
        "fill:"+colors[i]+";stroke:none;")
      feeds[i].status = 0
      go grabFeed(&feeds[i], feedChan, outputSVG)
      <- feedChan
    }

    outputSVG.End()
  }
```

Here, we retrieve the RSS feed and mark points on our SVG with the status of our retrieval and read events. Our `main()` function will primarily handle the setup of feeds, as follows:

```
func main() {

  runtime.GOMAXPROCS(2)

  timeout = 1000

  width = 1000
  height = 400

  feeds = append(feeds, Feed{index: 0, url:
    "https://groups.google.com/forum/feed/golang-
    nuts/msgs/rss_v2_0.xml?num=50", status: 0, itemCount: 0,
    complete: false, itemsComplete: false})
  feeds = append(feeds, Feed{index: 1, url:
    "http://www.reddit.com/r/golang/.rss", status: 0, itemCount:
    0, complete: false, itemsComplete: false})
  feeds = append(feeds, Feed{index: 2, url:
    "https://groups.google.com/forum/feed/golang-
    dev/msgs/rss_v2_0.xml?num=50", status: 0, itemCount: 0,
    complete: false, itemsComplete: false })
```

Here is our slice of `FeedItem` structs:

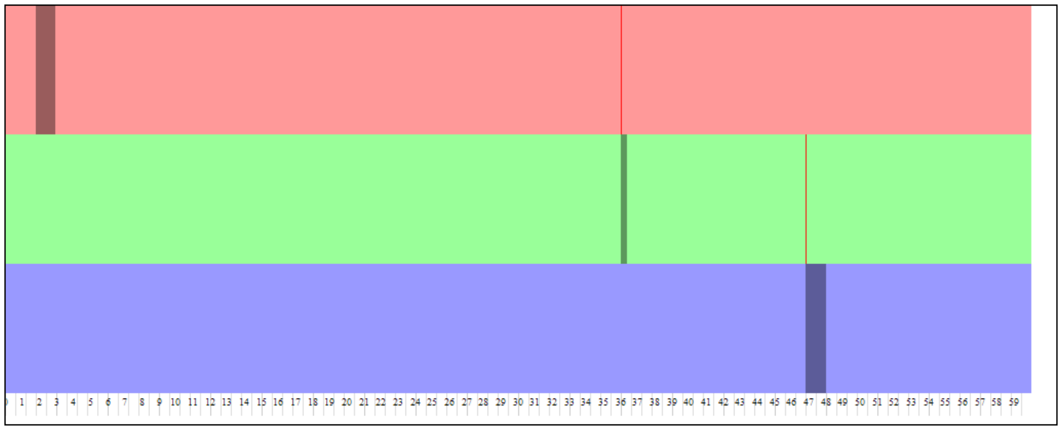
```
  colors = append(colors,"#ff9999")
  colors = append(colors,"#99ff99")
  colors = append(colors,"#9999ff")
```

In the print version, these colors may not be particularly useful, but testing it on your system will allow you to delineate between events inside the application. We'll need an HTTP route to act as an endpoint; here's how we'll set that up:

```
     http.Handle("/getrss", http.HandlerFunc(getRSS))
       err := http.ListenAndServe(":1900", nil)
       if err != nil {
           log.Fatal("ListenAndServe:", err)
       }
   }
```

When run, you should see the start and duration of the RSS feed retrieval and parsing, followed by a thin line indicating that the feed has been parsed and all items read.

Each of the three blocks expresses the full time to process each feed, demonstrating the nonconcurrent execution of this version, as shown in the following screenshot:



Note that we don't do anything interesting with the feed items, we simply read the URL. The next step will be to grab the items via HTTP, as shown in the following code snippet:

```
     url := *newitems[i].Guid
         response, _, err := http.Get(url)
         if err != nil {

         }
```

With this example, we stop at every step to provide some sort of feedback to the SVG that some event has occurred. Our channel here is buffered and we explicitly state that it must receive three Boolean messages before it can finish blocking, as shown in the following code snippet:

```
feedChan := make(chan bool, 3)

for i := range feeds {

  outputSVG.Rect(0, (i*feedSpace), width, feedSpace,
    "fill:"+colors[i]+";stroke:none;")
  feeds[i].status = 0
  go grabFeed(&feeds[i], feedChan, outputSVG)
  <- feedChan
}

outputSVG.End()
```

By giving `3` as the second parameter in our channel invocation, we tell Go that this channel must receive three responses before continuing the application. You should use caution with this, though, particularly in setting things explicitly as we have done here. What if one of the goroutines never sent a Boolean across the channel? The application would crash.

Note that we also increased our timeline here, from 800ms to 60 seconds, to allow for retrieval of all feeds. Keep in mind that if our script exceeds 60 seconds, all actions beyond that time will occur outside of this visual timeline representation.

By implementing the `WaitGroup` struct while reading feeds, we impose some serialization and synchronization to the application. The second feed will not start until the first feed has completed retrieving all URLs. You can probably see where this might introduce some errors going forward:

```
wg.Add(1)
rssFeed := rss.New(timeout, true, channelHandler,
  itemsHandler);
…
wg.Wait()
```

This tells our application to yield until we set the `Done()` command from the `itemsHandler()` function.

So what happens if we remove `WaitGroups` entirely? Given that the calls to grab the feed items are asynchronous, we may not see the status of all of our RSS calls; instead, we might see just one or two feeds or no feed at all.

Imposing a timeout

So what happens if nothing runs within our timeline? As you might expect, we'll get three bars with no activity in them. It's important to consider how to kill processes that aren't doing what we expect them to. In this case, the best method is a timeout. The `Get` method in the `http` package does not natively support a timeout, so you'll have to roll your own `rssFeed.Fetch` (and underlying `http.Get()`) implementation if you want to prevent these requests from going into perpetuity and killing your application. We'll dig into this a bit later; in the mean time, take a look at the `Transport` struct, available in the core `http` package at `http://golang.org/pkg/net/http/#Transport`.

A little bit about CSP

We touched on CSP briefly in the previous chapter, but it's worth exploring a bit more in the context of how Go's concurrency model operates.

CSP evolved in the late 1970s and early 1980s through the work of Sir Tony Hoare and is still in the midst of evolution today. Go's implementation is heavily based on CSP, but it neither entirely follows all the rules and conventions set forth in its initial description nor does it follow its evolution since.

One of the ways in which Go differs from true CSP is that as it is defined, a process in Go will only continue so long as there exists a channel ready to receive from that process. We've already encountered a couple of deadlocks that were the result of a listening channel with nothing to receive. The inverse is also true; a deadlock can result from a channel continuing without sending anything, leaving its receiving channel hanging indefinitely.

This behavior is endemic to Go's scheduler, and it should really only pose problems when you're working with channels initially.

Hoare's original work is now available (mostly) free from a number of institutions. You can read, cite, copy, and redistribute it free of charge (but not for commercial gain). If you want to read the whole thing, you can grab it at `http://www.cs.ucf.edu/courses/cop4020/sum2009/CSP-hoare.pdf`.

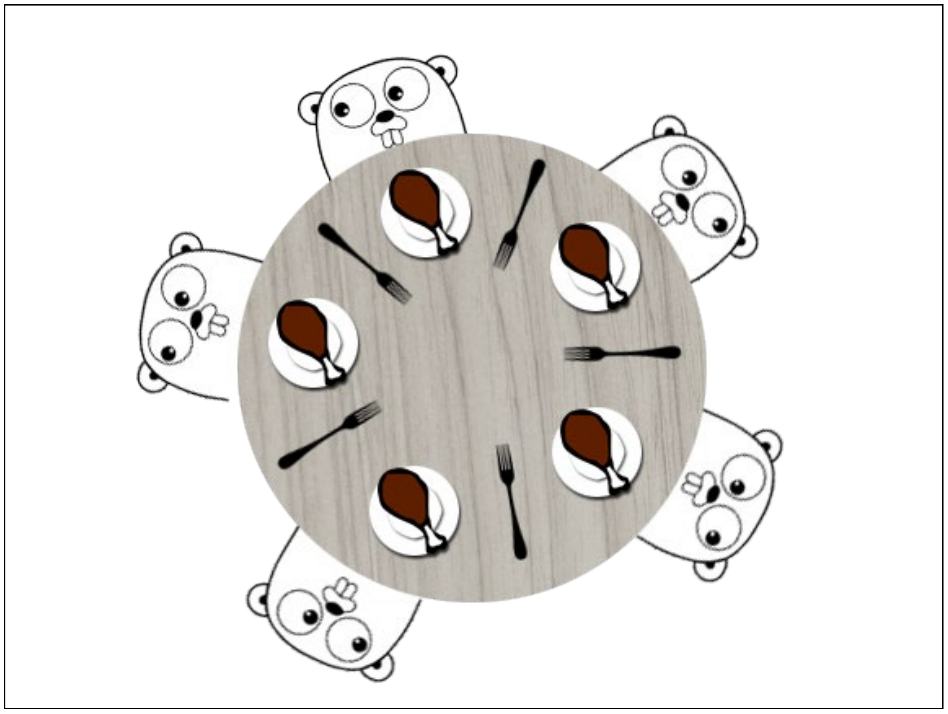
The complete book itself is also available at `http://www.usingcsp.com/cspbook.pdf`.

As of this publishing, Hoare is working as a researcher at Microsoft.

As per the designers of the application itself, the goal of Go's implementation of CSP concepts was to focus on simplicity—you don't have to worry about threads or mutexes unless you really want to or need to.

The dining philosophers problem

You may have heard of the dining philosophers problem, which describes the kind of problems concurrent programming was designed to solve. The dining philosophers problem was formulated by the great Edsger Dijkstra. The crux of the problem is a matter of resources—five philosophers sit at a table with five plates of food and five forks, and each can only eat when he has two forks (one to his left and another to his right). A visual representation is shown as follows:



With a single fork on either side, any given philosopher can only eat when he has a fork in both hands and must put both back on the table when complete. The idea is to coordinate the meal such that all of the philosophers can eat in perpetuity without any starving—two philosophers must be able to eat at any moment and there can be no deadlocks. They're philosophers because when they're not eating, they're thinking. In a programming analog, you can consider this as either a waiting channel or a sleeping process.

Go handles this problem pretty succinctly with goroutines. Given five philosophers (in an individual struct, for example), you can have all five alternate between thinking, receiving a notification when the forks are down, grabbing forks, dining with forks, and placing the forks down.

Receiving the notification that the forks are down acts as the listening channel, dining and thinking are separate processes, and placing the forks down operates as an announcement along the channel.

We can visualize this concept in the following pseudo Go code:

```
type Philosopher struct {
  leftHand bool
  rightHand bool
  status int
  name string
}

func main() {

  philosophers := [...]Philospher{"Kant", "Turing",
    "Descartes","Kierkegaard","Wittgenstein"}

  evaluate := func() {
    for {

      select {
        case <- forkUp:
          // philosophers think!
        case <- forkDown:
          // next philospher eats in round robin
      }

    }

  }

}
```

This example has been left very abstract and nonoperational so that you have a chance to attempt to solve it. We will build a functional solution for this in the next chapter, so make sure to compare your solution later on.

There are hundreds of ways to handle this problem, and we'll look at a couple of alternatives and how they can or cannot play nicely within Go itself.

Go and the actor model

The actor model is something that you'll likely be very familiar with if you're an Erlang or Scala user. The difference between CSP and the actor model is negligible but important. With CSP, messages from one channel can only be completely sent if another channel is listening and ready for them. The actor model does not necessarily require a ready channel for another to send. In fact, it stresses direct communication rather than relying on the conduit of a channel.

Both systems can be nondeterministic, which we've already seen demonstrated in Go/CSP in our earlier examples. CSP and goroutines are anonymous and transmission is specified by the channel rather than the source and destination. An easy way to visualize this in pseudocode in the actor model is as follows:

```
a = new Actor
b = new Actor
a -> b("message")
```

In CSP, it is as follows:

```
a = new Actor
b = new Actor
c = new Channel
a -> c("sending something")
b <- c("receiving something")
```

Both serve the same fundamental functionality but through slightly different ways.

Object orientation

As you work with Go, you will notice that there is a core characteristic that's often espoused, which users may feel is wrong. You'll hear that Go is not an object-oriented language, and yet you have structs that can have methods, those methods in turn can have methods, and you can have communication to and from any instantiation of it. Channels themselves may feel like primitive object interfaces, capable of setting and receiving values from a given data element.

The message passing implementation of Go is, indeed, a core concept of object-oriented programming. Structs with interfaces operate essentially as classes, and Go supports polymorphism (although not parametric polymorphism). Yet, many who work with the language (and who have designed it) stress that it is not object oriented. So what gives?

Much of this definition ultimately depends on who you ask. Some believe that Go lacks some of the requisite characteristics of object-oriented programming, and others believe it satisfies them. The most important thing to keep in mind is that you're not limited by Go's design. Anything that you can do in a *true* object-oriented language can be handled without much struggle within Go.

Demonstrating simple polymorphism in Go

As mentioned before, if you expect polymorphism to resemble object-oriented programming, this may not represent a syntactical analogue. However, the use of interfaces as an abstraction of class-bound polymorphic methods is just as clean, and in many ways, more explicit and readable. Let's look at a very simple implementation of polymorphism in Go:

```
type intInterface struct {

}

type stringInterface struct {

}

func (number intInterface) Add (a int, b int) int {
  return a + b;
}

func (text stringInterface) Add (a string, b string) string {
  return a + b
}

func main() {

  number := new (intInterface)
    fmt.Println( number.Add(1,2) )

  text := new (stringInterface)
    fmt.Println( text.Add("this old man"," he played one"))

}
```

As you can see, we use an interface (or its Go analog) to disambiguate methods. You cannot have generics the same way you might in Java, for example. This, however, boils down to a mere matter of style in the end. You should neither find this daunting nor will it impose any cruft or ambiguity into your code.

Using concurrency

It hasn't yet been mentioned, but we should be aware that concurrency is not always necessary and beneficial for an application. There exists no real rule of thumb, and it's rare that concurrency will introduce problems to an application; but if you really think about applications as a whole, not all will require concurrent processes.

So what works best? As we've seen in the previous example, anything that introduces potential latency or I/O blocking, such as network calls, disk reads, third-party applications (primarily databases), and distributed systems, can benefit from concurrency. If you have the ability to do work while other work is being done on an undetermined timeline, concurrency strategies can improve the speed and reliability of an application.

The lesson here is you should never feel compelled to shoehorn concurrency into an application that doesn't really require it. Programs with inter-process dependencies (or lack of blocking and external dependencies) may see little or no benefit from implementing concurrency structures.

Managing threads

So far, you've probably noticed that thread management is not a matter that requires the programmer's utmost concern in Go. This is by design. Goroutines aren't tied to a specific thread or threads that are handled by Go's internal scheduler. However, this doesn't mean that you neither have access to the threads nor the ability to control what individual threads do. As you know, you can already tell Go how many threads you have (or wish to use) by using `GOMAXPROCS`. We also know that using this can introduce asynchronous issues as it pertains to data consistency and execution order.

At this point, the main issue with threads is not how they're accessed or utilized, but how to properly control execution flow to guarantee that your data is predictable and synchronized.

Using sync and mutexes to lock data

One issue that you may have run into with the preceding examples is the notion of atomic data. After all, if you deal with variables and structures across multiple goroutines, and possibly processors, how do you ensure that your data is safe across them? If these processes run in parallel, coordinating data access can sometimes be problematic.

Go provides a bevy of tools in its `sync` package to handle these types of problems. How elegantly you approach them depends heavily on your approach, but you should never have to reinvent the wheel in this realm.

We've already looked at the `WaitGroup` struct, which provides a simple method to tell the main thread to pause until the next notification that says a waiting process has done what it's supposed to do.

Go also provides a direct abstraction to a mutex. It may seem contradictory to call something a direct abstraction, but the truth is you don't have access to Go's scheduler, only an approximation of a true mutex.

We can use a mutex to lock and unlock data and guarantee atomicity in our data. In many cases, this may not be necessary; there are a great many times where the order of execution does not impact the consistency of the underlying data. However, when we do have concerns about this value, it's helpful to be able to invoke a lock explicitly. Let's take the following example:

```
package main

import(
  "fmt"
  "sync"
)

func main() {
  current := 0
  iterations := 100
  wg := new (sync.WaitGroup);

  for i := 0; i < iterations; i++ {
    wg.Add(1)
    go func() {
      current++
      fmt.Println(current)
      wg.Done()
    }()
    wg.Wait()
  }


}
```

Unsurprisingly, this provides a list of 0 to 99 in your terminal. What happens if we change `WaitGroup` to know there will be 100 instances of `Done()` called, and put our blocking code at the end of the loop?

To demonstrate a simple proposition of why and how to best utilize `waitGroups` as a mechanism for concurrency control, let's do a simple number iterator and look at the results. We will also check out how a directly called mutex can augment this functionality, as follows:

```
func main() {
  runtime.GOMAXPROCS(2)
  current := 0
  iterations := 100
  wg := new (sync.WaitGroup);
  wg.Add(iterations)
  for i := 0; i < iterations; i++ {
    go func() {
      current++
      fmt.Println(current)
      wg.Done()
    }()

  }
  wg.Wait()

}
```

Now, our order of execution is suddenly off. You may see something like the following output:

```
95
96
98
```

```
99
100
3
4
```

We have the ability to lock and unlock the current command at will; however, this won't change the underlying execution order, it will only prevent reading and/or writing to and from a variable until an unlock is called.

Let's try to lock down the variable we're outputting using `mutex`, as follows:

```
for i := 0; i < iterations; i++ {
  go func() {
    mutex.Lock()
    fmt.Println(current)
    current++
    mutex.Unlock()
    fmt.Println(current)
    wg.Done()
  }()

}
```

You can probably see how a mutex control mechanism can be important to enforce data integrity in your concurrent application. We'll look more at mutexes and locking and unlocking processes in *Chapter 4, Data Integrity in an Application*.

Summary

In this chapter, we've tried to remove some of the ambiguity of Go's concurrency patterns and models by giving visual, real-time feedback to a few applications, including a rudimentary RSS aggregator and reader. We examined the dining philosophers problem and looked at ways you can use the Go concurrency topics to solve the problem neatly and succinctly. We compared the way CSP and actor models are similar and ways in which they differ.

In the next chapter, we will take these concepts and apply them to the process of developing a strategy to maintain concurrency in an application.

3 Developing a Concurrent Strategy

In the previous chapter, we looked at the concurrency model that Go relies on to make your life as a developer easier. We also saw a visual representation of parallelism and concurrency. These help us to understand the differences and overlaps between serialized, concurrent, and parallel applications.

However, the most critical part of any concurrent application is not the concurrency itself but communication and coordination between the concurrent processes.

In this chapter, we'll look at creating a plan for an application that heavily factors communication between processes and how a lack of coordination can lead to significant issues with consistency. We'll look at ways we can visualize our concurrent strategy on paper so that we're better equipped to anticipate potential problems.

Applying efficiency in complex concurrency

When designing applications, we often eschew complex patterns for simplicity, with the assumption that simple systems are often the fastest and most efficient. It seems only logical that a machine with fewer moving parts will be more efficient than one with more.

The paradox here, as it applies to concurrency, is that adding redundancy and significantly more movable parts often leads to a more efficient application. If we consider concurrent schemes, such as goroutines, to be infinitely scalable resources, employing more should always result in some form of efficiency benefit. This applies not just to parallel concurrency but to single core concurrency as well.

If you find yourself designing an application that utilizes concurrency at the cost of efficiency, speed, and consistency, you should ask yourself whether the application truly needs concurrency at all.

When we talk about efficiency, we aren't just dealing with speed. Efficiency should also weigh the CPU and memory overhead and the cost to ensure data consistency.

For example, should an application marginally benefit from concurrency but require an elaborate and/or computationally expensive process to guarantee data consistency, it's worth re-evaluating the strategy entirely.

Keeping your data reliable and up to date should be paramount; while having unreliable data may not always have a devastating effect, it will certainly compromise the reliability of your application.

Identifying race conditions with race detection

If you've ever written an application that depends on the exact timing and sequencing of functions or methods to create a desired output, you're already quite familiar with race conditions.

These are particularly common anytime you deal with concurrency and far more so when parallelism is introduced. We've actually encountered a few of them in the first few chapters, specifically with our incrementing number function.

The most commonly used educational example of race conditions is that of a bank account. Assume that you start with $1,000 and attempt 200 $5 transactions. Each transaction requires a query on the current balance of the account. If it passes, the transaction is approved and $5 is removed from the balance. If it fails, the transaction is declined and the balance remains unchanged.

This is all well and good until the query happens at some point during a concurrent transaction (in most cases in another thread). If, for example, a thread asks "Do you have $5 in your account?" as another thread is in the process of removing $5 but has not yet completed, you can end up with an approved transaction that should have been declined.

Tracking down the cause of race conditions can be—to say the least—a gigantic headache. With Version 1.1 of Go, Google introduced a race detection tool that can help you locate potential issues.

Let's take a very basic example of a multithreaded application with race conditions and see how Golang can help us debug it. In this example, we'll build a bank account that starts with $1,000 and runs 100 transactions for a random amount between $0 and $25.

Each transaction will be run in its own goroutine, as follows:

```
package main

import(
  "fmt"
  "time"
  "sync"
  "runtime"
  "math/rand"
)

var balance int
var transactionNo int

func main() {
  rand.Seed(time.Now().Unix())
  runtime.GOMAXPROCS(2)
  var wg sync.WaitGroup

  tranChan := make(chan bool)


  balance = 1000
  transactionNo = 0
  fmt.Println("Starting balance: $",balance)

  wg.Add(1)
  for i := 0; i < 100; i++ {
    go func(ii int, trChan chan(bool)) {
      transactionAmount := rand.Intn(25)
      transaction(transactionAmount)
      if (ii == 99) {
        trChan <- true
      }

    }(i,tranChan)
  }
```

```
    go transaction(0)
    select {

      case <- tranChan:
        fmt.Println("Transactions finished")
        wg.Done()

    }

    wg.Wait()
    close(tranChan)
    fmt.Println("Final balance: $",balance)
  }

  func transaction(amt int) (bool) {

    approved := false
    if (balance-amt) < 0 {
      approved = false
    }else {
      approved = true
      balance = balance - amt
    }

    approvedText := "declined"
    if (approved == true) {
      approvedText = "approved"
    }else {

    }
    transactionNo = transactionNo + 1
    fmt.Println(transactionNo,"Transaction for $",amt,approvedText)
    fmt.Println("\tRemaining balance $",balance)
    return approved
  }
```

Depending on your environment (and whether you enable multiple processors), you might have the previous goroutine operate successfully with a $0 or more final balance. You might, on the other hand, simply end up with transactions that exceed the balance at the time of transaction, resulting in a negative balance.

So how do we know for sure?

For most applications and languages, this process often involves a lot of running, rerunning, and logging. It's not unusual for race conditions to present a daunting and laborious debugging process. Google knows this and has given us a race condition detection tool. To test this, simply use the `-race` flag when testing, building, or running your application, as shown:

```
go run -race race-test.go
```

When run on the previous code, Go will execute the application and then report any possible race conditions, as follows:

```
>> Final balance: $0
>> Found 2 data race(s)
```

Here, Go is telling us there are two potential race conditions with data. It isn't telling us that these will surely create data consistency issues, but if you run into such problems, this may give you some clue as to why.

If you look at the top of the output, you'll get more detailed notes on what's causing a race condition. In this example, the details are as follows:

```
==================
WARNING: DATA RACE
Write by goroutine 5: main.transaction()   /var/go/race.go:75 +0xbd
  main.func┬┐001()   /var/go/race.go:31 +0x44

Previous write by goroutine 4: main.transaction()
  /var/go/race.go:75 +0xbd main.func┬┐001()   /var/go/race.go:31
    +0x44

Goroutine 5 (running) created at: main.main()   /var/go/race.go:36
  +0x21c

Goroutine 4 (finished) created at: main.main()   /var/go/race.go:36
  +0x21c
```

We get a detailed, full trace of where our potential race conditions exist. Pretty helpful, huh?

The race detector is guaranteed to not produce false positives, so you can take the results as strong evidence that there is a potential problem in your code. The potential is stressed here because a race condition can go undetected in normal conditions very often—an application may work as expected for days, months, or even years before a race condition can surface.

We've mentioned logging, and if you aren't intimately familiar with Go's core language, your mind might go in a number of directions—stdout, file logs, and so on. So far we've stuck to stdout, but you can use the standard library to handle this logging. Go's log package allows you to write to io or stdout as shown:

```
messageOutput := os.Stdout
logOut := log.New(messageOutput,"Message: ",log.
Ldate|log.Ltime|log.Llongfile);
logOut.Println("This is a message from the
application!")
```

This will produce the following output:

```
Message: 2014/01/21 20:59:11 /var/go/log.go:12: This is
  a message from the application!
```

So, what's the advantage of the log package versus rolling your own? In addition to being standardized, this package is also synchronized in terms of output.

So what now? Well, there are a few options. You can utilize your channels to ensure data integrity with a buffered channel, or you can use the `sync.Mutex` struct to lock your data.

Using mutual exclusions

Typically, mutual exclusion is considered a low-level and best-known approach to synchronicity in your application—you should be able to address data consistency within communication between your channels. However, there will be instances where you need to truly block read/write on a value while you work with it.

At the CPU level, a mutex represents an exchange of binary integer values across registers to acquire and release locks. We'll deal with something on a much higher level, of course.

We're already familiar with the sync package from our use of the `WaitGroup` struct, but the package also contains the conditional variables `struct Cond` and `Once`, which will perform an action just one time, and the mutual exclusion locks `RWMutex` and `Mutex`. As the name `RWMutex` implies, it is open to multiple readers and/or writers to lock and unlock; there is more on this later in this chapter and in *Chapter 5, Locks, Blocks, and Better Channels*.

All of these—as the package name implies—empower you to prevent race conditions on data that may be accessed by any number of goroutines and/or threads. Using any of the methods in this package does not ensure atomicity within data and structures, but it does give you the tools to manage atomicity effectively. Let's look at a few ways we can solidify our account balance in concurrent, threadsafe applications.

As mentioned previously, we can coordinate data changes at the channel level whether that channel is buffered or unbuffered. Let's offload the logic and data manipulation to the channel and see what the `-race` flag presents.

If we modify our main loop, as shown in the following code, to utilize messages received by the channel to manage the balance value, we will avoid race conditions:

```
package main

import(
  "fmt"
  "time"
  "sync"
  "runtime"
  "math/rand"
)

var balance int
var transactionNo int

func main() {
  rand.Seed(time.Now().Unix())
  runtime.GOMAXPROCS(2)
  var wg sync.WaitGroup
  balanceChan := make(chan int)
  tranChan := make(chan bool)


  balance = 1000
  transactionNo = 0
  fmt.Println("Starting balance: $",balance)

  wg.Add(1)
  for i:= 0; i<100; i++ {

    go func(ii int) {

      transactionAmount := rand.Intn(25)
      balanceChan <- transactionAmount
```

```
        if ii == 99 {
          fmt.Println("Should be quittin time")
          tranChan <- true
          close(balanceChan)
          wg.Done()
        }

      }(i)

    }

    go transaction(0)


      breakPoint := false
      for {
        if breakPoint == true {
          break
        }
        select {
          case amt:= <- balanceChan:
            fmt.Println("Transaction for $",amt)
            if (balance - amt) < 0 {
              fmt.Println("Transaction failed!")
            }else {
              balance = balance - amt
              fmt.Println("Transaction succeeded")
            }
            fmt.Println("Balance now $",balance)


          case status := <- tranChan:
            if status == true {
              fmt.Println("Done")
              breakPoint = true
              close(tranChan)

            }
        }
      }

    wg.Wait()

    fmt.Println("Final balance: $",balance)
  }
```

```
func transaction(amt int) (bool) {

  approved := false
  if (balance-amt) < 0 {
    approved = false
  }else {
    approved = true
    balance = balance - amt
  }

  approvedText := "declined"
  if (approved == true) {
    approvedText = "approved"
  }else {

  }
  transactionNo = transactionNo + 1
  fmt.Println(transactionNo,"Transaction for $",amt,approvedText)
  fmt.Println("\tRemaining balance $",balance)
  return approved
}
```

This time, we let the channel manage the data entirely. Let's look at what we're doing:

```
transactionAmount := rand.Intn(25)
balanceChan <- transactionAmount
```

This still generates a random integer between 0 and 25, but instead of passing it to a function, we pass the data along the channel. Channels allow you to control the ownership of data neatly. We then see the select/listener, which largely mirrors our `transaction()` function defined earlier in this chapter:

```
case amt:= <- balanceChan:
fmt.Println("Transaction for $",amt)
if (balance - amt) < 0 {
  fmt.Println("Transaction failed!")
}else {
  balance = balance - amt
  fmt.Println("Transaction succeeded")
}
fmt.Println("Balance now $",balance)
```

To test whether we've averted a race condition, we can run `go run` with the `-race` flag again and see no warnings.

Channels can be seen as the sanctioned go-to way of handling synchronized `dataUse Sync.Mutex()`.

As mentioned, having a built-in race detector is a luxury not afforded to developers in most languages, and having it allows us to test methodologies and get real-time feedback on each.

We noted that using an explicit mutex is discouraged in favor of channels of goroutines. This isn't always exactly true because there is a right time and place for everything, and mutexes are no exclusion. What's worth noting is that mutexes are implemented internally by Go for channels. As was previously mentioned, you can use explicit channels to handle reads and writes and juggle the data between them.

However, this doesn't mean there is no use for explicit locks. An application that has many reads and very few writes might benefit from explicit locks for writes; this doesn't necessarily mean that the reads will be dirty reads, but it could result in faster and/or more concurrent execution.

For the sake of demonstration, let's remove our race condition using an explicit lock. Our `-race` flag tells us where it encounters read/write race conditions, as shown:

```
Read by goroutine 5: main.transaction()   /var/go/race.go:62 +0x46
```

The previous line is just one among several others we'll get from the race detection report. If we look at line 62 in our code, we'll find a reference to `balance`. We'll also find a reference to `transactionNo`, our second race condition. The easiest way to address both is to place a mutual exclusion lock around the contents of the `transaction` function as this is the function that modifies the `balance` and `transactionNo` variables. The `transaction` function is as follows:

```
func transaction(amt int) (bool) {
  mutex.Lock()

  approved := false
  if (balance-amt) < 0 {
    approved = false
  }else {
    approved = true
    balance = balance - amt
  }


  approvedText := "declined"
  if (approved == true) {
    approvedText = "approved"
  }else {
  }
  transactionNo = transactionNo + 1
  fmt.Println(transactionNo,"Transaction for $",amt,approvedText)
  fmt.Println("\tRemaining balance $",balance)

  mutex.Unlock()
  return approved
}
```

We also need to define `mutex` as a global variable at the top of our application, as shown:

```
  var mutex sync.Mutex
```

If we run our application now with the `-race` flag, we get no warnings.

The `mutex` variable is, for practical purposes, an alternative to the `WaitGroup` struct, which functions as a conditional synchronization mechanism. This is also the way that the channels operate—data that moves along channels is contained and isolated between goroutines. A channel can effectively work as a first-in, first-out tool in this way by binding goroutine state to `WaitGroup`; data accessed across the channel can then be provided safety via the lower-level mutex.

Another worthwhile thing to note is the versatility of a channel—we have the ability to share a channel among an array of goroutines to receive and/or send data, and as a first-class citizen, we can pass them along in functions.

Exploring timeouts

Another noteworthy thing we can do with channels is explicitly kill them after a specified amount of time. This is an operation that will be a bit more involved should you decide to manually handle mutual exclusions.

The ability to kill a long-running routine through the channel is extremely helpful; consider a network-dependent operation that should not only be restricted to a short time period but also not allowed to run for a long period. In other words, you want to offer the process a few seconds to complete; but if it runs for more than a minute, our application should know that something has gone wrong enough to stop attempting to listen or send on that channel. The following code demonstrates using a timeout channel in a `select` call:

```
  func main() {

    ourCh := make(chan string,1)
    go func() {

    }()

    select {
      case <-time.After(10 * time.Second):
        fmt.Println("Enough's enough")
        close(ourCh)
    }

  }
```

If we run the previous simple application, we'll see that our goroutine will be allowed to do nothing for exactly 10 seconds, after which we implement a timeout safeguard that bails us out.

You can see this as being particularly useful in network applications; even in the days of blocking and thread-dependent servers, timeouts like these were implemented to prevent a single misbehaving request or process to gum up the entire server. This is the very basis of a classic web server problem that we'll revisit in more detail later.

Importance of consistency

In our example, we'll build an events scheduler. If we are available for a meeting and we get two concurrent requests for a meeting invite, we'll get double-booked should a race condition exist. Alternately, locked data across two goroutines may cause both the requests to be denied or will result in an actual deadlock.

We want to guarantee that any request for availability is consistent—there should neither be double-booking nor should a request for an event be blocked incorrectly (because two concurrent or parallel routines lock the data simultaneously).

Synchronizing our concurrent operations

The word synchronization literally refers to temporal existence—things occurring at the same time. It seems then that the most apt demonstration of synchronicity will be something involving time itself.

When we think about the ways time impacts us, it's generally a matter of scheduling, due dates, and coordination. Going back to our preliminary example from the Preface, if one wishes to plan their grandmother's birthday party, the following types of scheduled tasks can take several forms:

- Things that must be done by a certain time (the actual party)
- Things that cannot be done until another task is completed (putting up decorations before they're purchased)
- Things that can be done in any particular order without impacting the outcome (cleaning the house)
- Things that can be done in any order but may well impact the outcome (buying a cake before finding out what cake your grandmother likes the most)

With these in mind, we'll attempt to handle some rudimentary human scheduling by designing an appointment calendar that can handle any number of people with one hour timeslots between 9 a.m. and 5 p.m.

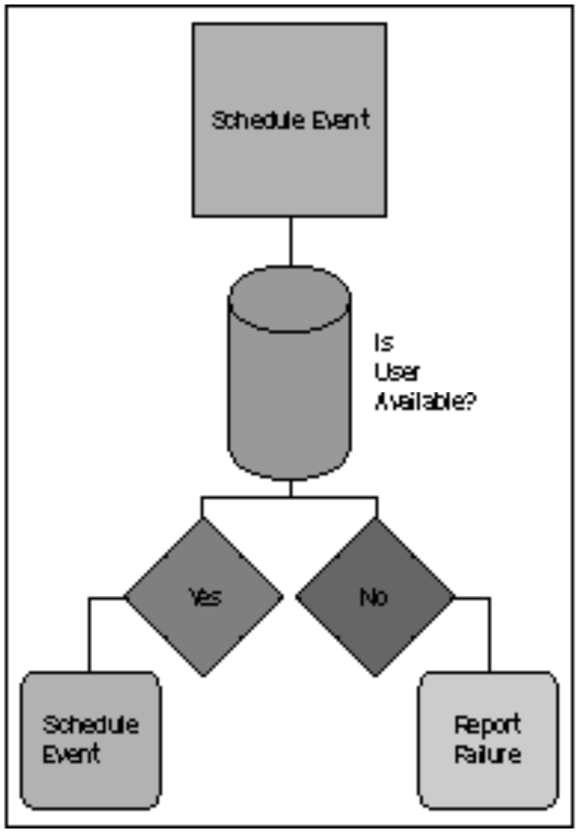
The project – multiuser appointment calendar

What do you do when you decide to write a program?

If you're like a lot of people, you think about the program; perhaps you and a team will write up a spec or requirements document, and then you'll get to coding. Sometimes, there will be a drawing representing some facsimile of the way the application will work.

Quite often, the best way to nail down the architecture and the inner workings of an application is to put pencil to paper and visually represent the way the program will work. For a lot of linear or serial applications, this is often an unnecessary step as things will work in a predictable fashion that should not require any specific coordination within the application logic itself (although coordinating third-party software likely benefits from specification).

You may be familiar with some logic that looks something like the following diagram:



The logic here makes sense. If you remember from our Preface, when humans draw out processes, we tend to serialize them. Visually, going from step one to step two with a finite number of processes is easy to understand.

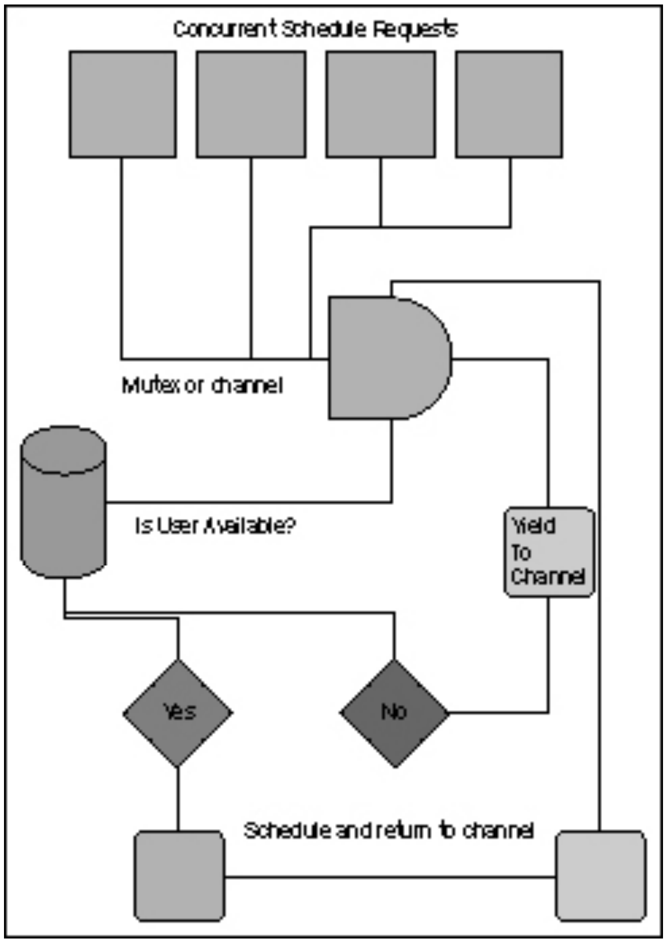
However, when designing a concurrent application, it's essential that we at least account for innumerable and concurrent requests, processes, and logic to make sure our application ends where we want, with the data and results we expect.

In the previous example, we completely ignore the possibility that "Is User Available" could fail or report old or erroneous data. Does it make more sense to address such problems if and when we find them, or should we anticipate them as part of a control flow? Adding complexity to the model can help us reduce the odds of data integrity issues down the road.

Let's visualize this again, taking into account availability pollers that will request availability for a user with any given request for a time/user pair.

Visualizing a concurrent pattern

As we have already discussed, we wish to create a basic blueprint of how our application should function as a starting point. Here, we'll implement some control flow, which relates to user activity, to help us decide what functionality we'll need to include. The following diagram illustrates how the control flow may look like:



In the previous diagram, we anticipate where data can be shared using concurrent and parallel processes to locate points of failure. If we design concurrent applications in such graphical ways, we're less likely to find race conditions later on.

While we talked about how Go helps you to locate these after the application has completed running, our ideal development workflow is to attempt to cut these problems off at the start.

Developing our server requirements

Now that we have an idea of how the scheduling process should work, we need to identify components that our application will need. In this case, the components are as follows:

- A web server handler
- A template for output
- A system for determining dates and times

Web server

In our visualizing concurrency example from the previous chapter, we used Go's built-in `http` package, and we'll do the same here. There are a number of good frameworks out there for this, but they primarily extend the core Go functionality rather than reinventing the wheel. The following are a few of these functionalities, listed from lightest to heaviest:

- Web.go: `http://webgo.io/`

  Web.go is very lightweight and lean, and it provides some routing functionality not available in the `net/http` package.

- Gorilla: `http://www.gorillatoolkit.org/`

  Gorilla is a Swiss army knife to augment the `net/http` package. It's not particularly heavy, and it is fast, utilitarian, and very clean.

- Revel: `http://robfig.github.io/revel/`

  Revel is the heaviest of the three, but it focuses on a lot of intuitive code, caching, and performance. Look for it if you need something mature that will face a lot of traffic.

In *Chapter 6, C10K – A Non-blocking Web Server in Go*, we'll roll our own web server and framework with the sole goal of extreme high performance.

The Gorilla toolkit

For this application, we'll partially employ the Gorilla web toolkit. Gorilla is a fairly mature web-serving platform that fulfills a few of our needs here natively, namely the ability to include regular expressions in our URL routing. (Note: Web.Go also extends some of this functionality.) Go's internal HTTP routing handler is rather simplistic; you can extend this, of course, but we'll take a shortcut down a well-worn and reliable path here.

We'll use this package solely for ease of URL routing, but the Gorilla web toolkit also includes packages to handle cookies, sessions, and request variables. We'll examine this package a little closer in *Chapter 6, C10K – A Non-blocking Web Server in Go*.

Using templates

As Go is intended as a system language, and as system languages often deal with the creation of servers with clients, some care was put into making it a well-featured alternative to create web servers.

Anyone who's dealt with a "web language" will know that on top of that you'll need a framework, ideally one that handles the presentation layer for the web. While it's true that if you take on such a project you'll likely look for or build your own framework, Go makes the templating side of things very easy.

The template package comes in two varieties: `text` and `http`. Though they both serve different end points, the same properties—affording dynamism and flexibility—apply to the presentation layer rather than strictly the application layer.

The `text` template package is intended for general plaintext documents, while the `http` template package handles the generation of HTML and related documents.

These templating paradigms are all too common these days; if you look at the `http/template` package, you'll find some very strong similarities to Mustache, one of the more popular variants. While there is a Mustache port in Go, there's nothing there that isn't handled by default in the template package.

For more information on Mustache, visit `http://mustache.github.io/`.

One potential advantage to Mustache is its availability in other languages. If you ever feel the need to port some of your application logic to another language (or existing templates into Go), utilizing Mustache could be advantageous. That said, you sacrifice a lot of the extended functionality of Go templates, namely the ability to take out Go code from your compiled package and move it directly into template control structures. While Mustache (and its variants) has control flows, they may not mirror Go's templating system. Take the following example:

```
<ul>
{{range .Users}}
<li>A User </li>
{{end}}
</ul>
```

Given the familiarity with Go's logic structures, it makes sense to keep them consistent in our templating language as well.

> We won't show all the specific templates in this thread, but we will show the output. If you wish to peruse them, they're available at `mastergoco.com/chapters/3/templates`.

Time

We're not doing a whole lot of math here; time will be broken into hour blocks and each will be set to either occupied or available. At this time, there aren't a lot of external `date/time` packages for Go. We're not doing any heavy-date math, but it doesn't really matter because Go's `time` package should suffice even if we were.

In fact, as we have literal hour blocks from 9 a.m. to 5 p.m., we just set these to the 24-hour time values of 9-17, and invoke a function to translate them into linguistic dates.

Endpoints

We'll want to identify the REST endpoints (via `GET` requests) and briefly describe how they'll work. You can think of these as modules or methods in the model-view-controller architecture. The following is a list of the endpoint patterns we'll use:

- `entrypoint/register/{name}`: This is where we'll go to add a name to the list of users. If the user exists, it will fail.
- `entrypoint/viewusers`: Here, we'll present a list of users with their timeslots, both available and occupied.
- `entrypoint/schedule/{name}/{time}`: This will initialize an attempt to schedule an appointment.

Each will have an accompanying template that will report the status of the intended action.

Custom structs

We'll deal with users and responses (web pages), so we need two structs to represent each. One struct is as follows:

```
type User struct {
  Name string
  email string
  times[int] bool
}
```

The other struct is as follows:

```
type Page struct {
  Title string
  Body string
}
```

We will keep the page as simple as possible. Rather than doing a lot of iterative loops, we will produce the HTML within the code for the most part.

Our endpoints for requests will relate to our previous architecture, using the following code:

```
func users(w http.ResponseWriter, r *http.Request) {
}
func register(w http.ResponseWriter, r *http.Request) {
}
func schedule(w http.ResponseWriter, r *http.Request) {
}
```

A multiuser Appointments Calendar

In this section, we'll quickly look at our sample Appointments Calendar application, which attempts to control consistency of specific elements to avoid obvious race conditions. The following is the full code, including the routing and templating:

```
package main

import(
  "net/http"
  "html/template"
  "fmt"
```

```
    "github.com/gorilla/mux"
    "sync"
    "strconv"
  )

  type User struct {
    Name string
    Times map[int] bool
    DateHTML template.HTML
  }

  type Page struct {
    Title string
    Body template.HTML
    Users map[string] User
  }

  var usersInit map[string] bool
  var userIndex int
  var validTimes []int
  var mutex sync.Mutex
  var Users map[string]User
  var templates = template.Must(template.New("template").
  ParseFiles("view_users.html", "register.html"))


  func register(w http.ResponseWriter, r *http.Request){
    fmt.Println("Request to /register")
    params := mux.Vars(r)
    name := params["name"]

    if _,ok := Users[name]; ok {
      t,_ := template.ParseFiles("generic.txt")
      page := &Page{ Title: "User already exists", Body:
        template.HTML("User " + name + " already exists")}
      t.Execute(w, page)
    }  else {
            newUser := User { Name: name }
            initUser(&newUser)
            Users[name] = newUser
            t,_ := template.ParseFiles("generic.txt")
            page := &Page{ Title: "User created!", Body:
              template.HTML("You have created user "+name)}
```

```
          t.Execute(w, page)
     }


 }

 func dismissData(st1 int, st2 bool) {

 // Does nothing in particular for now other than avoid Go compiler
   errors
 }

 func formatTime(hour int) string {
   hourText := hour
   ampm := "am"
   if (hour > 11) {
     ampm = "pm"
   }
   if (hour > 12) {
     hourText = hour - 12;
   }
 fmt.Println(ampm)
   outputString := strconv.FormatInt(int64(hourText),10) + ampm


   return outputString
 }

 func (u User) FormatAvailableTimes() template.HTML { HTML := ""
   HTML += "<b>"+u.Name+"</b> - "

   for k,v := range u.Times { dismissData(k,v)

     if (u.Times[k] == true) { formattedTime := formatTime(k) HTML
       += "<a href='/schedule/"+u.Name+"/"
         +strconv.FormatInt(int64(k),10)+"'
           class='button'>"+formattedTime+"</a> "

     } else {

     }

  } return template.HTML(HTML)
```

```
}

func users(w http.ResponseWriter, r *http.Request) {
  fmt.Println("Request to /users")


  t,_ := template.ParseFiles("users.txt")
  page := &Page{ Title: "View Users", Users: Users}
  t.Execute(w, page)
}

func schedule(w http.ResponseWriter, r *http.Request) {
  fmt.Println("Request to /schedule")
  params := mux.Vars(r)
  name := params["name"]
  time := params["hour"]
  timeVal,_ := strconv.ParseInt( time, 10, 0 )
  intTimeVal := int(timeVal)

  createURL := "/register/"+name

  if _,ok := Users[name]; ok {
    if Users[name].Times[intTimeVal] == true {
      mutex.Lock()
      Users[name].Times[intTimeVal] = false
      mutex.Unlock()
      fmt.Println("User exists, variable should be modified")
      t,_ := template.ParseFiles("generic.txt")
      page := &Page{ Title: "Successfully Scheduled!", Body:
        template.HTML("This appointment has been scheduled. <a
          href='/users'>Back to users</a>")}

      t.Execute(w, page)

    }  else {
            fmt.Println("User exists, spot is taken!")
            t,_ := template.ParseFiles("generic.txt")
            page := &Page{ Title: "Booked!", Body:
              template.HTML("Sorry, "+name+" is booked for
              "+time+" <a href='/users'>Back to users</a>")}
      t.Execute(w, page)

    }
  }  else {
          fmt.Println("User does not exist")
          t,_ := template.ParseFiles("generic.txt")
          page := &Page{ Title: "User Does Not Exist!", Body:
            template.HTML( "Sorry, that user does not exist. Click
              <a href='"+createURL+"'>here</a> to create it. <a
                href='/users'>Back to users</a>")}
    t.Execute(w, page)
  }
  fmt.Println(name,time)
}

func defaultPage(w http.ResponseWriter, r *http.Request) {

}

func initUser(user *User) {

  user.Times = make(map[int] bool)
  for i := 9; i < 18; i ++ {
    user.Times[i] = true
  }

}

func main() {
  Users = make(map[string] User)
  userIndex = 0
  bill := User {Name: "Bill"  }
  initUser(&bill)
  Users["Bill"] = bill
  userIndex++

  r := mux.NewRouter()  r.HandleFunc("/", defaultPage)
    r.HandleFunc("/users", users)
      r.HandleFunc("/register/{name:[A-Za-z]+}", register)
        r.HandleFunc("/schedule/{name:[A-Za-z]+}/{hour:[0-9]+}",
          schedule)     http.Handle("/", r)

  err := http.ListenAndServe(":1900", nil)  if err != nil {    //
    log.Fatal("ListenAndServe:", err)    }


}
```

Note that we seeded our application with a user, Bill. If you attempt to hit `/register/bill|bill@example.com`, the application will report that the user exists.

As we control the most sensitive data through channels, we avoid any race conditions. We can test this in a couple of ways. The first and easiest way is to keep a log of how many successful appointments are registered, and run this with Bill as the default user.

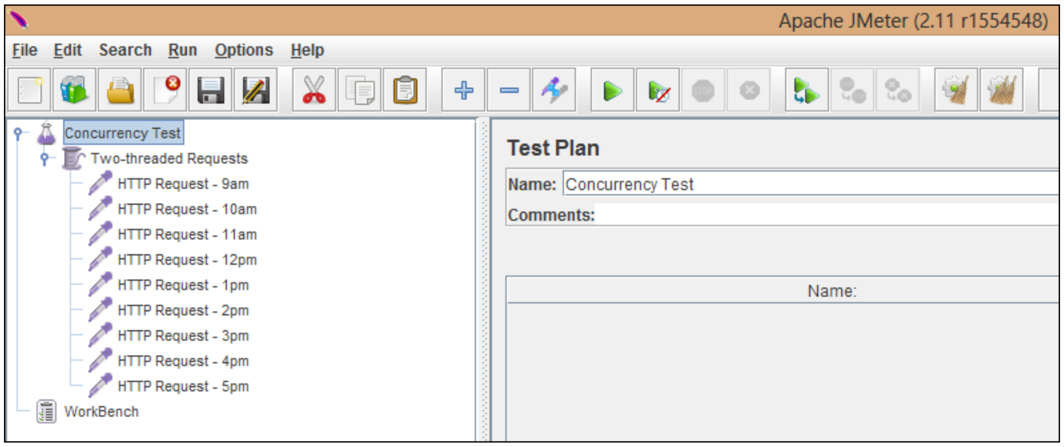
We can then run a concurrent load tester against the action. There are a number of such testers available, including Apache's ab and Siege. For our purposes, we'll use JMeter, primarily because it permits us to test against multiple URLs concurrently.

Although we're not necessarily using JMeter for load testing (rather, we use it to run concurrent tests), load testers can be extraordinarily valuable ways to find bottlenecks in applications at scales that don't yet exist.

For example, if you built a web application that had a blocking element and had 5,000-10,000 requests per day, you may not notice it. But at 5 million-10 million requests per day, it might result in the application crashing.

In the dawn of network servers, this is what happened; servers scaled until one day, suddenly, they couldn't scale further. Load/stress testers allow you to simulate traffic in order to better detect these issues and inefficiencies.

Given that we have one user and eight hours in a day, we should end our script with no more than eight total successful appointments. Of course, if you hit the `/register` endpoint, you will see eight times as many users as you've added. The following screenshot shows our benchmark test plan in JMeter:



Concurrency Test
Two-threaded Requests
HTTP Request - 9am
HTTP Request - 10am
HTTP Request - 11am
HTTP Request - 12pm
HTTP Request - 1pm
HTTP Request - 2pm
HTTP Request - 3pm
HTTP Request - 4pm
HTTP Request - 5pm
WorkBench

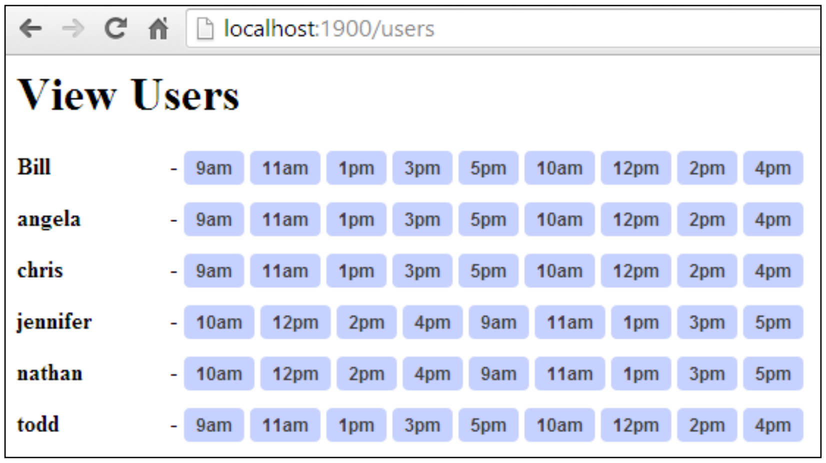
Test Plan
Name: Concurrency Test
Comments:
Name:

When you run your application, keep an eye on your console; at the end of our load test, we should see the following message:

```
Total registered appointments: 8
```

Had we designed our application as per the initial graphical mockup representation in this chapter (with race conditions), it's plausible—and in fact likely—that we'd register far more appointments than actually existed.

By isolating potential race conditions, we guarantee data consistency and ensure that nobody is waiting on an appointment with an otherwise occupied attendee. The following screenshot is the list we present of all the users and their available appointment times:



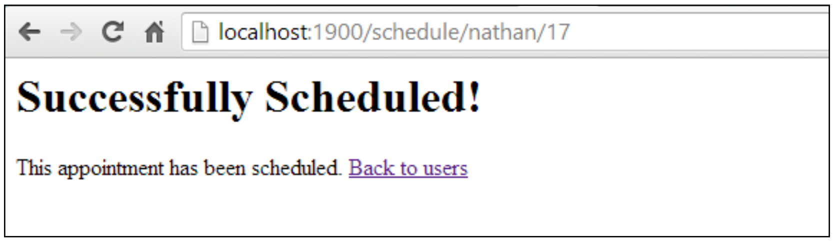
localhost:1900/users

View Users

Bill	-	9am	11am	1pm	3pm	5pm	10am	12pm	2pm	4pm
angela	-	9am	11am	1pm	3pm	5pm	10am	12pm	2pm	4pm
chris	-	9am	11am	1pm	3pm	5pm	10am	12pm	2pm	4pm
jennifer	-	10am	12pm	2pm	4pm	9am	11am	1pm	3pm	5pm
nathan	-	10am	12pm	2pm	4pm	9am	11am	1pm	3pm	5pm
todd	-	9am	11am	1pm	3pm	5pm	10am	12pm	2pm	4pm

The previous screenshot is our initial view that shows us available users and their available time slots. By selecting a timeslot for a user, we'll attempt to book them for that particular time. We'll start with Nathan at 5 p.m.

The following screenshot shows what happens when we attempt to schedule with an available user:

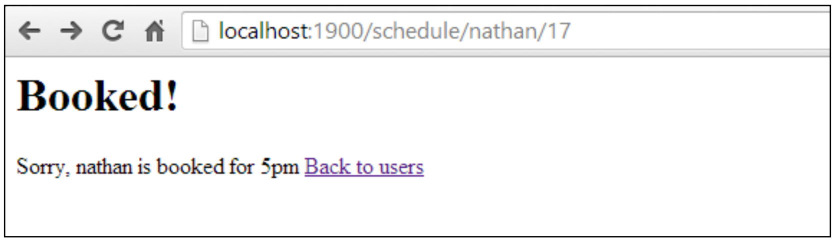


localhost:1900/schedule/nathan/17

Successfully Scheduled!

This appointment has been scheduled. Back to users

However, if we attempt to book again (even simultaneously), we'll be greeted with a sad message that Nathan cannot see us at 5 p.m, as shown in the following screenshot:



Booked!

Sorry, nathan is booked for 5pm Back to users

With that, we have a multiuser calendar app that allows for creating new users, scheduling, and blocking double-bookings.

Let's look at a few interesting new points in this application.

First, you will notice that we use a template called `generic.txt` for most parts of the application. There's not much to this, only a page title and body filled in by each handler. However, on the `/users` endpoint, we use `users.txt` as follows:

```
<html>
<head>
  <meta http-equiv="Content-Type" content="text/html; charset=utf-
    8">
  <title>{{.Title}}</title>
</head>
<body>

<h1>{{.Title}}</h1>

{{range .Users}}
<div class="user-row">

  {{.FormatAvailableTimes}}

</div>
{{end}}

</body>
</html>
```

We mentioned the range-based functionality in templates, but how does `{{.FormatAvailableTimes}}` work? In any given context, we can have type-specific functions that process the data in more complex ways than are available strictly in the template lexer.

In this case, the `User` struct is passed to the following line of code:

```
func (u User) FormatAvailableTimes() template.HTML {
```

This line of code then performs some conditional analysis and returns a string with some time conversion.

In this example, you can use either a channel to control the flow of `User.times` or an explicit mutex as we have here. We don't want to limit all locks, unless absolutely necessary, so we only invoke the `Lock()` function if we've determined the request has passed the tests necessary to modify the status of any given user/time pair. The following code shows where we set the availability of a user within a mutual exclusion:

```
if _,ok := Users[name]; ok {
  if Users[name].Times[intTimeVal] == true {
    mutex.Lock()
    Users[name].Times[intTimeVal] = false
    mutex.Unlock()
```

The outer evaluation checks that a user by that name (key) exists. The second evaluation checks that the time availability exists (true). If it does, we lock the variable, set it to `false`, and then move onto output rendering.

Without the `Lock()` function, many concurrent connections can compromise the consistency of data and cause the user to have more than one appointment in a given hour.

A note on style

You'll note that despite preferring camelCase for most of our variables, we have some uppercase variables within structs. This is an important Go convention worth mentioning: any struct variable that begins with a capital letter is **public**. Any variable that begins with a lowercase letter is **private**.

If you attempt to output a private (or nonexistent) variable in your template files, template rendering will fail.

A note on immutability

Note that whenever possible, we'll avoid using the string type for comparative operations, especially in multithreaded environments. In the previous example, we use integers and Booleans to decide availability for any given user. In some languages, you may feel empowered to assign the time values to a string for ease of use. For the most part, this is fine, even in Go; but assuming that we have an infinitely scalable, shared calendar application, we run the risk of introducing memory issues if we utilize strings in this way.

The string type is the sole immutable type in Go; this is noteworthy if you end up assigning and reassigning values to a string. Assuming that memory is yielded after a string is converted to a copy, this is not a problem. However, in Go (and a couple of other languages), it's entirely possible to keep the original value in memory. We can test this using the following example:

```
func main() {

  testString := "Watch your top / resource monitor"
  for i:= 0; i < 1000; i++ {

    testString = string(i)

  }
  doNothing(testString)

  time.Sleep(10 * time.Second)


}
```

When run in Ubuntu, this takes approximately 1.0 MB of memory; some of that no doubt overhead, but a useful reference point. Let's up the ante a bit—though having 1,000 relatively small pointers won't have much impact—using the following line of code:

```
for i:= 0; i < 100000000; i++ {
```

Now, having gone through 100 million memory assignments, you can see the impact on memory (it doesn't help that the string itself is at this point longer than the initial, but it doesn't account for the full effect). Garbage collection takes place here too, which impacts CPU. On our initial test here, both CPU and memory spiked. If we substitute this for an integer or a Boolean assignment, we get much smaller footprints.

This is not exactly a real-world scenario, but it's worth noting in a concurrent environment where garbage collection must happen so we can evaluate the properties and types of our logic.

It's also entirely possible, depending on your current version of Go, your machine(s), and so on, and this could run as efficiently in either scenario. While that might seem fine, part of our concurrent strategy planning should involve the possibility that our application will scale in input, output, physical resources, or all of them. Just because something works well now doesn't mean it's not worth implementing efficiencies that will keep it from causing performance problems at a 100x scale.

If you ever encounter a place where a string is logical, but you want or could benefit from a mutable type, consider a byte slice instead.

A constant is, of course, also immutable, but given that's the implied purpose of a constant variable, you should already know this. A mutable constant variable is, after all, an oxymoron.

Summary

This chapter has hopefully directed you towards exploring methods to plan and chart out your concurrent applications before delving in. By briefly touching on race conditions and data consistency, we attempted to highlight the importance of anticipatory design. At the same time, we utilized a few tools for identifying such issues, should they occur.

Creating a robust script flowchart with concurrent processes will help you locate possible pitfalls before you create them, and it will give you a better sense of how (and when) your application should be making decisions with logic and data.

In the next chapter, we'll examine data consistency issues and look at advanced channel communication options in an effort to avoid needless and often expensive mitigating functions, mutexes, and external processes.

4
Data Integrity in an Application

By now, you should be comfortable with the models and tools provided in Go's core to provide mostly race-free concurrency.

We can now create goroutines and channels with ease, manage basic communication across channels, coordinate data without race conditions, and detect such conditions as they arise.

However, we can neither manage larger distributed systems nor deal with potentially lower-level consistency problems. We've utilized a basic and simplistic mutex, but we are about to look at a more complicated and expressive way of handling mutual exclusions.

By the end of this chapter, you should be able to expand your concurrency patterns from the previous chapter into distributed systems using a myriad of concurrency models and systems from other languages. We'll also look—at a high level—at some consistency models that you can utilize to further express your precoding strategies for single-source and distributed applications.

Getting deeper with mutexes and sync

In *Chapter 2*, *Understanding the Concurrency Model*, we introduced `sync.mutex` and how to invoke a mutual exclusion lock within your code, but there's some more nuance to consider with the package and the mutex type.

We've mentioned that in an ideal world, you should be able to maintain synchronization in your application by using goroutines alone. In fact, this would probably be best described as the canonical method within Go, although the `sync` package does provide a few other utilities, including mutexes.

Whenever possible, we'll stick with goroutines and channels to manage consistency, but the mutex does provide a more traditional and granular approach to lock and access data. If you've ever managed another concurrent language (or package within a language), odds are you've had experience with either a mutex or a philosophical analog. In the following chapters, we'll look at ways of extending and exploiting mutexes to do a little more out of the box.

If we look at the `sync` package, we'll see there are a couple of different mutex structs.

The first is `sync.mutex`, which we've explored—but another is `RWMutex`. The `RWMutex` struct provides a multireader, single-writer lock. These can be useful if you want to allow reads to resources but provide mutex-like locks when a write is attempted. They can be best utilized when you expect a function or subprocess to do frequent reads but infrequent writes, but it still cannot afford a dirty read.

Let's look at an example that updates the date/time every 10 seconds (acquiring a lock), yet outputs the current value every other second, as shown in the following code:

```
package main

import (
  "fmt"
  "sync"
  "time"
)

type TimeStruct struct {
  totalChanges int
  currentTime time.Time
  rwLock sync.RWMutex
}

var TimeElement TimeStruct

func updateTime() {
  TimeElement.rwLock.Lock()
  defer TimeElement.rwLock.Unlock()
  TimeElement.currentTime = time.Now()
  TimeElement.totalChanges++
}

func main() {

  var wg sync.WaitGroup
```

```
    TimeElement.totalChanges = 0
    TimeElement.currentTime = time.Now()
    timer := time.NewTicker(1 * time.Second)
    writeTimer := time.NewTicker(10 * time.Second)
    endTimer := make(chan bool)

    wg.Add(1)
    go func() {

      for {
        select {
        case <-timer.C:
          fmt.Println(TimeElement.totalChanges,
            TimeElement.currentTime.String())
        case <-writeTimer.C:
          updateTime()
        case <-endTimer:
          timer.Stop()
          return
        }

      }

    }()

    wg.Wait()
    fmt.Println(TimeElement.currentTime.String())
  }
```

We don't explicitly run `Done()` on our `WaitGroup` struct, so this will run in perpetuity.

There are two different methods for performing locks/unlocks on `RWMutex`:

- `Lock()`: This will block variables for both reading and writing until an `Unlock()` method is called
- `happenedRlock()`: This locks bound variables solely for reads

The second method is what we've used for this example, because we want to simulate a real-world lock. The net effect is the `interval` function that outputs the current time that will return a single dirty read before `rwLock` releases the read lock on the `currentTime` variable. The `Sleep()` method exists solely to give us time to witness the lock in motion. An `RWLock` struct can be acquired by many readers or by a single writer.

The cost of goroutines

As you work with goroutines, you might get to a point where you're spawning dozens or even hundreds of them and wonder if this is going to be expensive. This is particularly true if your previous experience with concurrent and/or parallel programming was primarily thread-based. It's commonly accepted that maintaining threads and their respective stacks can begin to bog down a program with performance issues. There are a few reasons for this, which are as follows:

- Memory is required just for the creation of a thread
- Context switching at the OS level is more complex and expensive than in-process context switching
- Very often, a thread is spawned for a very small process that could be handled otherwise

It's for these reasons that a lot of modern concurrent languages implement something akin to goroutines (C# uses the async and await mechanism, Python has greenlets/green threads, and so on) that simulate threads using small-scale context switching.

However, it's worth knowing that while goroutines are (or can be) cheap and cheaper than OS threads, they are not free. At a large (perhaps enormous) measure, even cheap and light goroutines can impact performance. This is particularly important to note as we begin to look at distributed systems, which often scale larger and at faster rates.

The difference between running a function directly and running it in a goroutine is negligible of course. However, keep in mind that Go's documentation states:

> *It is practical to create hundreds of thousands of goroutines in the same address space.*

Given that stack creation uses a few kilobytes per goroutine, in a modern environment, it's easy to see how that could be perceived as a nonfactor. However, when you start talking about thousands (or millions) of goroutines running, it can and likely will impact the performance of any given subprocess or function. You can test this by wrapping functions in an arbitrary number of goroutines and benchmarking the average execution time and—more importantly—memory usage. At approximately 5KB per goroutine, you may find that memory can become a factor, particularly on low-RAM machines or instances. If you have an application that runs heavy on a high-powered machine, imagine it reaching criticality in one or more lower-powered machines. Consider the following example:

```
for i:= 0; i < 1000000000; i++ {
  go someFunction()
}
```

Even if the overhead for the goroutine is cheap, what happens at 100 million or—as we have here—a billion goroutines running?

As always, doing this in an environment that utilizes more than a single core can actually increase the overhead of this application due to the costs of OS threading and subsequent context switching.

These issues are almost always the ones that are invisible unless and until an application begins to scale. Running on your machine is one thing, running at scale across a distributed system with what amounts to low-powered application servers is quite another.

The relationship between performance and data consistency is important, particularly if you start utilizing a lot of goroutines with mutual exclusions, locks, or channel communication.

This becomes a larger issue when dealing with external, more permanent memory sources.

Working with files

Files are a great example of areas where data consistency issues such as race conditions can lead to more permanent and catastrophic problems. Let's look at a piece of code that might continuously attempt to update a file to see where we could run into race conditions, which in turn could lead to bigger problems such as an application failing or losing data consistency:

```
package main

import(
  "fmt"
  "io/ioutil"
  "strconv"
  "sync"
)


func writeFile(i int) {

  rwLock.RLock();
  ioutil.WriteFile("test.txt",
    []byte(strconv.FormatInt(int64(i),10)), 0x777)
  rwLock.RUnlock();
```

```
    writer<-true

  }

  var writer chan bool
  var rwLock sync.RWMutex

  func main() {

    writer = make(chan bool)

    for i:=0;i<10;i++ {
      go writeFile(i)
    }


    <-writer
    fmt.Println("Done!")
  }
```

Code involving file operations are rife for these sorts of potential issues, as mistakes are specifically *not ephemeral* and can be locked in time forever.

If our goroutines block at some critical point or the application fails midway through, we could end up with a file that has invalid data in it. In this case, we're simply iterating through some numbers, but you can also apply this situation to one involving database or datastore writes—the potential exists for persistent bad data instead of temporary bad data.

This is not a problem that is exclusively solved by channels or mutual exclusions; rather, it requires some sort of sanity check at every step to make certain that data is where you and the application expect it to be at every step in the execution. Any operation involving `io.Writer` relies on primitives, which Go's documentation explicitly notes that we should not assume they are safe for parallel execution. In this case, we have wrapped the file writing in a mutex.

Getting low – implementing C

One of the most interesting developments in language design in the past decade or two is the desire to implement lower-level languages and language features via API. Java lets you do this purely externally, and Python provides a C library for interaction between the languages. It warrants mentioning that the reasons for doing this vary—among them applying Go's concurrency features as a wrapper for legacy C code—and you will likely have to deal with some of the memory management associated with introducing unmanaged code to garbage-collected applications.

Go takes a hybrid approach, allowing you to call a C interface through an import, which requires a frontend compiler such as GCC:

```
import "C"
```

So why would we want to do this?

There are some good and bad reasons to implement C directly in your project. An example of a good reason might be to have direct access to the inline assembly, which you can do in C but not directly in Go. A bad reason could be any that has a solution inherent in Golang itself.

To be fair, even a bad reason is not bad if you build your application reliably, but it does impose an additional level of complexity to anyone else who might use your code. If Go can satisfy the technical and performance requirements, it's always better to use a single language in a single project.

There's a famous quote from C++ creator Bjarne Stroustrup on C and C++:

> *C makes it easy to shoot yourself in the foot; C++ makes it harder, but when you do, it blows your whole leg off.*

Jokes aside (Stroustrup has a vast collection of such quips and quotes), the fundamental reasoning is that the complexity of C often prevents people from accidentally doing something catastrophic.

As Stroustrup says, C makes it easy to make big mistakes, but the repercussions are often smaller due to language design than higher-level languages. Issues dealing with security and stability are easy to be introduced in any low-level language.

By simplifying the language, C++ provides abstractions that make low-level operations easier to carry out. You can see how this might apply to using C directly in Go, given the latter language's syntactical sweetness and programmer friendliness.

That said, working with C can highlight some of the potential pitfalls with regard to memory, pointers, deadlocks, and consistency, so we'll touch upon a simple example as follows:

```
package main

// #include <stdio.h>
// #include <string.h>
//  int string_length (char* str) {
//    return strlen(str);
//  }
import "C"
import "fmt"
func main() {
  v := C.CString("Don't Forget My Memory Is Not Visible To Go!")
  x := C.string_length(v)
  fmt.Println("A C function has determined your string
    is",x,"characters in length")
}
```

Touching memory in cgo

The most important takeaway from the preceding example is to remember that anytime you go into or out of C, you need to manage memory manually (or at least more directly than with Go alone). If you've ever worked in C (or C++), you know that there's no automatic garbage collection, so if you request memory space, you must also free it. Calling C from Go does not preclude this.

The structure of cgo

Importing C into Go will take you down a syntactical side route, as you probably noticed in the preceding code. The first thing that will appear glaringly different is the actual implementation of C code within your application.

Any code (in comments to stop Go's compiler from failing) directly above the `import "C"` directive will be interpreted as C code. The following is an example of a C function declared above our Go code:

```
/*
  int addition(int a, int b) {
    return a + b;
  }
```

Bear in mind that Go won't validate this, so if you make an error in your C code, it could lead to silent failure.

Another related warning is to remember your syntax. While Go and C share a lot of syntactical overlap, leave off a curly bracket or a semicolon and you could very well find yourself in one of those silent failure situations. Alternately, if you're working in the C part of your application and you go back to Go, you will undoubtedly find yourself wrapping loop expressions in parentheses and ending your lines with semicolons.

Also remember that you'll frequently have to handle type conversions between C and Go that don't have one-to-one analogs. For example, C does not have a built-in string type (you can, of course, include additional libraries for types), so you may need to convert between strings and char arrays. Similarly, `int` and `int64` might need some nonimplicit conversion, and again, you may not get the debugging feedback that you might expect when compiling these.

The other way around

Using C within Go is obviously a potentially powerful tool for code migration, implementing lower-level code, and roping in other developers, but what about the inverse? Just as you can call C from within Go, you can call Go functions as external functions within your embedded C.

The end game here is the ability to work with and within C and Go in the same application. By far the easiest way to handle this is by using gccgo, which is a frontend for GCC. This is different than the built-in Go compiler; it is possible to go back and forth between C and Go without gccgo, but using it makes this process much simpler.

gopart.go

The following is the code for the Go part of the interaction, which the C part will call as an external function:

```
package main

func MyGoFunction(num C.int) int {

  squared := num * num
  fmt.Println(num,"squared is",squared)
  return squared
}
```

cpart.c

Now for the C part, where we make our call to our Go application's exported function `MyGoFunction`, as shown in the following code snippet:

```
#include <stdio.h>

extern int square_it(int) __asm__ ("cross.main.MyGoFunction")

int main() {

  int output = square_it(5)
  printf("Output: %d",output)
  return 0;
}
```

Makefile

Unlike using C directly in Go, at present, doing the inverse requires the use of a makefile for C compilation. Here's one that you can use to get an executable from the earlier simple example:

```
all: main

main: cpart.o cpart.c
    gcc cpart.o cpart.c -o main

gopart.o: gopart.go
    gccgo -c gopart.go -o gopart.o -fgo-prefix=cross

clean:
    rm -f main *.o
```

Running the makefile here should produce an executable file that calls the function from within C.

However, more fundamentally, cgo allows you to define your functions as external functions for C directly:

```
package output

import "C"

//export MyGoFunction
func MyGoFunction(num int) int {

  squared := num * num
  return squared
}
```

Next, you'll need to use the `cgo` tool directly to generate header files for C as shown in the following line of code:

```
go tool cgo goback.go
```

At this point, the Go function is available for use in your C application:

```
#include <stdio.h>
#include "_obj/_cgo_export.h"

extern int MyGoFunction(int num);

int main() {

  int result = MyGoFunction(5);
  printf("Output: %d",result);
  return 0;

}
```

Note that if you export a Go function that contains more than one return value, it will be available as a struct in C rather than a function, as C does not provide multiple variables returned from a function.

At this point, you may be realizing that the true power of this functionality is the ability to interface with a Go application directly from existing C (or even C++) applications.

While not necessarily a true API, you can now treat Go applications as linked libraries within C apps and vice versa.

One caveat about using `//export` directives: if you do this, your C code must reference these as extern-declared functions. As you may know, extern is used when a C application needs to call a function from another linked C file.

When we build our Go code in this manner, cgo generates the header file `_cgo_export.h`, as you saw earlier. If you want to take a look at that code, it can help you understand how Go translates compiled applications into C header files for this type of use:

```
/* Created by cgo - DO NOT EDIT. */
#include "_cgo_export.h"

extern void crosscall2(void (*fn)(void *, int), void *, int);

extern void _cgoexp_d133c8d0d35b_MyGoFunction(void *, int);

GoInt64 MyGoFunction(GoInt p0)
{
  struct {
    GoInt p0;
    GoInt64 r0;
  } __attribute__((packed)) a;
  a.p0 = p0;
  crosscall2(_cgoexp_d133c8d0d35b_MyGoFunction, &a, 16);
  return a.r0;
}
```

You may also run into a rare scenario wherein the C code is not exactly as you expect, and you're unable to cajole the compiler to produce what you expect. In that case, you're always free to modify the header file before the compilation of your C application, despite the `DO NOT EDIT` warning.

Getting even lower – assembly in Go

If you can shoot your foot off with C and you can blow your leg off with C++, just imagine what you can do with assembly in Go.

It isn't possible to use assembly directly in Go, but as Go provides access to C directly and C provides the ability to call inline assembly, you can indirectly use it in Go.

But again, just because something is possible doesn't mean it should be done—if you find yourself in need of assembly in Go, you should consider using assembly directly and connecting via an API.

Among the many roadblocks that you may encounter with assembly in (C and then in) Go is the lack of portability. Writing inline C is one thing—your code should be relatively transferable between processor instruction sets and operating systems—but assembly is obviously something that requires a lot of specificity.

All that said, it's certainly better to have the option to shoot yourself in the foot whether you choose to take the shot or not. Use great care when considering whether you need C or assembly directly in your Go application. If you can get away with communicating between dissonant processes through an API or interprocess conduit, always take that route first.

One very obvious drawback of using assembly in Go (or on its own or in C) is you lose the cross-compilation capabilities that Go provides, so you'd have to modify your code for every destination CPU architecture. For this reason, the only practical times to use Go in C are when there is a single platform on which your application should run.

Here's an example of what an ASM-in-C-in-Go application might look like. Keep in mind that we've included no ASM code, because it varies from one processor type to another. Experiment with some boilerplate assembly in the following `__asm__` section:

```
package main

/*
#include <stdio.h>

void asmCall() {

__asm__( "" );
    printf("I come from a %s","C function with embedded asm\n");
}
*/
import "C"

func main() {

    C.asmCall()

}
```

If nothing else, this may provide an avenue for delving deeper into ASM even if you're familiar with neither assembly nor C itself. The more high-level you consider C and Go to be, the more practical you might see this.

For most uses, Go (and certainly C) is low-level enough to be able to squeeze out any performance hiccups without landing at assembly. It's worth noting again that while you do lose some immediate control of memory and pointers in Go when you invoke C applications, that caveat applies tenfold with assembly. All of those nifty tools that Go provides may not work reliably or not work at all. If you think about the Go race detector, consider the following application:

```
package main

/*
int increment(int i) {
  i++;
  return i;
}
*/
import "C"
import "fmt"

var myNumber int

func main() {
  fmt.Println(myNumber)

  for i:=0;i<100;i++ {
    myNumber = int( C.increment(C.int(myNumber)) )
    fmt.Println(myNumber)
  }

}
```

You can see how tossing your pointers around between Go and C might leave you out in the dark when you don't get what you expect out of the program.

Keep in mind that here there is a somewhat unique and perhaps unexpected kicker to using goroutines with cgo; they are treated by default as blocking. This isn't to say that you can't manage concurrency within C, but it won't happen by default. Instead, Go may well launch another system thread. You can manage this to some degree by utilizing the runtime function `runtime.LockOSThread()`. Using `LockOSThread` tells Go that a particular goroutine should stay within the present thread and no other concurrent goroutine may use this thread until `runtime.UnlockOSThread()` is called.

The usefulness of this depends heavily on the necessity to call C or a C library directly; some libraries will play happily as new threads are created, a few others may segfault.

Another useful runtime call you should find useful within your Go code is `NumGcoCall()`. This returns the number of cgo calls made by a current process. If you need to lock and unlock threads, you can also use this to build an internal queue report to detect and prevent deadlocks.

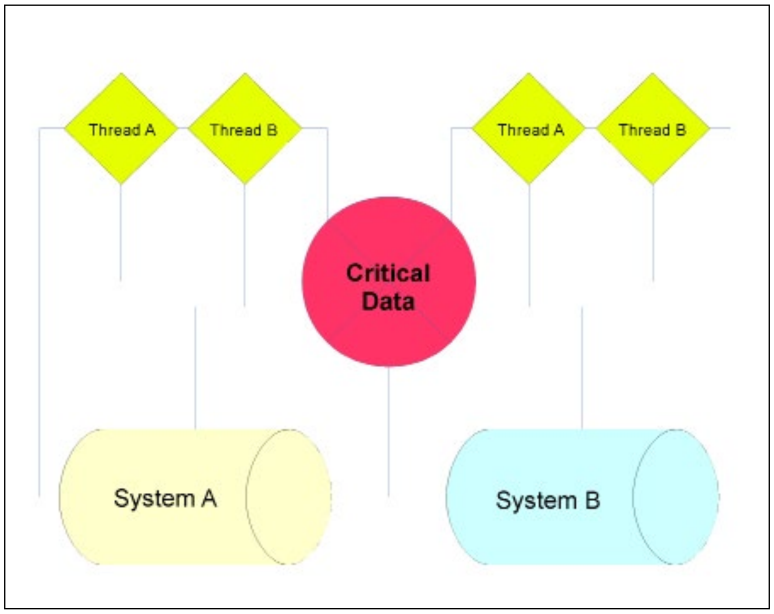
None of this precludes the possibility of race conditions should you choose to mix and match Go and C within goroutines.

Of course, C itself has a few race detector tools available. Go's race detector itself is based on the `ThreadSanitizer` library. It should go without saying that you probably do not want several tools that accomplish the same thing within a single project.

Distributed Go

So far, we've talked quite a bit about managing data within single machines, though with one or more cores. This is complicated enough as is. Preventing race conditions and deadlocks can be hard to begin with, but what happens when you introduce more machines (virtual or real) to the mix?

The first thing that should come to mind is that you can throw out a lot of the inherent tools that Go provides, and to a large degree that's true. You can mostly guarantee that Go can handle internal locking and unlocking of data within its own, singular goroutines and channels, but what about one or more additional instances of an application running? Consider the following model:



Here we see that either of these threads across either process could be reading from or writing to our **Critical Data** at any given point. With that in mind, there exists a need to coordinate access to that data.

At a very high level, there are two direct strategies for handling this, a distributed lock or consistency hash table (consistent hashing).

The first strategy is an extension of mutual exclusions except that we do not have direct and shared access to the same address space, so we need to create an abstraction. In other words, it's our job to concoct a lock mechanism that's visible to all available external entities.

The second strategy is a pattern designed specifically for caching and cache validation/invalidation, but it has relevancy here as well, because you can use it to manage where data lives in the more global address space.

However, when it comes to ensuring consistency across these systems, we need to go deeper than this general, high-level approach.

Split this model down the middle and it becomes easy: channels will handle the concurrent flow of data and data structures, and where they don't, you can use mutexes or low-level atomicity to add additional safeguards.

However, look to the right. Now you have another VM/instance or machine attempting to work with the same data. How can we make sure that we do not encounter reader/writer problems?

Some common consistency models

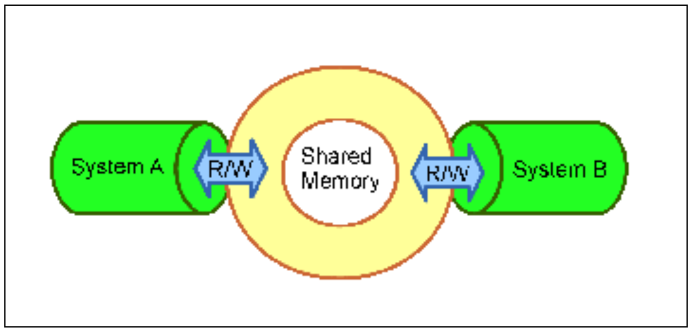
Luckily, there are some non-core Go solutions and strategies that we can utilize to improve our ability to control data consistency.

Let's briefly look at a few consistency models that we can employ to manage our data in distributed systems.

Distributed shared memory

On its own, a **Distributed Shared Memory** (**DSM**) system does not intrinsically prevent race conditions, as it is merely a method for more than one system to share real or partitioned memory.

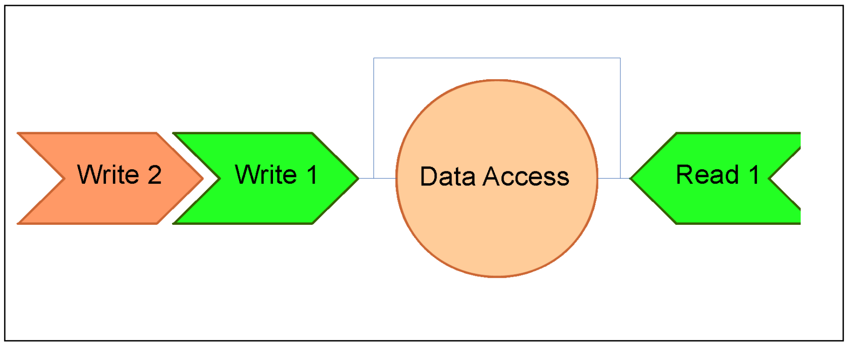
In essence, you can imagine two systems with 1 GB of memory, each allocating 500 MB to a shared memory space that is accessible and writable by each. Dirty reads are possible as are race conditions unless explicitly designed. The following figure is a visual representation of how two systems can coordinate using shared memory:



We'll look at one prolific but simple example of DSM shortly, and play with a library available to Go for test driving it.

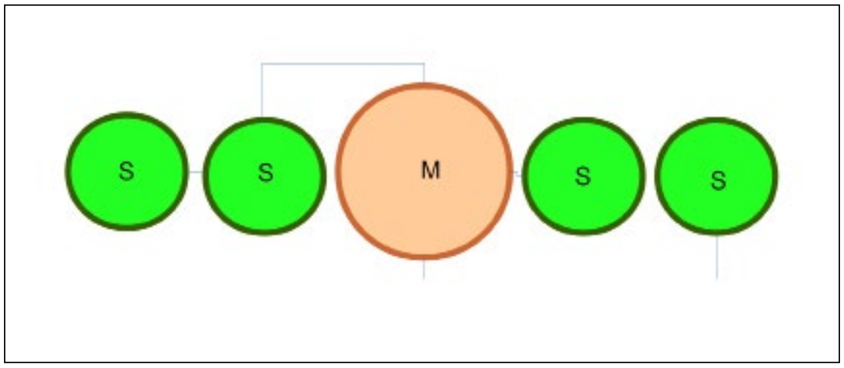
First-in-first-out – PRAM

Pipelined RAM (**PRAM**) consistency is a form of first-in-first-out methodology, in which data can be read in order of the queued writes. This means that writes read by any given, separate process may be different. The following figure represents this concept:



Looking at the master-slave model

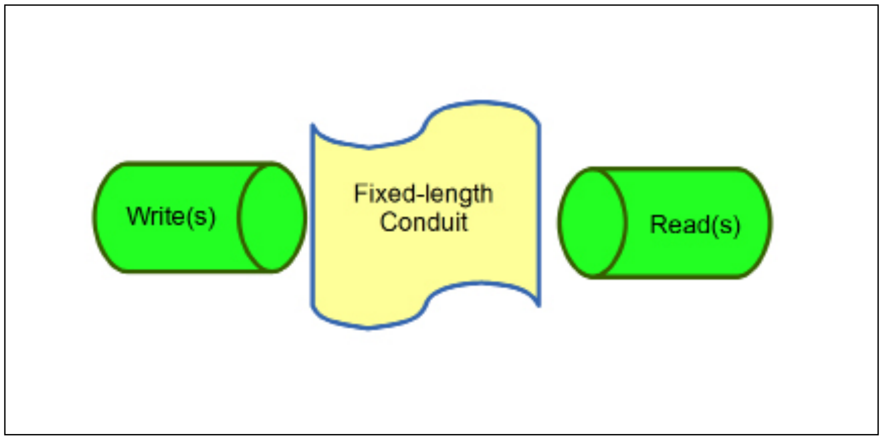
The master-slave consistency model is similar to the leader/follower model that we'll look at shortly, except that the master manages all operations on data and broadcasts rather than receiving write operations from a slave. In this case, replication is the primary method of transmission of changes to data from the master to the slave. In the following diagram, you will find a representation of the master-slave model with a master server and four slaves:



While we can simply duplicate this model in Go, we have more elegant solutions available to us.

The producer-consumer problem

In the classic producer-consumer problem, the producer writes chunks of data to a conduit/buffer, while a consumer reads chunks. The issue arises when the buffer is full: if the producer adds to the stack, the data read will not be what you intend. To avoid this, we employ a channel with waits and signals. This model looks a bit like the following figure:



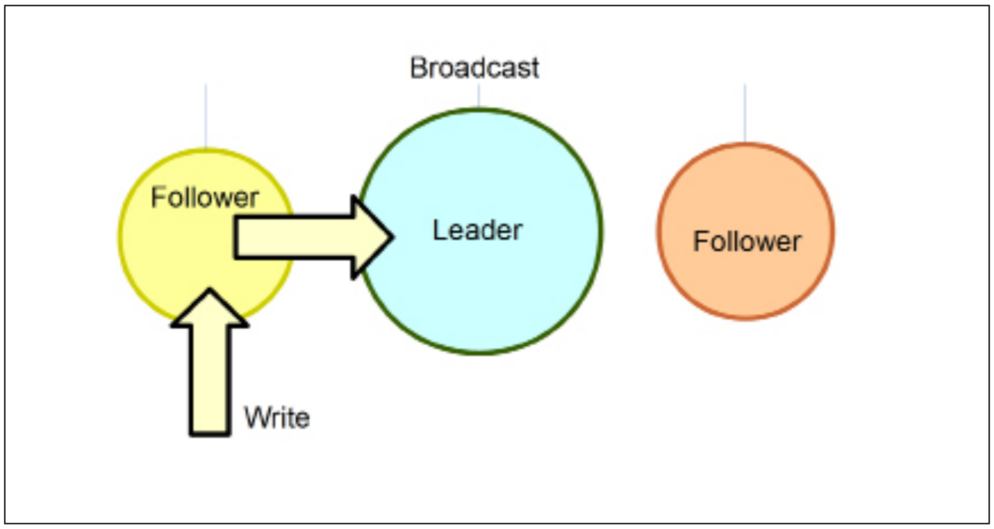
If you're looking for the semaphore implementation in Go, there is no explicit usage of the semaphore. However, think about the language here—fixed-size channels with waits and signals; sounds like a buffered channel. Indeed, by providing a buffered channel in Go, you give the conduit here an explicit length; the channel mechanism gives you the communication for waits and signals. This is incorporated in Go's concurrency model. Let's take a quick look at a producer-consumer model as shown in the following code:

```
package main

import(
  "fmt"
)

var comm = make(chan bool)
var done = make(chan bool)

func producer() {
  for i:=0; i< 10; i++ {
    comm <- true
  }
```

```
    done <- true
  }
  func consumer() {
    for {
      communication := <-comm
      fmt.Println("Communication from producer
        received!",communication)
    }
  }

  func main() {
    go producer()
    go consumer()
    <- done
    fmt.Println("All Done!")
  }
```

Looking at the leader-follower model

In the leader/follower model, writes are broadcasted from a single source to any followers. Writes can be passed through any number of followers or be restricted to a single follower. Any completed writes are then broadcasted to the followers. This can be visually represented as the following figure:



We can see a channel analog here in Go as well. We can, and have, utilized a single channel to handle broadcasts to and from other followers.

Atomic consistency / mutual exclusion

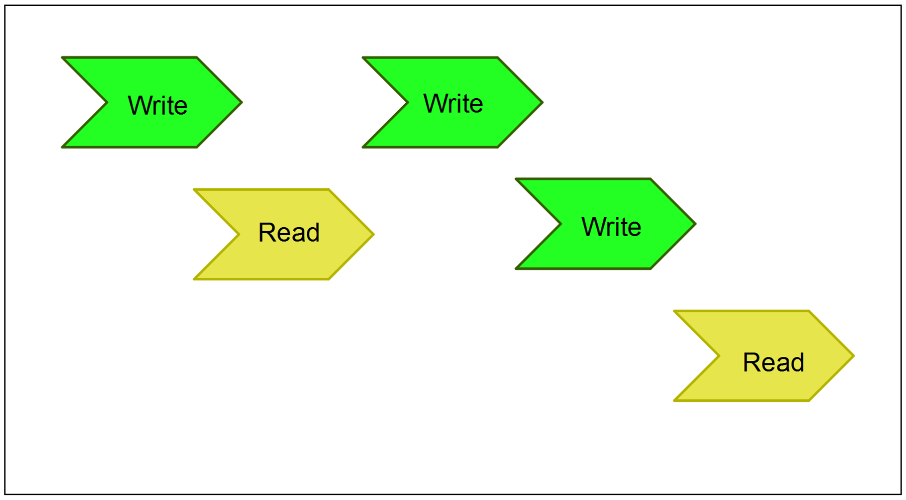
We've looked at atomic consistency quite a bit. It ensures that anything that is not created and used at essentially the same time will require serialization to guarantee the strongest form of consistency. If a value or dataset is not atomic in nature, we can always use a mutex to force linearizability on that data.

Serial or sequential consistency is inherently strong, but can also lead to performance issues and degradation of concurrency.

Atomic consistency is often considered the strongest form of ensuring consistency.

Release consistency

The release consistency model is a DSM variant that can delay a write's modifications until the time of first acquisition from a reader. This is known as lazy release consistency. We can visualize lazy release consistency in the following serialized model:



This model as well as an eager release consistency model both require an announcement of a release (as the name implies) when certain conditions are met. In the eager model, that condition requires that a write would be read by all read processes in a consistent manner.

In Go, there exists alternatives for this, but there are also packages out there if you're interested in playing with it.

Using memcached

If you're not familiar with memcache(d), it's a wonderful and seemingly obvious way to manage data across distributed systems. Go's built-in channels and goroutines are fantastic to manage communication and data integrity within a single machine's processes, but neither are built for distributed systems out of the box.

Memcached, as the name implies, allows data sharing memory among multiple instances or machines. Initially, memcached was intended to store data for quick retrieval. This is useful for caching data for systems with high turnover such as web applications, but it's also a great way to easily share data across multiple servers and/or to utilize shared locking mechanisms.

In our earlier models, memcached falls under DSM. All available and invoked instances share a common, mirrored memory space within their respective memories.

It's worth pointing out that race conditions can and do exist within memcached, and you still need a way to deal with that. Memcached provides one method to share data across distributed systems, but does not guarantee data atomicity. Instead, memcached operates on one of two methods for invalidating cached data as follows:

- Data is explicitly assigned a maximum age (after which, it is removed from the stack)
- Or data is pushed from the stack due to all available memory being used by newer data

It's important to note that storage within memcache(d) is, obviously, ephemeral and not fault resistant, so it should only be used where data should be passed without threat of critical application failure.

At the point where either of these conditions is met, the data disappears and the next call to this data will fail, meaning the data needs to be regenerated. Of course, you can work with some elaborate lock generation methods to make memcached operate in a consistent manner, although this is not standard built-in functionality of memcached itself. Let's look at a quick example of memcached in Go using Brad Fitz's gomemcache interface (`https://github.com/bradfitz/gomemcache`):

```
package main

import (
  "github.com/bradfitz/gomemcache/memcache"
  "fmt"
)
```

```
func main() {
     mC := memcache.New("10.0.0.1:11211", "10.0.0.2:11211",
       "10.0.0.3:11211", "10.0.0.4:11211")
     mC.Set(&memcache.Item{Key: "data", Value: []byte("30") })

     dataItem, err := mc.Get("data")
}
```

As you might note from the preceding example, if any of these memcached clients are writing to the shared memory at the same time, a race condition could still exist.

The key data can exist across any of the clients that have memcached connected and running at the same time.

Any client can also unset or overwrite the data at any time.

Unlike a lot of implementations, you can set some more complex types through memcached, such as structs, assuming they are serialized. This caveat means that we're somewhat limited with the data we can share directly. We are obviously unable to use pointers as memory locations will vary from client to client.

One method to handle data consistency is to design a master-slave system wherein only one node is responsible for writes and the other clients listen for changes via a key's existence.

We can utilize any other earlier mentioned models to strictly manage a lock on this data, although it can get especially complicated. In the next chapter, we'll explore some ways by which we can build distributed mutual exclusion systems, but for now, we'll briefly look at an alternative option.

Circuit

An interesting third-party library to handle distributed concurrency that has popped up recently is Petar Maymounkov's Go' circuit. Go' circuit attempts to facilitate distributed coroutines by assigning channels to listen to one or more remote goroutines.

The coolest part of Go' circuit is that simply including the package makes your application ready to listen and operate on remote goroutines and work with channels with which they are associated.

Go' circuit is in use at Tumblr, which proves it has some viability as a large-scale and relatively mature solutions platform.

Go' circuit can be found at `https://github.com/gocircuit/circuit`.

Installing Go' circuit is not simple—you cannot run a simple `go get` on it—and requires Apache Zookeeper and building the toolkit from scratch.

Once done, it's relatively simple to have two machines (or two processes if running locally) running Go code to share a channel. Each cog in this system falls under a sender or listener category, just as with goroutines. Given that we're talking about network resources here, the syntax is familiar with some minor modifications:

```
homeChannel := make(chan bool)

circuit.Spawn("etphonehome.example.com",func() {
  homeChannel <- true
})

for {
  select {
    case response := <- homeChannel:
      fmt.Print("E.T. has phoned home with:",response)

  }
}
```

You can see how this might make the communication between disparate machines playing with the same data a lot cleaner, whereas we used memcached primarily as a networked in-memory locking system. We're dealing with native Go code directly here; we have the ability to use circuits like we would in channels, without worrying about introducing new data management or atomicity issues. In fact, the circuit is built upon a goroutine itself.

This does, of course, still introduce some additional management issues, primarily as it pertains to knowing what remote machines are out there, whether they are active, updating the machines' statuses, and so on. These types of issues are best suited for a suite such as Apache Zookeeper to handle coordination of distributed resources. It's worth noting that you should be able to produce some feedback from a remote machine to a host: the circuit operates via passwordless SSH.

That also means you may need to make sure that user rights are locked down and that they meet with whatever security policies you may have in place.

You can find Apache Zookeeper at `http://zookeeper.apache.org/`.

Summary

Equipped now with some methods and models to manage not only local data across single or multithreaded systems, but also distributed systems, you should start to feel pretty comfortable with protecting the validity of data in concurrent and parallel processes.

We've looked at both forms of mutual exclusions for read and read/write locks, and we have started to apply these to distributed systems to prevent blocks and race conditions across multiple networked systems.

In the next chapter, we'll explore these exclusion and data consistency concepts a little deeper, building non-blocking networked applications and learn to work with timeouts and give parallelism with channels a deeper look.

We'll also dig a little deeper into the sync and OS packages, in particular looking at the `sync.atomic` operations.

5
Locks, Blocks, and Better Channels

Now that we're starting to get a good grasp of utilizing goroutines in safe and consistent ways, it's time to look a bit more at what causes code blocking and deadlocks. Let's also explore the `sync` package and dive into some profiling and analysis.

So far, we've built some relatively basic goroutines and complementary channels, but we now need to utilize some more complex communication channels between our goroutines. To do this, we'll implement more custom data types and apply them directly to channels.

We've not yet looked at some of Go's lower-level tools for synchronization and analysis, so we'll explore `sync.atomic`, a package that—along with `sync.Mutex`—allows for more granular control over state.

Finally, we'll delve into pprof, a fabulous tool provided by Go that lets us analyze our binaries for detailed information about our goroutines, threads, overall heap, and blocking profiles.

Armed with some new tools and methods to test and analyze our code, we'll be ready to generate a robust, highly-scalable web server that can be used to safely and quickly handle any amount of traffic thrown at it.

Understanding blocking methods in Go

So far, we've encountered a few pieces of blocking code, intentional and unintentional, through our exploration and examples. At this point, it's prudent to look at the various ways we can introduce (or inadvertently fall victim to) blocking code.

By looking at the various ways Go code can be blocked, we can also be better prepared to debug cases when concurrency is not operating as expected in our application.

Blocking method 1 – a listening, waiting channel

The most concurrently-focused way to block your code is by leaving a serial channel listening to one or more goroutines. We've seen this a few times by now, but the basic concept is shown in the following code snippet:

```
func thinkAboutKeys() {
  for {
    fmt.Println("Still Thinking")
    time.Sleep(1 * time.Second)
  }
}

func main() {
  fmt.Println("Where did I leave my keys?")

  blockChannel := make(chan int)
  go thinkAboutKeys()

  <-blockChannel

  fmt.Println("OK I found them!")
}
```

Despite the fact that all of our looping code is concurrent, we're waiting on a signal for our `blockChannel` to continue linear execution. We can, of course, see this in action by sending along the channel, thus continuing code execution as shown in the following code snippet:

```
func thinkAboutKeys(bC chan int) {
  i := 0
  max := 10
  for {
    if i >= max {
      bC <- 1
    }
    fmt.Println("Still Thinking")
    time.Sleep(1 * time.Second)
    i++
  }
}
```

Here, we've modified our goroutine function to accept our blocking channel and deliver an end message to it when we've hit our maximum. These kinds of mechanisms are important for long-running processes because we may need to know when and how to kill them.

Sending more data types via channels

Go's use of channels (structs and functions) as first-class citizens provides us with a lot of interesting ways of executing, or at least trying, new approaches of communication between channels.

One such example is to create a channel that handles translation through a function itself, and instead of communicating directly through the standard syntax, the channel executes its function. You can even do this on a slice/array of functions iterating through them in the individual functions.

Creating a function channel

So far, we've almost exclusively worked in single data type and single value channels. So, let's try sending a function across a channel. With first-class channels, we need no abstraction to do this; we can just send almost anything directly over a channel as shown in the following code snippet:

```
func abstractListener(fxChan chan func() string ) {

  fxChan <- func() string {

    return "Sent!"
  }
}


func main() {

  fxChan := make (chan func() string)
  defer close(fxChan)
  go abstractListener(fxChan)
  select {
    case rfx := <- fxChan:
    msg := rfx()
    fmt.Println(msg)
    fmt.Println("Received!")

  }

}
```

This is like a callback function. However, it also is intrinsically different, as it is not just the method called after the execution of a function, but also serves as the mode of communication between functions.

Keep in mind that there are often alternatives to passing functions across channels, so this will likely be something very specific to a use case rather than a general practice.

Since your channel's type can be virtually any available type, this functionality opens up a world of possibilities, which can be potentially confusing abstractions. A struct or interface as a channel type is pretty self-explanatory, as you can make application-related decisions on any of its defined properties.

Let's see an example of using an interface in this way in the next section.

Using an interface channel

As with our function channel, being able to pass an interface (which is a complementary data type) across a channel can be incredibly useful. Let's look at an example of sending across an interface:

```
type Messenger interface {
  Relay() string
}

type Message struct {
  status string
}

func (m Message) Relay() string {
  return m.status
}

func alertMessages(v chan Messenger, i int) {
  m := new(Message)
  m.status = "Done with " + strconv.FormatInt(int64(i),10)
  v <- m
}

func main () {

  msg := make(chan Messenger)

  for i:= 0; i < 10; i++ {
    go alertMessages(msg,i)
  }

  select {
    case message := <-msg:
      fmt.Println (message.Relay())
  }
  <- msg
}
```

This is a very basic example of how to utilize interfaces as channels; in the previous example, the interface itself is largely ornamental. In actuality, we're passing newly-created message types through the interface's channel rather than interacting directly with the interface.

Using structs, interfaces, and more complex channels

Creating a custom type for our channel allows us to dictate the way our intra-channel communication will work while still letting Go dictate the context switching and behind-the-scenes scheduling.

Ultimately, this is mostly a design consideration. In the previous examples, we used individual channels for specific pieces of communication in lieu of a one-size-fits-all channel that passes a multitude of data. However, you may also find it advantageous to use a single channel to handle a large amount of communication between goroutines and other channels.

The primary consideration in deciding whether to segregate channels into individual bits of communication or a package of communications depends on the aggregate mutability of each.

For example, if you'll always want to send a counter along with a function or string and they will always be paired in terms of data consistency, such a method might make sense. If any of those components can lose synchronicity en route, it's more logical to keep each piece independent.

Maps in Go

As mentioned, maps in Go are like hash tables elsewhere and immediately related to slices or arrays.

In the previous example we were checking to see if a username/key exists already; for this purpose Go provides a simple method for doing so. When attempting to retrieve a hash with a nonexistent key, a zero value is returned, as shown in the following lines of code:

```
if Users[user.name] {
  fmt.Fprintln(conn, "Unfortunately, that username
is in
    use!");
}
```

This makes it syntactically simple and clean to test against a map and its keys.

One of the best features of maps in Go is the ability to make keys out of any comparable type, which includes strings, integers, Booleans as well as any map, struct, slice, or channel that is comprised exclusively of those types.

This one-to-many channel can work as a master-slave or broadcaster-subscriber model. We'll have a channel that listens for messages and routes them to appropriate users and a channel that listens for broadcast messages and queues them to all users.

To best demonstrate this, we'll create a simple multiuser chat system that allows Twitter style `@user` communication with a single user, with the ability to broadcast standard messages to all users and creates a universal broadcast chat note that can be read by all users. Both will be simple, custom type struct channels, so we can delineate various communication pieces.

Structs in Go

As a first-class, anonymous, and extensible type, a struct is one of the most versatile and useful data constructs available. It's simple to create analogs to other data structures such as databases and data stores, and while we hesitate to call them objects they can certainly be viewed as such.

The rule of thumb as it pertains to using structs within functions is to pass by reference rather than by value if the struct is particularly complex. Two points of clarification are as follows:

- Reference is in quotations because (and this is validated by Go's FAQ) technically everything in Go is passed by value. By that we mean that though a reference to a pointer still exists, at some step in the process the value(s) is copied.
- "Particularly complex" is, understandably, tough to quantify, so personal judgment might come into play. However, we can consider a simple struct one with no more than five methods or properties.

You can think of this in terms of a help desk system, and while in the present day we'd be unlikely to create a command-line interface for such a thing, eschewing the web portion allows us to gloss over all of the client-side code that isn't necessarily relevant to Go.

You could certainly take such an example and extrapolate it to the Web utilizing some frontend libraries for asynchronous functionality (such as `backbone.js` or `socket.io`).

To accomplish this, we'll need to create both a client and a server application, and we'll try to keep each as bare bone as possible. You can clearly and simply augment this to include any functionality you see fit such as making Git comments and updating a website.

We'll start with the server, which will be the most complicated part. The client application will mostly receive messages back through the socket, so much of the reading and routing logic will be invisible to the client-side of the process.

The net package – a chat server with interfaced channels

Here, we'll need to introduce a relevant package that will be required to handle most of the communication for our application(s). We've touched on the `net` package a bit while dabbling in the SVG output generation example to show concurrency—`net/http` is just a small part of a broader, more complex, and more feature-full package.

The basic components that we'll be using will be a TCP listener (server) and a TCP dialer (client). Let's look at the basic setup for these.

Server

Listening on a TCP port couldn't be easier. Simply initiate the `net.Listen()` method and handle the error as shown in the following lines of code:

```
listener, err := net.Listen("tcp", ":9000")
if err != nil {
  fmt.Println ("Could not start server!")
}
```

If you get an error starting the server, check your firewall or modify the port—it's possible that something is utilizing port 9000 on your system.

As easy as that is, it's just as simple on our client/dialer side.

Client

In this case, we have everything running on localhost as shown in the following lines of code. However, in a real-world application we'd probably have an intranet address used here:

```
conn, err := net.Dial("tcp","127.0.0.1:9000")
if err != nil {
  fmt.Println("Could not connect to server!")
}
```

In this application, we demonstrate two different ways to handle byte buffers of unknown lengths on `Read()`. The first is a rather crude method of trimming a string using `strings.TrimRight()`. This method allows you to define characters you aren't interested in counting as part of the input as shown in the following line of code. Mostly, it's whitespace characters that we can assume are unused parts of the buffer length.

```
sendMessage := []byte(cM.name + ": " +
  strings.TrimRight(string(buf)," \t\r\n"))
```

Dealing with strings this way is often both inelegant and unreliable. What happens if we get something we don't expect here? The string will be the length of the buffer, which in this case is 140 bytes.

The other way we deal with this is by using the end of the buffer directly. In this case, we assign the `n` variable to the `conn.Read()` function, and then can use that as a buffer length in the string to buffer conversion as shown in the following lines of code:

```
messBuff := make([]byte,1024)
n, err := conn.Read(messBuff)
if err != nil {

}
message := string(messBuff[:n])
```

Here we're taking the first `n` bytes of the message buffer's received value.

This is more reliable and efficient, but you will certainly run into text ingestion cases where you will want to remove certain characters to create cleaner input.

Each connection in this application is a struct and each user is as well. We keep track of our users by pushing them to the `Users` slice as they join.

The selected username is a command-line argument as follows:

```
./chat-client nathan
chat-client.exe nathan
```

We do not check to to ensure there is only one user with that name, so that logic might be required, particularly if chats with direct messages contain sensitive information.

Handling direct messages

For the most part, this chat client is a simple echo server, but as mentioned, we also include an ability to do non-globally broadcast messages by invoking the Twitter style `@` syntax.

We handle this mainly through regular expressions, wherein if a message matches `@user` then only that user will see the message; otherwise, it's broadcasted to all. This is somewhat inelegant, because senders of the direct message will not see their own direct message if their usernames do not match the intended names of the users.

To do this, we direct every message through a `evalMessageRecipient()` function before broadcasting. As this is relying on user input to create the regular expression (in the form of the username), please take note that we should escape this with the `regexp.QuoteMeta()` method to prevent regex failures.

Let's first examine our chat server, which is responsible for maintaining all connections and passing them to goroutines to listen and receive, as shown in the following code:

```
chat-server.go
package main

import
(
  "fmt"
  "strings"
  "net"
  "strconv"
  "regexp"
)

var connectionCount int
var messagePool chan(string)

const (
  INPUT_BUFFER_LENGTH = 140
)
```

We utilize a maximum character buffer. This restricts our chat messages to no more than 140 characters. Let's look at our `User` struct to see the information we might keep about a user that joins, as follows:

```
type User struct {
  Name string
  ID int
  Initiated bool
```

The initiated variable tells us that `User` is connected after a connection and announcement. Let's examine the following code to understand the way we'd listen on a channel for a logged-in user:

```
    UChannel chan []byte
    Connection *net.Conn
  }
  The User struct contains all of the information we will maintain
    for each connection. Keep in mind here we don't do any sanity
    checking to make sure a user doesn't exist - this doesn't
    necessarily pose a problem in an example, but a real chat client
    would benefit from a response should a user name already be
    in use.

  func (u *User) Listen() {
    fmt.Println("Listening for",u.Name)
    for {
      select {
        case msg := <- u.UChannel:
          fmt.Println("Sending new message to",u.Name)
          fmt.Fprintln(*u.Connection,string(msg))

      }
    }
  }
```

This is the core of our server: each `User` gets its own `Listen()` method, which maintains the `User` struct's channel and sends and receives messages across it. Put simply, each user gets a concurrent channel of his or her own. Let's take a look at the `ConnectionManager` struct and the `Initiate()` function that creates our server in the following code:

```
  type ConnectionManager struct {
    name      string
    initiated bool
  }

  func Initiate() *ConnectionManager {
    cM := &ConnectionManager{
      name:      "Chat Server 1.0",
      initiated: false,
    }

    return cM
  }
```

Our `ConnectionManager` struct is initiated just once. This sets some relatively ornamental attributes, some of which could be returned on request or on chat login. We'll examine the `evalMessageRecipient` function that attempts to roughly identify the intended recipient of any message sent as follows:

```
func evalMessageRecipient(msg []byte, uName string) bool {
  eval := true
  expression := "@"
  re, err := regexp.MatchString(expression, string(msg))
  if err != nil {
    fmt.Println("Error:", err)
  }
  if re == true {
    eval = false
    pmExpression := "@" + uName
    pmRe, pmErr := regexp.MatchString(pmExpression, string(msg))
    if pmErr != nil {
      fmt.Println("Regex error", err)
    }
    if pmRe == true {
      eval = true
    }
  }
  return eval
}
```

This is our router of sorts taking the `@` part of the string and using it to detect an intended recipient to hide from public consumption. We do not return an error if the user doesn't exist or has left the chat.

The format for regular expressions using the `regexp` package relies on the `re2` syntax, which is described at `https://code.google.com/p/re2/wiki/Syntax`.

Let's take a look at the code for the `Listen()` method of the `ConnectionManager` struct:

```
func (cM *ConnectionManager) Listen(listener net.Listener) {
  fmt.Println(cM.name, "Started")
  for {

    conn, err := listener.Accept()
    if err != nil {
      fmt.Println("Connection error", err)
    }
```

```
      connectionCount++
      fmt.Println(conn.RemoteAddr(), "connected")
      user := User{Name: "anonymous", ID: 0, Initiated: false}
      Users = append(Users, &user)
      for _, u := range Users {
        fmt.Println("User online", u.Name)
      }
      fmt.Println(connectionCount, "connections active")
      go cM.messageReady(conn, &user)
    }
  }

  func (cM *ConnectionManager) messageReady(conn net.Conn, user
    *User) {
    uChan := make(chan []byte)

    for {

      buf := make([]byte, INPUT_BUFFER_LENGTH)
      n, err := conn.Read(buf)
      if err != nil {
        conn.Close()
        conn = nil
      }
      if n == 0 {
        conn.Close()
        conn = nil
      }
      fmt.Println(n, "character message from user", user.Name)
      if user.Initiated == false {
        fmt.Println("New User is", string(buf))
        user.Initiated = true
        user.UChannel = uChan
        user.Name = string(buf[:n])
        user.Connection = &conn
        go user.Listen()

        minusYouCount := strconv.FormatInt(int64(connectionCount-1),
          10)
        conn.Write([]byte("Welcome to the chat, " + user.Name + ",
          there are " + minusYouCount + " other users"))

      } else {
        sendMessage := []byte(user.Name + ": " +
          strings.TrimRight(string(buf), " \t\r\n"))

        for _, u := range Users {
          if evalMessageRecipient(sendMessage, u.Name) == true {
            u.UChannel <- sendMessage
          }

        }

      }

    }
  }geReady (per connectionManager) function instantiates new
    connections into a User struct, utilizing first sent message as
    the user's name.

  var Users []*User
  This is our unbuffered array (or slice) of user structs.
  func main() {
    connectionCount = 0
    serverClosed := make(chan bool)

    listener, err := net.Listen("tcp", ":9000")
    if err != nil {
      fmt.Println ("Could not start server!",err)
    }

    connManage := Initiate()
    go connManage.Listen(listener)

    <-serverClosed
  }
```

As expected, `main()` primarily handles the connection and error and keeps our server open and nonblocked with the `serverClosed` channel.

There are a number of methods we could employ to improve the way we route messages. The first method would be to invoke a map (or hash table) bound to a username. If the map's key exists, we could return some error functionality if a user already exists, as shown in the following code snippet:

```
type User struct {
  name string
}
var Users map[string] *User

func main() {
  Users := make(map[string] *User)
}
```

Examining our client

Our client application is a bit simpler primarily because we don't care as much about blocking code.

While we do have two concurrent operations (wait for the message and wait for user input to send the message), this is significantly less complicated than our server, which needs to concurrently listen to each created user and distribute sent messages, respectively.

Let's now compare our chat client to our chat server. Obviously, the client has less overall maintenance of connections and users, and so we do not need to use nearly as many channels. Let's take a look at our chat client's code:

```
chat-client.go
package main

import
(
  "fmt"
  "net"
  "os"
  "bufio"
  "strings"
)
```

```
type Message struct {
  message string
  user string
}

var recvBuffer [140]byte

func listen(conn net.Conn) {
  for {

      messBuff := make([]byte,1024)
      n, err := conn.Read(messBuff)
      if err != nil {
        fmt.Println("Read error",err)
      }
      message := string(messBuff[:n])
      message = message[0:]

      fmt.Println(strings.TrimSpace(message))
      fmt.Print("> ")
  }

}

func talk(conn net.Conn, mS chan Message) {

      for {
      command := bufio.NewReader(os.Stdin)
        fmt.Print("> ")
                line, err := command.ReadString('\n')

                line = strings.TrimRight(line, " \t\r\n")
        _, err = conn.Write([]byte(line))
                if err != nil {
                        conn.Close()
                        break

                }
      doNothing(command)
        }

}
```

```
func doNothing(bf *bufio.Reader) {
  // A temporary placeholder to address io reader usage


}
func main() {

  messageServer := make(chan Message)

  userName := os.Args[1]

  fmt.Println("Connecting to host as",userName)

  clientClosed := make(chan bool)

  conn, err := net.Dial("tcp","127.0.0.1:9000")
  if err != nil {
    fmt.Println("Could not connect to server!")
  }
  conn.Write([]byte(userName))
  introBuff := make([]byte,1024)
  n, err := conn.Read(introBuff)
  if err != nil {

  }
  message := string(introBuff[:n])
  fmt.Println(message)

  go talk(conn,messageServer)
  go listen(conn)

  <- clientClosed
}
```

Blocking method 2 – the select statement in a loop

Have you noticed yet that the `select` statement itself blocks? Fundamentally, the `select` statement is not different from an open listening channel; it's just wrapped in conditional code.

The `<- myChannel` channel operates the same way as the following code snippet:

```
select {
  case mc := <- myChannel:
    // do something
}
```

An open listening channel is not a deadlock as long as there are no goroutines sleeping. You'll find this on channels that are listening but will never receive anything, which is another method of basically waiting.

These are useful shortcuts for long-running applications you wish to keep alive but you may not necessarily need to send anything along that channel.

Cleaning up goroutines

Any channel that is left waiting and/or left receiving will result in a deadlock. Luckily, Go is pretty adept at recognizing these and you will almost without fail end up in a panic when running or building the application.

Many of our examples so far have utilized the deferred `close()` method of immediately and cleanly grouping together similar pieces of code that should execute at different points.

While garbage collection handles a lot of the cleanup, we're largely left to take care of open channels to ensure we don't have a process waiting to receive and/or something waiting to send, both waiting at the same time for each other. Luckily, we'll be unable to compile any such program with a detectable deadlock condition, but we also need to manage closing channels that are left waiting.

Quite a few of the examples so far have ended with a generic integer or Boolean channel that just waits—this is employed almost exclusively for the channel's blocking effect and allows us to demonstrate the effects and output of concurrent code while the application is still running. In many cases, this generic channel is an unnecessary bit of syntactical cruft as shown in the following lines of code:

```
<-youMayNotNeedToDoThis
close(youmayNotNeedToDoThis)
```

The fact that there's no assignment happening is a good indicator this is an example of such cruft. If we had instead modified that to include an assignment, the previous code would be changed to the following instead:

```
v := <-youMayNotNeedToDoThis
```

It might indicate that the value is useful and not just arbitrary blocking code.

Blocking method 3 – network connections and reads

If you run the code from our earlier chat server's client without starting the server, you'll notice that the `Dial` function blocks any subsequent goroutine. We can test this by imposing a longer-than-normal timeout on the connection or by simply closing the client application after logging in, as we did not implement a method for closing the TCP connection.

As the network reader we're using for the connection is buffered, we'll always have a blocking mechanism while waiting for data via TCP.

Creating channels of channels

The preferred and sanctioned way of managing concurrency and state is exclusively through channels.

We've demonstrated a few more complex types of channels, but we haven't looked at what can become a daunting but powerful implementation: channels of channels. This might at first sound like some unmanageable wormhole, but in some situations we want a concurrent action to generate more concurrent actions; thus, our goroutines should be capable of spawning their own.

As always, the way you manage this is through design while the actual code may simply be an aesthetic byproduct here. Building an application this way should make your code more concise and clean most of the time.

Let's revisit a previous example of an RSS feed reader to demonstrate how we could manage this, as shown in the following code:

```
package main

import (
 "fmt"
)

type master chan Item

var feedChannel chan master
var done chan bool
```

```
type Item struct {
 Url  string
 Data []byte
}
type Feed struct {
 Url   string
 Name  string
 Items []Item
}

var Feeds []Feed

func process(feedChannel *chan master, done *chan bool) {
 for _, i := range Feeds {
  fmt.Println("feed", i)
  item := Item{}
  item.Url = i.Url
  itemChannel := make(chan Item)
  *feedChannel <- itemChannel
  itemChannel <- item
 }
 *done <- true
}
func processItem(url string) {
 // deal with individual feed items here
 fmt.Println("Got url", url)
}

func main() {
 done := make(chan bool)
 Feeds = []Feed{Feed{Name: "New York Times", Url: "http://rss.nytimes.
com/services/xml/rss/nyt/HomePage.xml"},
  Feed{Name: "Wall Street Journal", Url: "http://feeds.wsjonline.com/
wsj/xml/rss/3_7011.xml"}}
 feedChannel := make(chan master)
 go func(done chan bool, feedChannel chan master) {
  for {
   select {
   case fc := <-feedChannel:
    select {
    case item := <-fc:
     processItem(item.Url)
    }
```

```
        default:
        }
      }
    }(done, feedChannel)
    go process(&feedChannel, &done)
    <-done
    fmt.Println("Done!")
  }
```

Here, we manage `feedChannel` as a custom struct that is itself a channel for our `Item` type. This allows us to rely exclusively on channels for synchronization handled through a semaphore-esque construct.

If we want to look at another way of handling a lower-level synchronization, `sync.atomic` provides some simple iterative patterns that allow you to manage synchronization directly in memory.

As per Go's documentation, these operations require great care and are prone to data consistency errors, but if you need to touch memory directly, this is the way to do it. When we talk about advanced concurrency features, we'll utilize this package directly.

Pprof – yet another awesome tool

Just when you think you've seen the entire spectrum of Go's amazing tool set, there's always one more utility that, once you realize it exists, you'll wonder how you ever survived without it.

Go format is great for cleaning up your code; the `-race` flag is essential for detecting possible race conditions, but an even more robust, hands-in-the-dirt tool exists that is used to analyze your final application, and that is pprof.

Google created pprof initially to analyze loop structures and memory allocation (and related types) for C++ applications.

It's particularly useful if you think you have performance issues not uncovered by the testing tools provided in the Go runtime. It's also a fantastic way to generate a visual representation of the data structures in any application.

Some of this functionality also exists as part of the Go testing package and its benchmarking tools—we'll explore that more in *Chapter 7, Performance and Scalability*.

Getting the runtime version of pprof to work requires a few pieces of setup first. We'll need to include the `runtime.pprof` package and the `flag` package, which allows command-line parsing (in this case, for the output of pprof).

If we take our chat server code, we can add a couple of lines and have the application prepped for performance profiling.

Let's make sure we include those two packages along with our other packages. We can use the underscore syntax to indicate to the compiler that we're only interested in the package's side effects (meaning we get the package's initialization functions and global variables) as shown in the following lines of code:

```
import
(
  "fmt"
...
  _ "runtime/pprof"
)
```

Next, in our `main()` function, we include a flag parser that will parse and interpret the data produced by pprof as well as create the CPU profile itself if it does not exist (and bailing if it cannot be created), as shown in the following code snippet:

```
var profile = flag.String("cpuprofile", "", "output pprof data to
  file")

func main() {
  flag.Parse()
  if *profile != "" {
    flag,err := os.Create(*profile)
    if err != nil {
      fmt.Println("Could not create profile",err)
    }
    pprof.StartCPUProfile(flag)
    defer pprof.StopCPUProfile()

  }
}
```

This tells our application to generate a CPU profiler if it does not exist, start the profiling at the beginning of the execution, and defer the end of the profiling until the application exits successfully.

With this created, we can run our binary with the `cpuprofile` flag, which tells the program to generate a profile file as follows:

```
./chat-server -cpuprofile=chat.prof
```

For the sake of variety (and exploiting more resources arbitrarily), we'll abandon the chat server for a moment and create a loop generating scores of goroutines before exiting. This should give us a more exciting demonstration of profiling data than a simple and long-living chat server would, although we'll return to that briefly:

Here is our example code that generates more detailed and interesting profiling data:

```
package main

import (
  "flag"
  "fmt"
  "math/rand"
  "os"
  "runtime"
  "runtime/pprof"
)

const ITERATIONS = 99999
const STRINGLENGTH = 300

var profile = flag.String("cpuprofile", "", "output pprof data to
  file")

func generateString(length int, seed *rand.Rand, chHater chan
  string) string {
  bytes := make([]byte, length)
  for i := 0; i < length; i++ {
    bytes[i] = byte(rand.Int())
  }
  chHater <- string(bytes[:length])
  return string(bytes[:length])
}

func generateChannel() <-chan int {
  ch := make(chan int)
  return ch
}

func main() {

  goodbye := make(chan bool, ITERATIONS)
  channelThatHatesLetters := make(chan string)

  runtime.GOMAXPROCS(2)
  flag.Parse()
  if *profile != "" {
    flag, err := os.Create(*profile)
    if err != nil {
      fmt.Println("Could not create profile", err)
    }
    pprof.StartCPUProfile(flag)
    defer pprof.StopCPUProfile()

  }
```

```
    seed := rand.New(rand.NewSource(19))

    initString := ""

    for i := 0; i < ITERATIONS; i++ {
      go func() {
        initString = generateString(STRINGLENGTH, seed,
          channelThatHatesLetters)
        goodbye <- true
      }()

    }
    select {
    case <-channelThatHatesLetters:

    }
    <-goodbye

    fmt.Println(initString)

  }
```

When we generate a profile file out of this, we can run the following command:

`go tool pprof chat-server chat-server.prof`

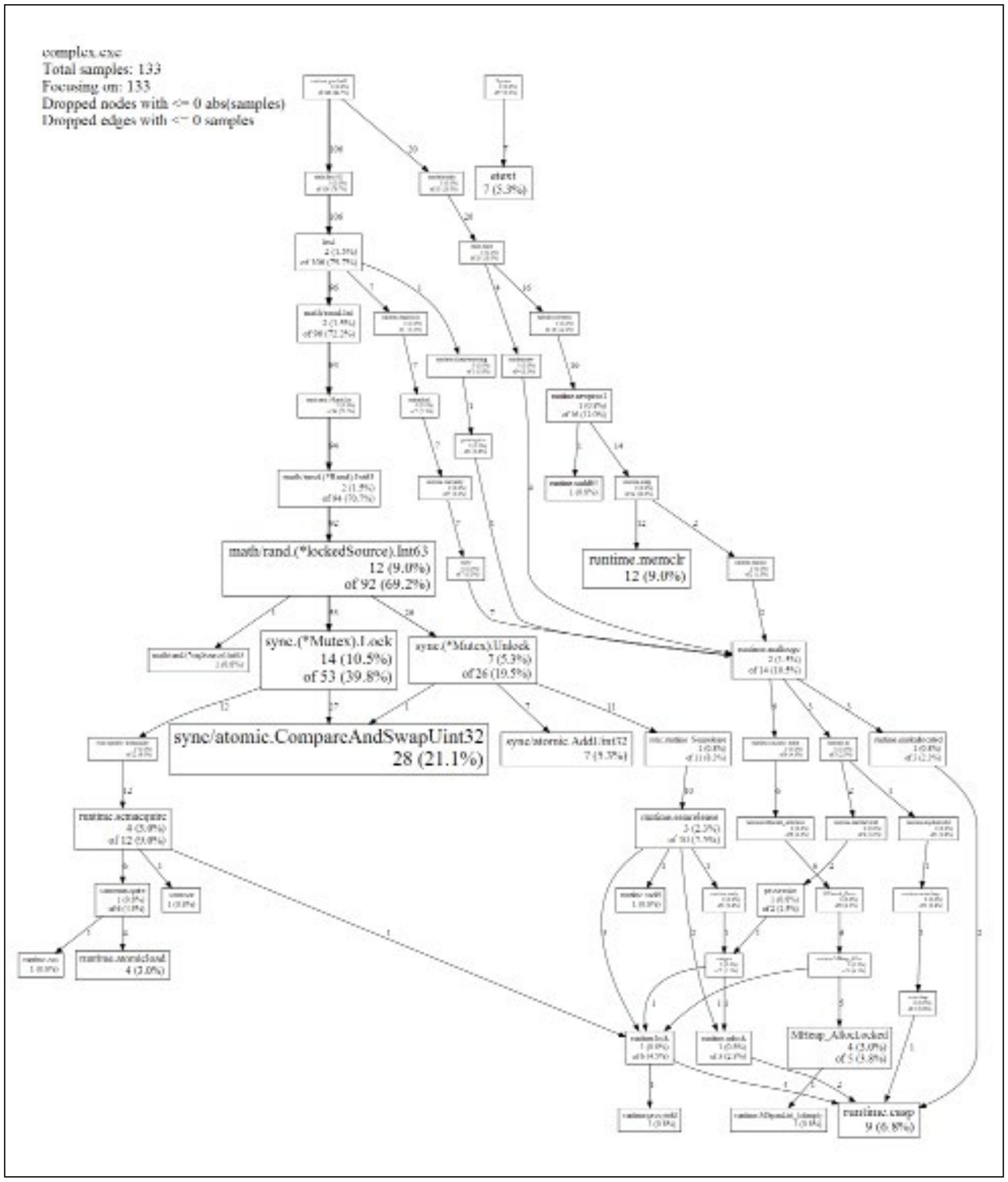
This will start the pprof application itself. This gives us a few commands that report on the static, generated file as follows:

- `topN`: This shows the top *N* samples from the profile file, where *N* represents the explicit number you want to see.
- `web`: This creates a visualization of data, exports it to SVG, and opens it in a web browser. To get the SVG output, you'll need to install Graphviz as well (`http://www.graphviz.org/`).

You can also run pprof with some flags directly to output in several formats or launch a browser as follows:

- `--text`: This generates a text report
- `--web`: This generates an SVG and opens in the browser
- `--gv`: This generates the Ghostview postscript
- `--pdf`: This generates the PDF to output
- `--SVG`: This generates the SVG to output
- `--gif`: This generates the GIF to output

The command-line results will be telling enough, but it's especially interesting to see the blocking profile of your application presented in a descriptive, visual way as shown in the following figure. When you're in the pprof tool, just type in `web` and a browser will spawn with the CPU profiling detailed in SVG form.

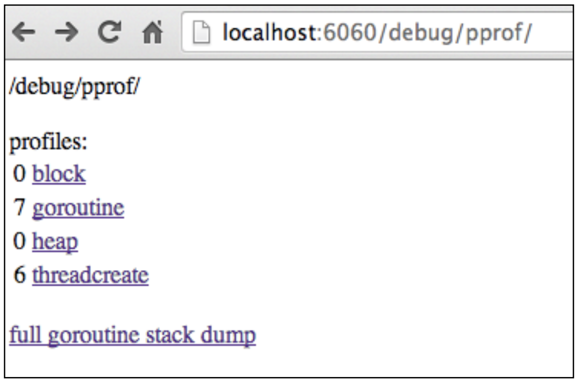


The idea here is less about the text and more about the complexity

And voila, we suddenly have an insight into how our program utilizes the CPU time consumption and a general view of how our application executes, loops, and exits.

In typical Go fashion, the pprof tool also exists in the `net/http` package, although it's more data-centric than visual. This means that rather than dealing exclusively with a command-line tool, you can output the results directly to the Web for analysis.

Like the command-line tool, you'll see block, goroutine, heap, and thread profiles as well as a full stack outline directly through localhost, as shown in the following screenshot:


localhost:6060/debug/pprof/

/debug/pprof/

profiles:
0 block
7 goroutine
0 heap
6 threadcreate

full goroutine stack dump

To generate this server, you just need to include a few key lines of code in your application, build it, and then run it. For this example, we've included the code in our chat server application, which allows us to get the Web view of an otherwise command-line-only application.

Make sure you have the `net/http` and `log` packages included. You'll also need the `http/pprof` package. The code snippet is as follows:

```
import(_(_
  "net/http/pprof"
  "log"
  "net/http"
)
```

Then simply include this code somewhere in your application, ideally, near the top of the `main()` function, as follows:

```
  go func() {
    log.Println(http.ListenAndServe("localhost:6060", nil))
  }()
```

As always, the port is entirely a matter of preference.

You can then find a number of profiling tools at `localhost:6060`, including the following:

- All tools can be found at `http://localhost:6060/debug/pprof/`
- Blocking profiles cab be found at `http://localhost:6060/debug/pprof/block?debug=1`
- A profile of all goroutines can be found at `http://localhost:6060/debug/pprof/goroutine?debug=1`
- A detailed profile of the heap can be found at `http://localhost:6060/debug/pprof/heap?debug=1`
- A profile of threads created can be found at `http://localhost:6060/debug/pprof/threadcreate?debug=1`

In addition to the blocking profile, you may find a utility to track down inefficiency in your concurrent strategy through the thread creation profile. If you find a seemingly abnormal amount of threads created, you can toy with the synchronization structure as well as runtime parameters to streamline this.

Keep in mind that using pprof this way will also include some analyses and profiles that can be attributed to the `http` or `pprof` packages rather than your core code. You will find certain lines that are quite obviously not part of your application; for example, a thread creation analysis of our chat server includes a few telling lines, as follows:

```
#       0x7765e         net/http.HandlerFunc.ServeHTTP+0x3e     /usr/
local/go/src/pkg/net/http/server.go:1149
#       0x7896d         net/http.(*ServeMux).ServeHTTP+0x11d /usr/
local/go/src/pkg/net/http/server.go:1416
```

Given that we specifically eschewed delivering our chat application via HTTP or web sockets in this iteration, this should be fairly evident.

On top of that, there are even more obvious smoking guns, as follows:

```
#       0x139541        runtime/pprof.writeHeap+0x731           /usr/
local/go/src/pkg/runtime/pprof/pprof.go:447
#       0x137aa2        runtime/pprof.(*Profile).WriteTo+0xb2   /usr/
local/go/src/pkg/runtime/pprof/pprof.go:229
#       0x9f55f         net/http/pprof.handler.ServeHTTP+0x23f  /usr/
local/go/src/pkg/net/http/pprof/pprof.go:165
#       0x9f6a5         net/http/pprof.Index+0x135              /usr/
local/go/src/pkg/net/http/pprof/pprof.go:177
```

Some system and Go core mechanisms we will never be able to reduce out of our final compiled binaries are as follows:

```
#       0x18d96 runtime.starttheworld+0x126
  /usr/local/go/src/pkg/runtime/proc.c:451
```

The hexadecimal value represents the address in the memory of the function when run.

A note for Windows users: pprof is a breeze to use in *nix environments but may take some more arduous tweaking under Windows. Specifically, you may need a bash replacement such as Cygwin. You may also find some necessary tweaks to pprof itself (in actuality, a Perl script) may be in order. For 64-bit Windows users, make sure you install ActivePerl and execute the pprof Perl script directly using the 64-bit version of Perl.

At publish time, there are also some issues running this on 64-bit OSX.

Handling deadlocks and errors

Anytime you encounter a deadlock error upon compilation in your code, you'll see the familiar string of semi-cryptic errors explaining which goroutine was left holding the bag, so to speak.

However, keep in mind you always have the ability to invoke your own panic using Go's built-in panic, and this can be incredibly useful for building your own error-catching safeguards to ensure data consistency and ideal operation. The code is as follows:

```
package main

import
(
  "os"
)

func main() {
  panic("Oh No, we forgot to write a program!")
  os.Exit(1)
}
```

This can be utilized anywhere you wish to give detailed exit information to either developers or end users.

Summary

Having explored some new ways to examine the way that Go code can block and deadlock, we also have some tools at our disposal that can be used to examine CPU profiles and resource usage now.

Hopefully, by this point, you can build some complex concurrent systems with simple goroutines and channels all the way up to multiplexed channels of structs, interfaces, and other channels.

We've built some somewhat-functional applications so far, but next we're going to utilize everything we've done to build a usable web server that solves a classic problem and can be used to design intranets, file storage systems, and more.

In the next chapter, we'll take what we've done in this chapter with regard to extensible channels and apply it to solving one of the oldest challenges the Internet has to offer: concurrently serving 10,000 (or more) connections.

6
C10K – A Non-blocking Web Server in Go

Up to this point, we've built a few usable applications; things we can start with and leapfrog into real systems for everyday use. By doing so, we've been able to demonstrate the basic and intermediate-level patterns involved in Go's concurrent syntax and methodology.

However, it's about time we take on a real-world problem—one that has vexed developers (and their managers and VPs) for a great deal of the early history of the Web.

In addressing and, hopefully, solving this problem, we'll be able to develop a high-performance web server that can handle a very large volume of live, active traffic.

For many years, the solution to this problem was solely to throw hardware or intrusive caching systems at the problem; so, alternately, solving it with programming methodology should excite any programmer.

We'll be using every technique and language construct we've learned so far, but we'll do so in a more structured and deliberate way than we have up to now. Everything we've explored so far will come into play, including the following points:

- Creating a visual representation of our concurrent application
- Utilizing goroutines to handle requests in a way that will scale
- Building robust channels to manage communication between goroutines and the loop that will manage them
- Profiling and benchmarking tools (JMeter, ab) to examine the way our event loop actually works
- Timeouts and concurrency controls—when necessary—to ensure data and request consistency

Attacking the C10K problem

The genesis of the C10K problem is rooted in serial, blocking programming, which makes it ideal to demonstrate the strength of concurrent programming, especially in Go.

The proposed problem came from developer Dan Kegel, who famously asked:

> *It's time for web servers to handle ten thousand clients simultaneously, don't you think? After all, the web is a big place now.*
>
> *- Dan Kegel (http://www.kegel.com/c10k.html)*

When he asked this in 1999, for many server admins and engineers, serving 10,000 concurrent visitors was something that would be solved with hardware. The notion that a single server on common hardware could handle this type of CPU and network bandwidth without falling over seemed foreign to most.

The crux of his proposed solutions relied on producing non-blocking code. Of course, in 1999, concurrency patterns and libraries were not widespread. C++ had some polling and queuing options available via some third-party libraries and the earliest predecessor to multithreaded syntaxes, later available through Boost and then C++11.

Over the coming years, solutions to the problem began pouring in across various flavors of languages, programming design, and general approaches. At the time of publishing this book, the C10K problem is not one without solutions, but it is still an excellent platform to conduct a very real-world challenge to high-performance Go.

Any performance and scalability problem will ultimately be bound to the underlying hardware, so as always, your mileage may vary. Squeezing 10,000 concurrent connections on a 486 processor with 500 MB of RAM will certainly be more challenging than doing so on a barebones Linux server stacked with memory and multiple cores.

It's also worth noting that a simple echo server would obviously be able to assume more cores than a functional web server that returns larger amounts of data and accepts greater complexity in requests, sessions, and so on, as we'll be dealing with here.

Failing of servers at 10,000 concurrent connections

As you may recall, when we discussed concurrent strategies back in *Chapter 3*, *Developing a Concurrent Strategy*, we talked a bit about Apache and its load-balancing tools.

When the Web was born and the Internet commercialized, the level of interactivity was pretty minimal. If you're a graybeard, you may recall the transition from NNTP/IRC and the like and how extraordinarily rudimentary the Web was.

To address the basic proposition of [page request] → [HTTP response], the requirements on a web server in the early 1990s were pretty lenient. Ignoring all of the error responses, header readings and settings, and other essential (but unrelated to the in → out mechanism) functions, the essence of the early servers was shockingly simple, at least compared to the modern web servers.

> The first web server was developed by the father of the Web, Tim Berners-Lee.
>
> Developed at CERN (such as WWW/HTTP itself), CERN httpd handled many of the things you would expect in a web server today—hunting through the code, you'll find a lot of notation that will remind you that the very core of the HTTP protocol is largely unchanged. Unlike most technologies, HTTP has had an extraordinarily long shelf life.
>
> Written in C in 1990, it was unable to utilize a lot of concurrency strategies available in languages such as Erlang. Frankly, doing so was probably unnecessary—the majority of web traffic was a matter of basic file retrieval and protocol. The meat and potatoes of a web server were not dealing with traffic, but rather dealing with the rules surrounding the protocol itself.
>
> You can still access the original CERN httpd site and download the source code for yourself from `http://www.w3.org/Daemon/`. I highly recommend that you do so as both a history lesson and a way to look at the way the earliest web server addressed some of the earliest problems.

However, the Web in 1990 and the Web when the C10K question was first posed were two very different environments.

By 1999, most sites had some level of secondary or tertiary latency provided by third-party software, CGI, databases, and so on, all of which further complicated the matter. The notion of serving 10,000 flat files concurrently is a challenge in itself, but try doing so by running them on top of a Perl script that accesses a MySQL database without any caching layer; the challenge is immediately exacerbated.

By the mid 1990s, the Apache web server had taken hold and largely controlled the market (by 2009, it had become the first server software to serve more than 100 million websites).

Apache's approach was rooted heavily in the earliest days of the Internet. At its launch, connections were initially handled first in, first out. Soon, each connection was assigned a thread from the thread pool. There are two problems with the Apache server. They are as follows:

- Blocking connections can lead to a domino effect, wherein one or more slowly resolved connections could avalanche into inaccessibility
- Apache had hard limits on the number of threads/workers you could utilize, irrespective of hardware constraints

It's easy to see the opportunity here, at least in retrospect. A concurrent server that utilizes actors (Erlang), agents (Clojure), or goroutines (Go) seems to fit the bill perfectly. Concurrency does not *solve* the C10k problem in itself, but it absolutely provides a methodology to facilitate it.

The most notable and visible example of an approach to the C10K problem today is Nginx, which was developed using concurrency patterns, widely available in C by 2002 to address—and ultimately solve—the C10k problem. Nginx, today, represents either the #2 or #3 web server in the world, depending on the source.

Using concurrency to attack C10K

There are two primary approaches to handle a large volume of concurrent requests. The first involves allocating threads per connection. This is what Apache (and a few others) do.

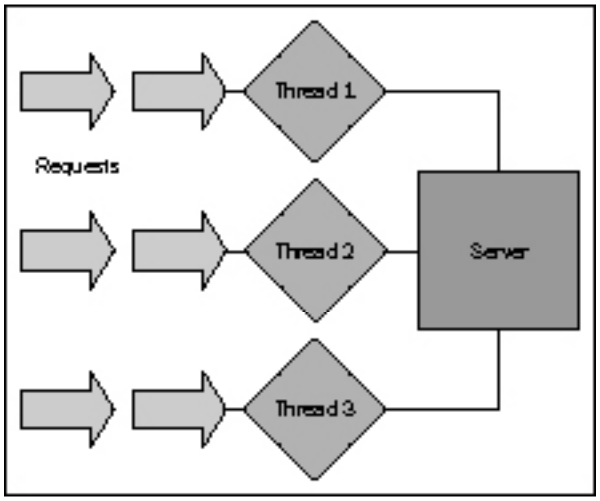
On the one hand, allocating a thread to a connection makes a lot of sense—it's isolated, controllable via the application's and kernel's context switching, and can scale with increased hardware.

One problem for Linux servers—on which the majority of the Web lives—is that each allocated thread reserves 8 MB of memory for its stack by default. This can (and should) be redefined, but this imposes a largely unattainable amount of memory required for a single server. Even if you set the default stack size to 1 MB, we're dealing with a minimum of 10 GB of memory just to handle the overhead.

This is an extreme example that's unlikely to be a real issue for a couple of reasons: first, because you can dictate the maximum amount of resources available to each thread, and second, because you can just as easily load balance across a few servers and instances rather than add 10 GB to 80 GB of RAM.

Even in a threaded server environment, we're fundamentally bound to the issue that can lead to performance decreases (to the point of a crash).

First, let's look at a server with connections bound to threads (as shown in the following diagram), and visualize how this can lead to logjams and, eventually, crashes:



This is obviously what we want to avoid. Any I/O, network, or external process that can impose some slowdown can bring about that avalanche effect we talked about, such that our available threads are taken (or backlogged) and incoming requests begin to stack up.

We can spawn more threads in this model, but as mentioned earlier, there are potential risks there too, and even this will fail to mitigate the underlying problem.

Taking another approach

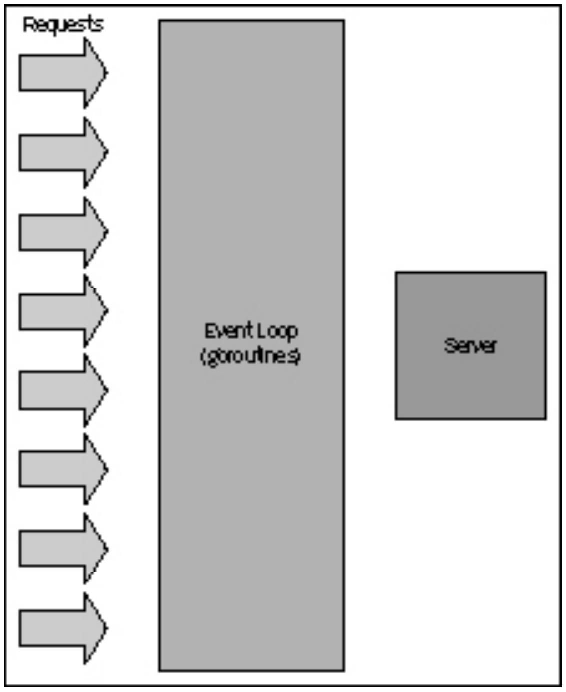
In an attempt to create our web server that can handle 10,000 concurrent connections, we'll obviously leverage our goroutine/channel mechanism to put an event loop in front of our content delivery to keep new channels recycled or created constantly.

For this example, we'll assume we're building a corporate website and infrastructure for a rapidly expanding company. To do this, we'll need to be able to serve both static and dynamic content.

The reason we want to introduce dynamic content is not just for the purposes of demonstration—we want to challenge ourselves to show 10,000 true concurrent connections even when a secondary process gets in the way.

As always, we'll attempt to map our concurrency strategy directly to goroutines and channels. In a lot of other languages and applications, this is directly analogous to an event loop, and we'll approach it as such. Within our loop, we'll manage the available goroutines, expire or reuse completed ones, and spawn new ones where necessary.

In this example visualization, we show how an event loop (and corresponding goroutines) can allow us to scale our connections without employing too many *hard* resources such as CPU threads or RAM:



The most important step for us here is to manage that event loop. We'll want to create an open, infinite loop to manage the creation and expiration of our goroutines and respective channels.

As part of this, we will also want to do some internal logging of what's happening, both for benchmarking and debugging our application.

Building our C10K web server

Our web server will be responsible for taking requests, routing them, and serving either flat files or dynamic files with templates parsed against a few different data sources.

As mentioned earlier, if we exclusively serve flat files and remove much of the processing and network latency, we'd have a much easier time with handling 10,000 concurrent connections.

Our goal is to approach as much of a real-world scenario as we can—very little of the Web operates on a single server in a static fashion. Most websites and applications utilize databases, **CDNs** (**Content Delivery Networks**), dynamic and uncached template parsing, and so on. We need to replicate them whenever possible.

For the sake of simplicity, we'll separate our content by type and filter them through URL routing, as follows:

- `/static/[request]`: This will serve `request.html` directly
- `/template/[request]`: This will serve `request.tpl` after its been parsed through Go
- `/dynamic/[request][number]`: This will also serve `request.tpl` and parse it against a database source's record

By doing this, we should get a better mixture of possible HTTP request types that could impede the ability to serve large numbers of users simultaneously, especially in a blocking web server environment.

We'll utilize the `html/template` package to do parsing—we've briefly looked at the syntax before, and going any deeper is not necessarily part of the goals of this book. However, you should look into it if you're going to parlay this example into something you use in your environment or have any interest in building a framework.

You can find Go's exceptional library to generate safe data-driven templating at `http://golang.org/pkg/html/template/`.

By safe, we're largely referring to the ability to accept data and move it directly into templates without worrying about the sort of injection issues that are behind a large amount of malware and cross-site scripting.

For the database source, we'll use MySQL here, but feel free to experiment with other databases if you're more comfortable with them. Like the `html/template` package, we're not going to put a lot of time into outlining MySQL and/or its variants.

Benchmarking against a blocking web server

It's only fair to add some starting benchmarks against a blocking web server first so that we can measure the effect of concurrent versus nonconcurrent architecture.

For our starting benchmarks, we'll eschew any framework, and we'll go with our old stalwart, Apache.

For the sake of completeness here, we'll be using an Intel i5 3GHz machine with 8 GB of RAM. While we'll benchmark our final product on Ubuntu, Windows, and OS X here, we'll focus on Ubuntu for our example.

Our localhost domain will have three plain HTML files in `/static`, each trimmed to 80 KB. As we're not using a framework, we don't need to worry about raw dynamic requests, but only about static and dynamic requests in addition to data source requests.

For all examples, we'll use a MySQL database (named `master`) with a table called `articles` that will contain 10,000 duplicate entries. Our structure is as follows:

```
CREATE TABLE articles (
  article_id INT NOT NULL AUTO_INCREMENT,
  article_title VARCHAR(128) NOT NULL,
  article_text VARCHAR(128) NOT NULL,
  PRIMARY KEY (article_id)
)
```

With ID indexes ranging sequentially from 0-10,000, we'll be able to generate random number requests, but for now, we just want to see what kind of basic response we can get out of Apache serving static pages with this machine.

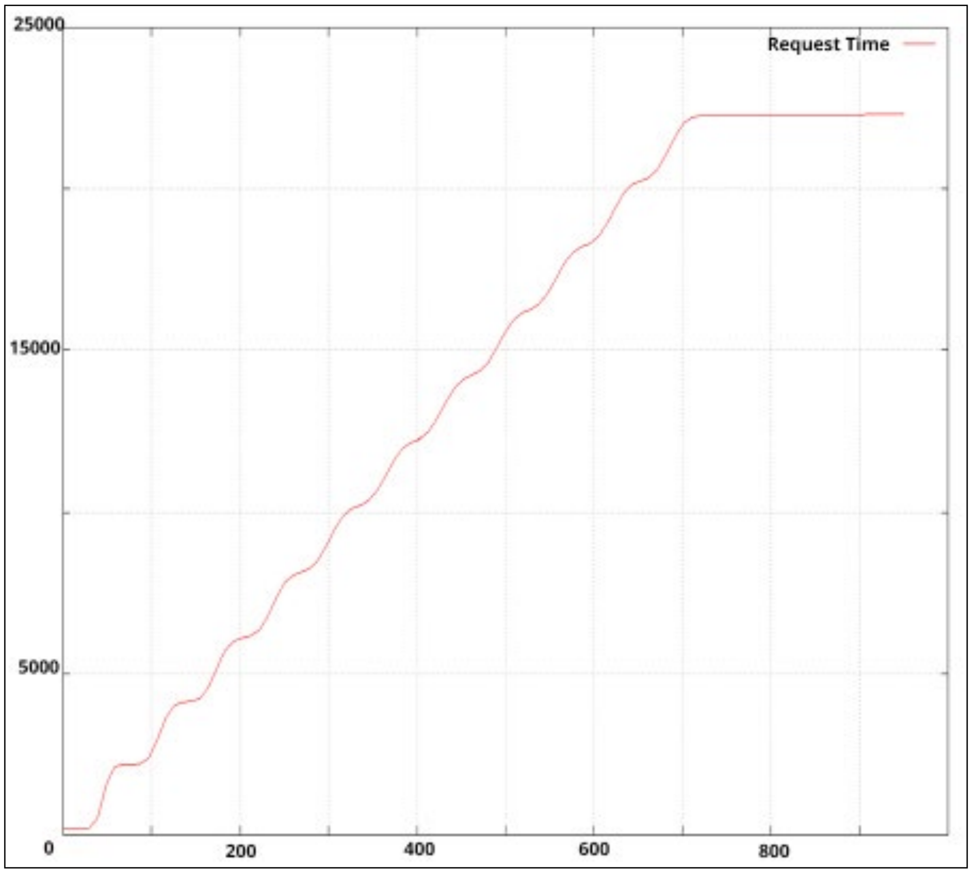
For this test, we'll use Apache's ab tool and then gnuplot to sequentially map the request time as the number of concurrent requests and pages; we'll do this for our final product as well, but we'll also go through a few other benchmarking tools for it to get some better details.

Apache's AB comes with the Apache web server itself. You can read more about it at `http://httpd.apache.org/docs/2.2/programs/ab.html`.

You can download it for Linux, Windows, OS X, and more from `http://httpd.apache.org/download.cgi`.

The gnuplot utility is available for the same operating systems at `http://www.gnuplot.info/`.

So, let's see how we did it. Have a look at the following graph:



Ouch! Not even close. There are things we can do to tune the connections available (and respective threads/workers) within Apache, but this is not really our goal. Mostly, we want to know what happens with an out-of-the-box Apache server. In these benchmarks, we start to drop or refuse connections at around 800 concurrent connections.

More troubling is that as these requests start stacking up, we see some that exceed 20 seconds or more. When this happens in a blocking server, each request behind it is queued; requests behind that are similarly queued and the entire thing starts to fall apart.

Even if we cannot hit 10,000 concurrent connections, there's a lot of room for improvement. While a single server of any capacity is no longer the way we expect a web server environment to be designed, being able to squeeze as much performance as possible out of that server, ostensibly with our concurrent, event-driven approach, should be our goal.

Handling requests

In an earlier chapter, we handled URL routing with Gorilla, a compact but feature-full framework. The Gorilla toolkit certainly makes this easier, but we should also know how to intercept the functionality to impose our own custom handler.

Here is a simple web router wherein we handle and direct requests using a custom `http.Server` struct, as shown in the following code:

```
var routes []string

type customRouter struct {

}

func (customRouter) ServeHTTP(rw http.ResponseWriter, r
  *http.Request) {

  fmt.Println(r.URL.Path);
}

func main() {

  var cr customRouter;

  server := &http.Server {
      Addr: ":9000",
      Handler:cr,
      ReadTimeout: 10 * time.Second,
      WriteTimeout: 10 * time.Second,
      MaxHeaderBytes: 1 << 20,
  }

  server.ListenAndServe()
}
```

Here, instead of using a built-in URL routing muxer and dispatcher, we're creating a custom server and custom handler type to accept URLs and route requests. This allows us to be a little more robust with our URL handling.

In this case, we created a basic, empty struct called `customRouter` and passed it to our custom server creation call.

We can add more elements to our `customRouter` type, but we really don't need to for this simple example. All we need to do is to be able to access the URLs and pass them along to a handler function. We'll have three: one for static content, one for dynamic content, and one for dynamic content from a database.

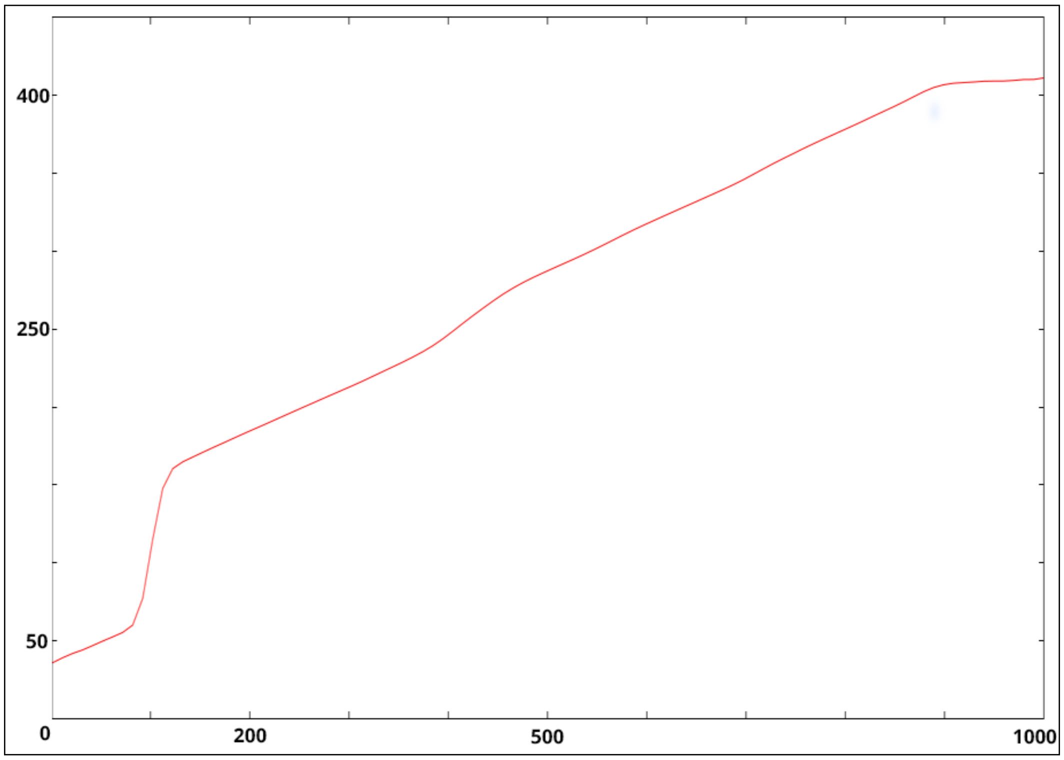
Before we go so far though, we should probably see what our absolute barebones HTTP server written in Go does when presented with the same traffic that we sent Apache's way.

By old school, we mean that the server will simply accept a request and pass along a static, flat file. You could do this using a custom router as we did earlier, taking requests, opening files, and then serving them, but Go provides a much simpler mode to handle this basic task in the `http.FileServer` method.

So, to get some benchmarks for the most basic of Go servers against Apache, we'll utilize a simple FileServer and test it against a `test.html` page (which contains the same 80 KB file that we had with Apache).

As our goal with this test is to improve our performance in serving flat and dynamic pages, the actual specs for the test suite are somewhat immaterial. We'd expect that while the metrics will not match from environment to environment, we should see a similar trajectory. That said, it's only fair we supply the environment used for these tests; in this case, we used a MacBook Air with a 1.4 GHz i5 processor and 4 GB of memory.

First, we'll do this with our absolute best performance out of the box with Apache, which had 850 concurrent connections and 900 total requests.



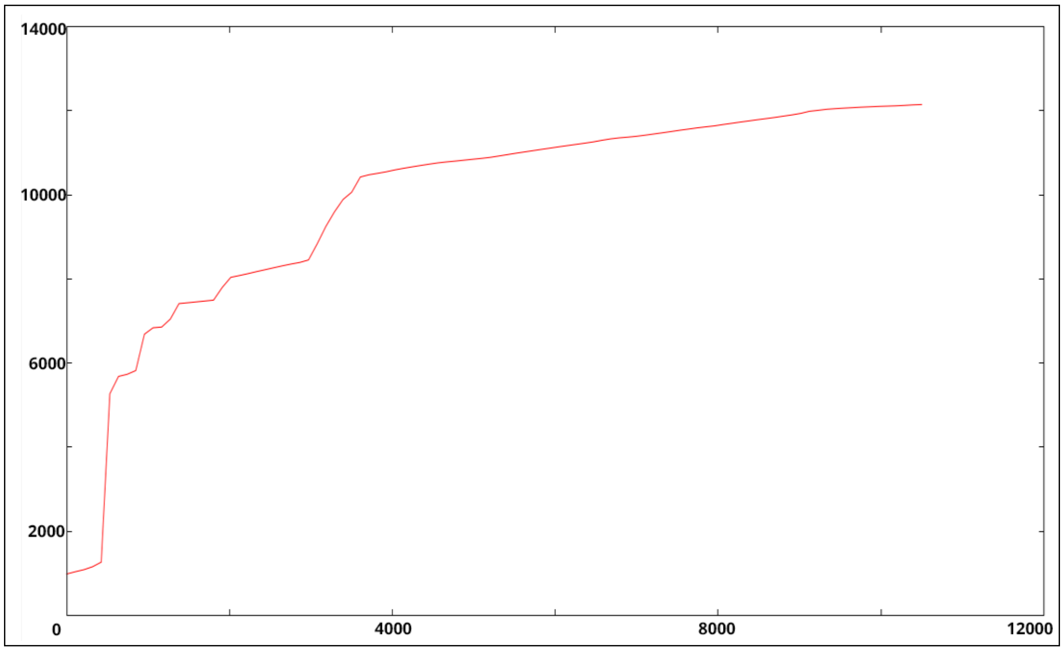
The results are certainly encouraging as compared to Apache. Neither of our test systems were tweaked much (Apache as installed and basic FileServer in Go), but Go's FileServer handles 1,000 concurrent connections without so much as a blip, with the slowest clocking in at 411 ms.

Apache has made a great number of strides pertaining to concurrency and performance options in the last five years, but to get there does require a bit of tuning and testing. The intent of this experiment is not intended to denigrate Apache, which is well tested and established. Instead, it's to compare the out-of-the-box performance of the world's number 1 web server against what we can do with Go.

To really get a baseline of what we can achieve in Go, let's see if Go's FileServer can hit 10,000 connections on a single, modest machine out of the box:

```
ab -n 10500 -c 10000 -g test.csv http://localhost:8080/a.html
```

We will get the following output:



Success! Go's FileServer by itself will easily handle 10,000 concurrent connections, serving flat, static content.

Of course, this is not the goal of this particular project—we'll be implementing real-world obstacles such as template parsing and database access, but this alone should show you the kind of starting point that Go provides for anyone who needs a responsive server that can handle a large quantity of basic web traffic.

Routing requests

So, let's take a step back and look again at routing our traffic through a traditional web server to include not only our static content, but also the dynamic content.

We'll want to create three functions that will route traffic from our `customRouter:serveStatic():: read` function and serve a flat file `serveRendered():`, parse a template to display `serveDynamic():`, connect to MySQL, apply data to a struct, and parse a template.

To take our requests and reroute, we'll change the `ServeHTTP` method for our `customRouter` struct to handle three regular expressions.

For the sake of brevity and clarity, we'll only be returning data on our three possible requests. Anything else will be ignored.

In a real-world scenario, we can take this approach to aggressively and proactively reject connections for requests we think are invalid. This would include spiders and nefarious bots and processes, which offer no real value as nonusers.

Serving pages

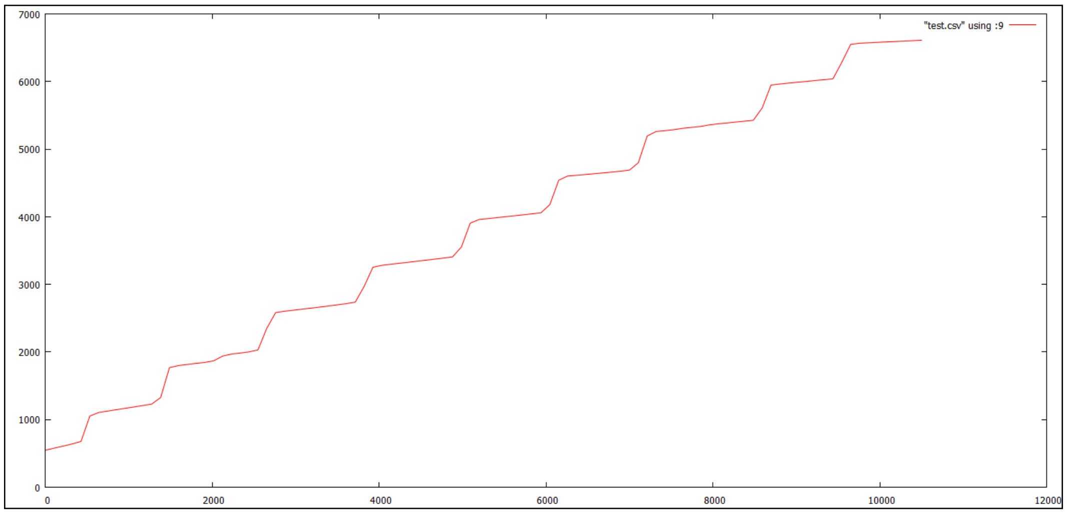
First up are our static pages. While we handled this the idiomatic way earlier, there exists the ability to rewrite our requests, better handle specific 404 error pages, and so on by using the `http.ServeFile` function, as shown in the following code:

```
path := r.URL.Path;

staticPatternString := "static/(.*)"
templatePatternString := "template/(.*)"
dynamicPatternString := "dynamic/(.*)"

staticPattern := regexp.MustCompile(staticPatternString)
templatePattern := regexp.MustCompile(templatePatternString)
dynamicDBPattern := regexp.MustCompile(dynamicPatternString)

if staticPattern.MatchString(path) {
  page := staticPath + staticPattern.ReplaceAllString(path,
   "${1}") + ".html"

  http.ServeFile(rw, r, page)
}
```

Here, we simply relegate all requests starting with `/static/(.*)` to match the request in addition to the `.html` extension. In our case, we've named our test file (the 80 KB example file) `test.html`, so all requests to it will go to `/static/test`.

We've prepended this with `staticPath`, a constant defined upcode. In our case, it's `/var/www/`, but you'll want to modify it as necessary.

So, let's see what kind of overhead is imposed by introducing some regular expressions, as shown in the following graph:



How about that? Not only is there no overhead imposed, it appears that the FileServer functionality itself is heavier and slower than a distinct `FileServe()` call. Why is that? Among other reasons, not explicitly calling the file to open and serve imposes an additional OS call, one which can cascade as requests mount up at the expense of concurrency and performance.

Sometimes it's the little things

Other than strictly serving flat pages here, we're actually doing one other task per request using the following line of code:

```
fmt.Println(r.URL.Path)
```

While this ultimately may have no impact on your final performance, we should take care to avoid unnecessary logging or related activities that may impart seemingly minimal performance obstacles that become much larger ones at scale.

Parsing our template

In our next phase, we'll measure the impact of reading and parsing a template. To effectively match the previous tests, we'll take our HTML static file and impose some variables on it.

If you recall, our goal here is to mimic real-world scenarios as closely as possible. A real-world web server will certainly handle a lot of static file serving, but today, dynamic calls make up the vast bulk of web traffic.

Our data structure will resemble the simplest of data tables without having access to an actual database:

```
type WebPage struct {
  Title string
  Contents string
}
```

We'll want to take any data of this form and render a template with it. Remember that Go creates the notion of public or private variables through the syntactical sugar of capitalized (public) or lowercase (private) values.

If you find that the template fails to render but you're not given explicit errors in the console, check your variable naming. A private value that is called from an HTML (or text) template will cause rendering to stop at that point.

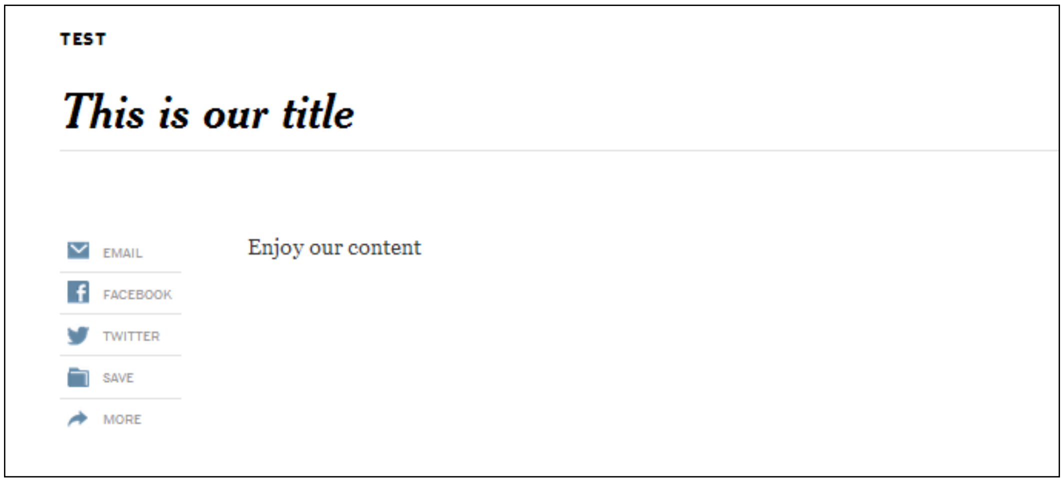
Now, we'll take that data and apply it to a template for any calls to a URL that begins with the `/(.*)` template. We could certainly do something more useful with the wildcard portion of that regular expression, so let's make it part of the title using the following code:

```
    } else if templatePattern.MatchString(path) {

      urlVar := templatePattern.ReplaceAllString(path, "${1}")
      page := WebPage{ Title: "This is our URL: "+urlVar, Contents:
        "Enjoy our content" }
      tmp, _ := template.ParseFiles(staticPath+"template.html")
      tmp.Execute(rw,page)

    }
```

Hitting `localhost:9000/template/hello` should render a template with a primary body of the following code:

```
<h1>{{.Title}}</h1>
<p>{{.Contents}}</p>
```

We will do this with the following output:

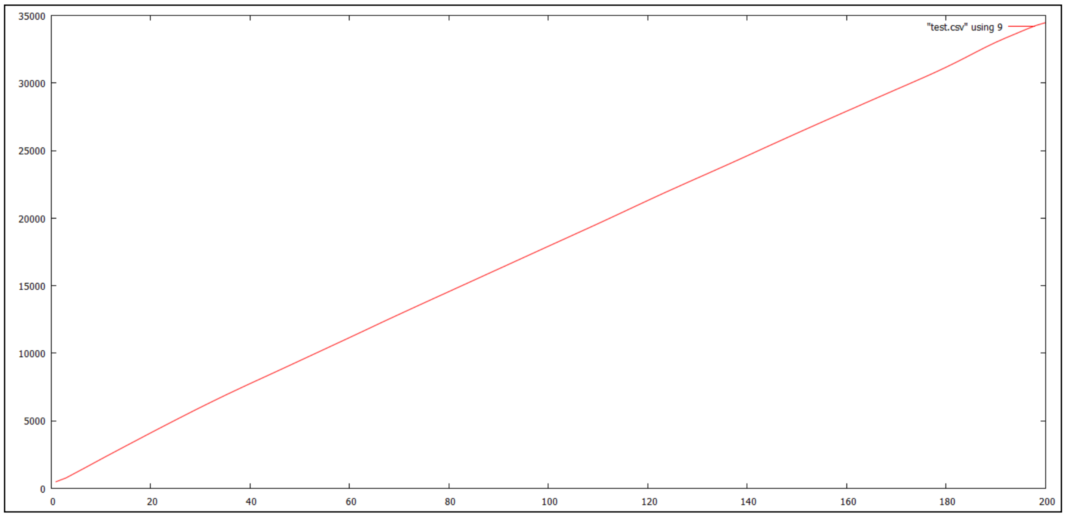

TEST

This is our title

Enjoy our content

One thing to note about templates is that they are not compiled; they remain dynamic. That is to say, if you create a renderable template and start your server, the template can be modified and the results are reflected.

This is noteworthy as a potential performance factor. Let's run our benchmarks again, with template rendering as the added complexity to our application and its architecture:



Yikes! What happened? We've gone from easily hitting 10,000 concurrent requests to barely handling 200.

To be fair, we introduced an intentional stumbling block, one not all that uncommon in the design of any given CMS.

You'll notice that we're calling the `template.ParseFiles()` method on every request. This is the sort of seemingly cheap call that can really add up when you start stacking the requests.

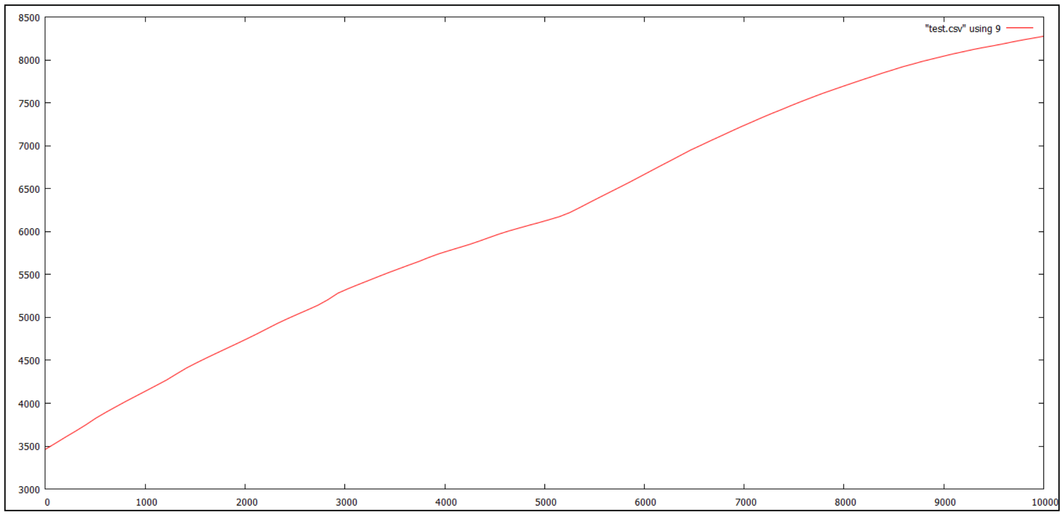
It may then make sense to move the file operations outside of the request handler, but we'll need to do more than that—to eliminate overhead and a blocking call, we need to set an internal cache for the requests.

Most importantly, all of our template creation and parsing should happen outside the actual request handler if you want to keep your server non-blocking, fast, and responsive. Here's another take:

```
var customHTML string
var customTemplate template.Template
var page WebPage
var templateSet bool


func main() {
  var cr customRouter;
  fileName := staticPath + "template.html"
  cH,_ := ioutil.ReadFile(fileName)
  customHTML = string(cH[:])

  page := WebPage{ Title: "This is our URL: ", Contents: "Enjoy
    our content" }
  cT,_ := template.New("Hey").Parse(customHTML)
  customTemplate = *cT
```

Even though we're using the `Parse()` function prior to our request, we can still modify our URL-specific variables using the `Execute()` method, which does not carry the same overhead as `Parse()`.

When we move this outside of the `customRouter` struct's `ServeHTTP()` method, we're back in business. This is the kind of response we'll get with these changes:



External dependencies

Finally, we need to bring in our biggest potential bottleneck, which is the database. As mentioned earlier, we'll simulate random traffic by generating a random integer between 1 and 10,000 to specify the article we want.

Randomization isn't just useful on the frontend—we'll want to work around any query caching within MySQL itself to limit nonserver optimizations.

Connecting to MySQL

We can route our way through a custom connection to MySQL using native Go, but as is often the case, there are a few third-party packages that make this process far less painful. Given that the database here (and associated libraries) is tertiary to the primary exercise, we'll not be too concerned about the particulars here.

The two mature MySQL driver libraries are as follows:

- **Go-MySQL-Driver** (`https://github.com/go-sql-driver/mysql`)
- **MyMySQL** (`https://github.com/ziutek/mymysql`)

For this example, we'll go with the Go-MySQL-Driver. We'll quickly install it using the following command:

```
go get github.com/go-sql-driver/mysql
```

Both of these implement the core SQL database connectivity package in Go, which provides a standardized method to connect to a SQL source and iterate over rows.

One caveat is if you've never used the SQL package in Go but have in other languages—typically, in other languages, the notion of an `Open()` method implies an open connection. In Go, this simply creates the struct and relevant implemented methods for a database. This means that simply calling `Open()` on `sql.database` may not give you relevant connection errors such as username/password issues and so on.

One advantage of this (or disadvantage depending on your vantage point) is that connections to your database may not be left open between requests to your web server. The impact of opening and reopening connections is negligible in the grand scheme.

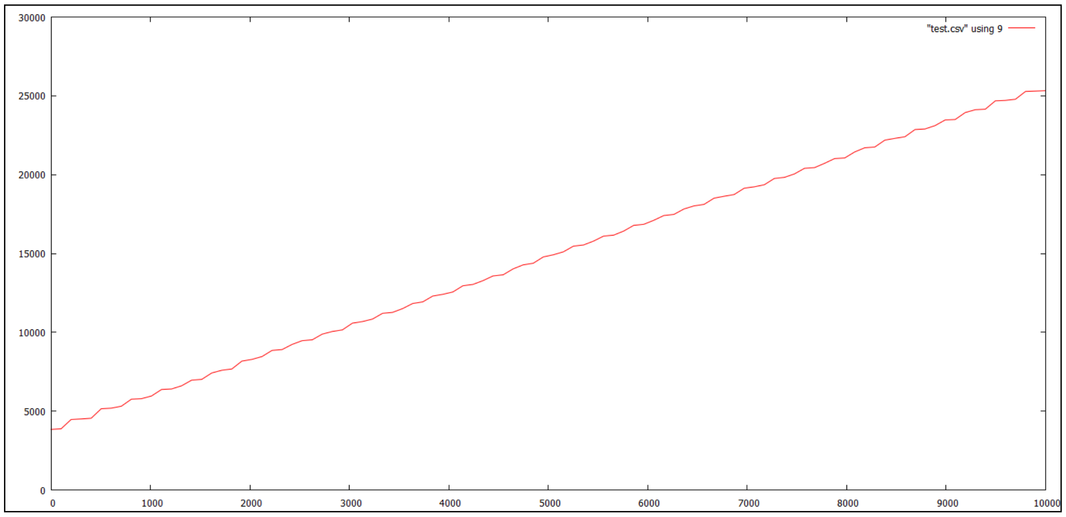
As we're utilizing a pseudo-random article request, we'll build a MySQL piggyback function to get an article by ID, as shown in the following code:

```
func getArticle(id int) WebPage {
  Database,err := sql.Open("mysql", "test:test@/master")
  if err != nil {
    fmt.Println("DB error!!!")
  }

  var articleTitle string
  sqlQ := Database.QueryRow("SELECT article_title from articles
    where article_id=? LIMIT 1", 1).Scan(&articleTitle)
  switch {
    case sqlQ == sql.ErrNoRows:
      fmt.Printf("No rows!")
    case sqlQ != nil:
      fmt.Println(sqlQ)
    default:

  }

  wp := WebPage{}
  wp.Title = articleTitle
  return wp

}
```

We will then call the function directly from our `ServeHTTP()` method, as shown in the following code:

```
}else if dynamicDBPattern.MatchString(path) {
  rand.Seed(9)
  id := rand.Intn(10000)
  page = getArticle(id)
  customTemplate.Execute(rw,page)
}
```

How did we do here? Take a look at the following graph:



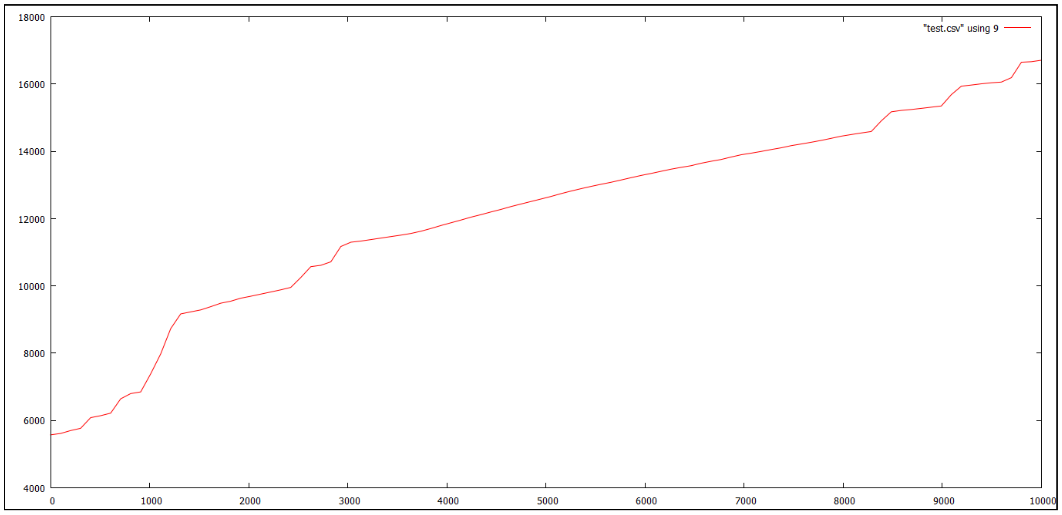
Slower, no doubt, but we held up to all 10,000 concurrent requests, entirely from uncached MySQL calls.

Given that we couldn't hit 1,000 concurrent requests with a default installation of Apache, this is nothing to sneeze at.

Multithreading and leveraging multiple cores

You may be wondering how performance may vary when invoking additional processor cores—as mentioned earlier, this can sometimes have an unexpected effect.

In this case, we should expect only improved performance in our dynamic requests and static requests. Any time the cost of context switching in the OS might outweigh the performance advantages of additional cores, we can see paradoxical performance degradation. In this case, we do not see this effect and instead see a relatively similar line, as shown in the following graph:



Exploring our web server

Our final web server is capable of serving static, template-rendered, and dynamic content well within the confines of the goal of 10,000 concurrent connections on even the most modest of hardware.

The code—much like the code in this book—can be considered a jumping-off point and will need refinement if put into production. This server lacks anything in the form of error handling but can ably serve valid requests without any issue. Let's take a look at the following server's code:

```
package main

import
(
"net/http"
"html/template"
"time"
"regexp"
"fmt"
"io/ioutil"
"database/sql"
"log"
"runtime"
_ "github.com/go-sql-driver/mysql"
)
```

Most of our imports here are fairly standard, but note the MySQL line that is called solely for its side effects as a database/SQL driver:

```
const staticPath string = "static/"
```

The relative `static/` path is where we'll look for any file requests—as mentioned earlier, this does no additional error handling, but the `net/http` package itself will deliver 404 errors should a request to a nonexistent file hit it:

```
type WebPage struct {

  Title string
  Contents string
  Connection *sql.DB

}
```

Our `WebPage` type represents the final output page before template rendering. It can be filled with static content or populated by data source, as shown in the following code:

```
type customRouter struct {

}

func serveDynamic() {

}

func serveRendered() {

}

func serveStatic() {

}
```

Use these if you choose to extend the web app—this makes the code cleaner and removes a lot of the cruft in the `ServeHTTP` section, as shown in the following code:

```
func (customRouter) ServeHTTP(rw http.ResponseWriter, r
  *http.Request) {
  path := r.URL.Path;

  staticPatternString := "static/(.*)"
  templatePatternString := "template/(.*)"
  dynamicPatternString := "dynamic/(.*)"

  staticPattern := regexp.MustCompile(staticPatternString)
  templatePattern := regexp.MustCompile(templatePatternString)
  dynamicDBPattern := regexp.MustCompile(dynamicPatternString)

  if staticPattern.MatchString(path) {
     serveStatic()
    page := staticPath + staticPattern.ReplaceAllString(path,
      "${1}") + ".html"
    http.ServeFile(rw, r, page)
  }else if templatePattern.MatchString(path) {

    serveRendered()
    urlVar := templatePattern.ReplaceAllString(path, "${1}")

    page.Title = "This is our URL: " + urlVar
    customTemplate.Execute(rw,page)

  }else if dynamicDBPattern.MatchString(path) {

    serveDynamic()
    page = getArticle(1)
    customTemplate.Execute(rw,page)
  }

}
```

All of our routing here is based on regular expression pattern matching. There are a lot of ways you can do this, but `regexp` gives us a lot of flexibility. The only time you may consider simplifying this is if you have so many potential patterns that it could cause a performance hit—and this means thousands. The popular web servers, Nginx and Apache, handle a lot of their configurable routing through regular expressions, so it's fairly safe territory:

```
func gobble(s []byte) {

}
```

Go is notoriously cranky about unused variables, and while this isn't always the best practice, you will end up, at some point, with a function that does nothing specific with data but keeps the compiler happy. For production, this is not the way you'd want to handle such data.

```
var customHTML string
var customTemplate template.Template
var page WebPage
var templateSet bool
var Database sql.DB

func getArticle(id int) WebPage {
  Database,err := sql.Open("mysql", "test:test@/master")
  if err != nil {
    fmt.Println("DB error!")
  }

  var articleTitle string
  sqlQ := Database.QueryRow("SELECT article_title from articles
    WHERE article_id=? LIMIT 1", id).Scan(&articleTitle)
  switch {
    case sqlQ == sql.ErrNoRows:
      fmt.Printf("No rows!")
    case sqlQ != nil:
      fmt.Println(sqlQ)
    default:

  }


  wp := WebPage{}
  wp.Title = articleTitle
  return wp

}
```

Our `getArticle` function demonstrates how you can interact with the `database/sql` package at a very basic level. Here, we open a connection and query a single row with the `QueryRow()` function. There also exists the `Query` command, which is also usually a `SELECT` command but one that could return more than a single row.

```
func main() {

  runtime.GOMAXPROCS(4)

  var cr customRouter;

  fileName := staticPath + "template.html"
  cH,_ := ioutil.ReadFile(fileName)
  customHTML = string(cH[:])

  page := WebPage{ Title: "This is our URL: ", Contents: "Enjoy
    our content" }
  cT,_ := template.New("Hey").Parse(customHTML)
  customTemplate = *cT

  gobble(cH)
  log.Println(page)
  fmt.Println(customTemplate)


  server := &http.Server {
      Addr: ":9000",
      Handler:cr,
      ReadTimeout: 10 * time.Second,
      WriteTimeout: 10 * time.Second,
      MaxHeaderBytes: 1 << 20,
  }

  server.ListenAndServe()

}
```

Our main function sets up the server, builds a default `WebPage` and `customRouter`, and starts listening on port `9000`.

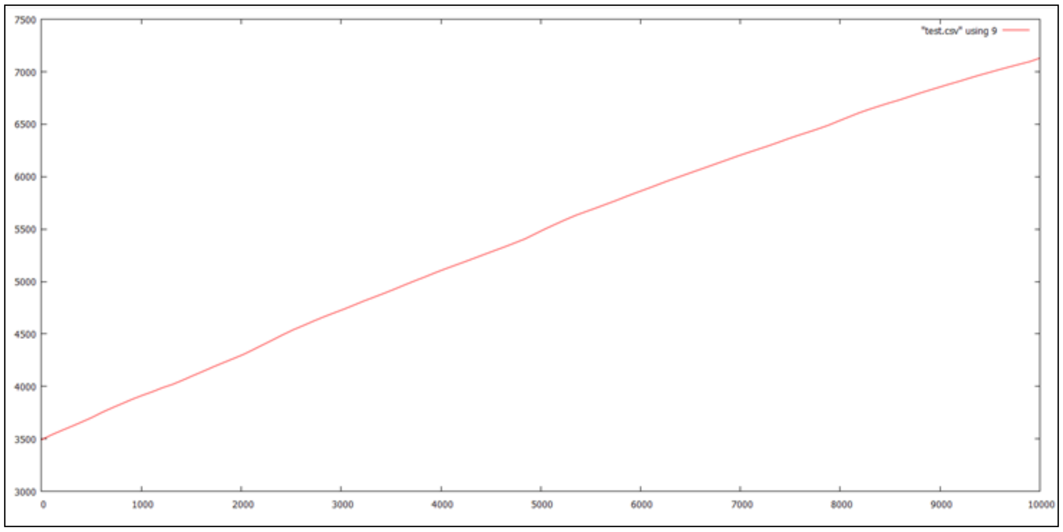
Timing out and moving on

One thing we did not focus on in our server is the notion of lingering connection mitigation. The reason we didn't worry much about it is because we were able to hit 10,000 concurrent connections in all three approaches without too much issue, strictly by utilizing Go's powerful built-in concurrency features.

Particularly when working with third-party or external applications and services, it's important to know that we can and should be prepared to call it quits on a connection (if our application design permits it).

Note the custom server implementation and two notes-specific properties: `ReadTimeout` and `WriteTimeout`. These allow us to handle this use case precisely.

In our example, this is set to an absurdly high 10 seconds. For a request to be received, processed, and sent, up to 20 seconds can transpire. This is an eternity in the Web world and has the potential to cripple our application. So, what does our C10K look like with 1 second on each end? Let's take a look at the following graph:



Here, we've saved nearly 5 seconds off the tail end of our highest volume of concurrent requests, almost certainly at the expense of complete responses to each.

It's up to you to decide how long it's acceptable to keep slow-running connections, but it's another tool in the arsenal to keep your server swift and responsive.

There will always be a tradeoff when you decide to kill a connection—too early and you'll have a bevy of complaints about a nonresponsive or error-prone server; too late and you'll be unable to cope with the connection volume programmatically. This is one of those considerations that will require QA and hard data.

Summary

The C10K problem may seem like a relic today, but the call to action was symptomatic of the type of approaches to systems' applications that were primarily employed prior to the rapid expansion of concurrent languages and application design.

Just 15 years ago, this seemed a largely insurmountable problem facing systems and server developers worldwide; now, it's handled with only minor tweaking and consideration by a server designer.

Go makes it easy to get there (with a little effort), but reaching 10,000 (or 100,000 or even 1,000,000) concurrent connections is only half the battle. We must know what to do when problems arise, how to seek out maximum performance and responsiveness out of our servers, and how to structure our external dependencies such that they do not create roadblocks.

In our next chapter, we'll look at squeezing even more performance out of our concurrent applications by testing some distributed computing patterns and best utilizing memory management.

7
Performance and Scalability

To build a high-powered web server in Go with just a few hundred lines of code, you should be quite aware of how concurrent Go provides us with exceptional tools for performance and stability out of the box.

Our example in *Chapter 6*, *C10K – A Non-blocking Web Server in Go*, also showed how imposing blocking code arbitrarily or inadvertently into our code can introduce some serious bottlenecks and quickly torpedo any plans to extend or scale your application.

What we'll look at in this chapter are a few ways that can better prepare us to take our concurrent application and ensure that it's able to continuously scale in the future and that it is capable of being expanded in scope, design, and/or capacity.

We'll expand a bit on **pprof**, the CPU profiling tool we looked at briefly in previous chapters, as a way to elucidate the way our Go code is compiled and to locate possible unintended bottlenecks.

Then we'll expand into distributed Go and into ways to offer some performance-enhancing parallel-computing concepts to our applications. We'll also look at the Google App Engine, and at how you can utilize it for your Go-based applications to ensure scalability is placed in the hands of one of the most reliable hosting infrastructures in the world.

Lastly, we'll look at memory utilization, preservation, and how Google's garbage collector works (and sometimes doesn't). We'll finally delve a bit deeper into using memory caching to keep data consistent as well as less ephemeral, and we will also see how that dovetails with distributed computing in general.

High performance in Go

Up to this point, we've talked about some of the tools we can use to help discover slowdowns, leaks, and inefficient looping.

Go's compiler and its built-in deadlock detector keep us from making the kind of mistake that's common and difficult to detect in other languages.

We've run time-based benchmarks based on specific changes to our concurrency patterns, which can help us design our application using different methodologies to improve overall execution speed and performance.

Getting deeper into pprof

The pprof tool was first encountered in *Chapter 5*, *Locks, Blocks, and Better Channels*, and if it still feels a bit cryptic, that's totally understandable. What pprof shows you in export is a **call graph**, and we can use this to help identify issues with loops or expensive calls on the heap. These include memory leaks and processor-intensive methods that can be optimized.

One of the best ways to demonstrate how something like this works is to build something that doesn't. Or at least something that doesn't work the way it should.

You might be thinking that a language with garbage collection might be immune to these kinds of memory issues, but there are always ways to hide mistakes that can lead to memory leakage. If the GC can't find it, it can sometimes be a real pain to do so yourself, leading to a lot of—often feckless—debugging.

To be fair, what constitutes a memory leak is sometimes debated among computer science members and experts. A program that continuously consumes RAM may not be leaking memory by technical definition if the application itself could re-access any given pointers. But that's largely irrelevant when you have a program that crashes and burns after consuming memory like an elephant at a buffet.

The basic premise of creating a memory leak in a garbage-collected language relies on hiding the allocation from the compiler—indeed, any language in which you can access and utilize memory directly provides a mechanism for introducing leaks.

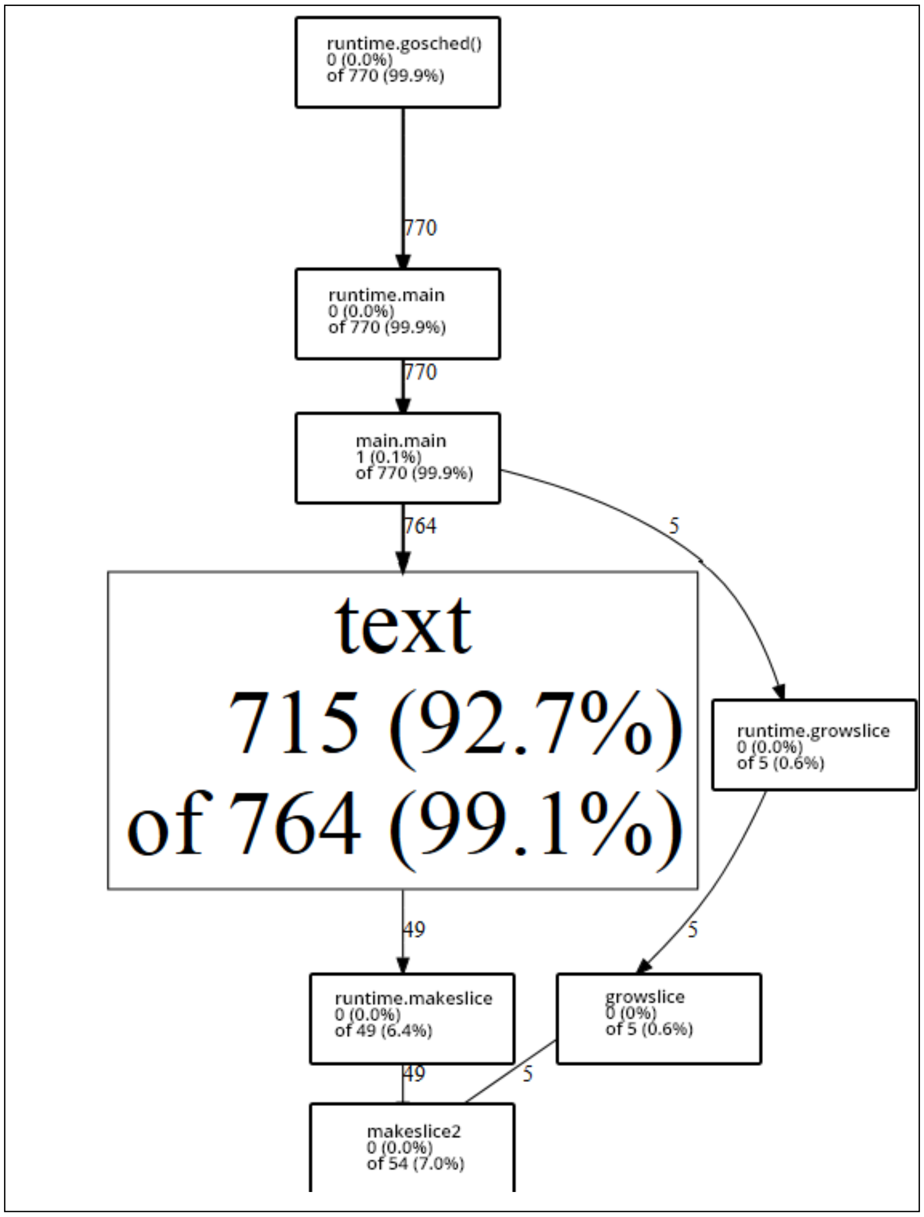
We'll review a bit more about garbage collection and Go's implementation later in this chapter.

So how does a tool like pprof help? Very simply put, by showing you **where** your memory and CPU utilization goes.

Let's first design a very obvious CPU hog as follows to see how pprof highlights this for us:

```
package main

import (
"os"
"flag"
"fmt"
"runtime/pprof"
)

const TESTLENGTH = 100000
type CPUHog struct {
  longByte []byte
}

func makeLongByte() []byte {
  longByte := make([]byte,TESTLENGTH)

  for i:= 0; i < TESTLENGTH; i++ {
    longByte[i] = byte(i)
  }
  return longByte
}

var profile = flag.String("cpuprofile", "", "output pprof data to
  file")


func main() {
  var CPUHogs []CPUHog

  flag.Parse()
    if *profile != "" {
      flag,err := os.Create(*profile)
      if err != nil {
        fmt.Println("Could not create profile",err)
      }
      pprof.StartCPUProfile(flag)
      defer pprof.StopCPUProfile()

    }

  for i := 0; i < TESTLENGTH; i++ {
    hog := CPUHog{}
    hog.longByte = makeLongByte()
    _ = append(CPUHogs,hog)
  }
}
```

The output of the preceding code is shown in the following diagram:



In this case, we know where our stack resource allocation is going, because we willfully introduced the loop (and the loop within that loop).

Imagine that we didn't intentionally do that and had to locate resource hogs. In this case, pprof makes this pretty easy, showing us the creation and memory allocation of simple strings comprising the majority of our samples.

We can modify this slightly to see the changes in the pprof output. In an effort to allocate more and more memory to see whether we can vary the pprof output, we might consider heavier types and more memory.

The easiest way to accomplish that is to create a slice of a new type that includes a significant amount of these heavier types such as int64. We're blessed with Go: in that, we aren't prone to common C issues such as buffer overflows and memory protection and management, but this makes debugging a little trickier when we cannot intentionally break the memory management system.

The unsafe package

Despite the built-in memory protection provided, there is still another interesting tool provided by Go: the **unsafe** package. As per Go's documentation:

Package unsafe contains operations that step around the type safety of Go programs.

This might seem like a curious library to include—indeed, while many low-level languages allow you to shoot your foot off, it's fairly unusual to provide a segregated language.

Later in this chapter, we'll examine `unsafe.Pointer`, which allows you to read and write to arbitrary bits of memory allocation. This is obviously extraordinarily dangerous (or useful and nefarious, depending on your goal) functionality that you would generally try to avoid in any development language, but it does allow us to debug and understand our programs and the Go garbage collector a bit better.

So to increase our memory usage, let's switch our string allocation as follows, for random type allocation, specifically for our new struct `MemoryHog`:

```
type MemoryHog struct {
  a,b,c,d,e,f,g int64
  h,i,j,k,l,m,n float64
  longByte []byte
}
```

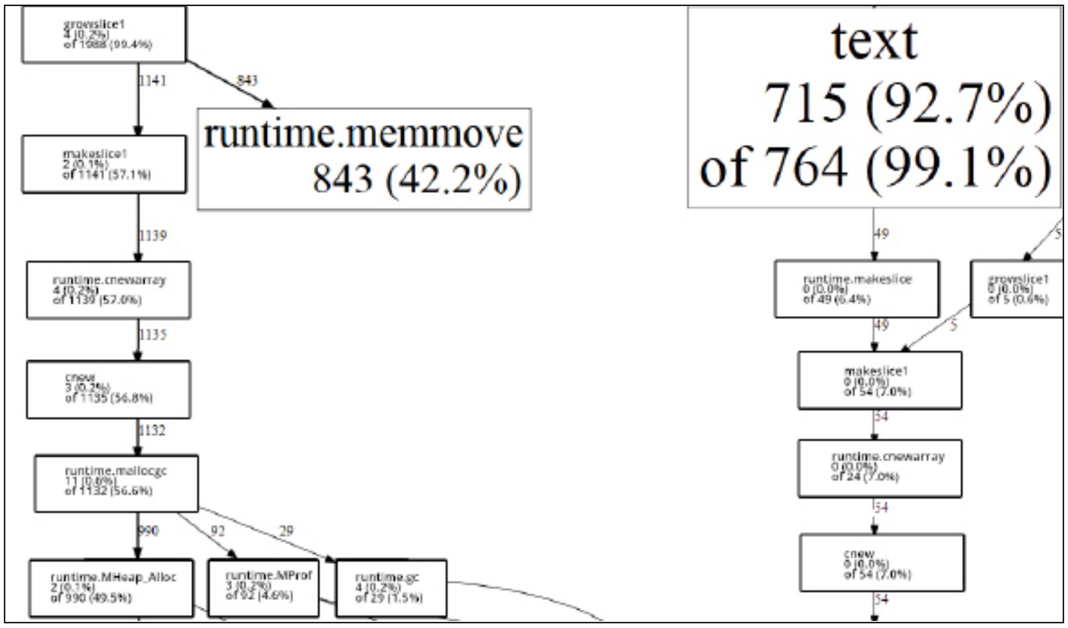
There's obviously nothing preventing us from extending this into some ludicrously large set of slices, huge arrays of int64s, and so on. But our primary goal is solely to change the output of pprof so that we can identify movement in the call graph's samples and its effect on our stack/heap profiles.

Our arbitrarily expensive code looks as follows:

```
type MemoryHog struct {
  a,b,c,d,e,f,g int64
  h,i,j,k,l,m,n float64
  longByte []byte
}

func makeMemoryHog() []MemoryHog {

  memoryHogs := make([]MemoryHog,TESTLENGTH)

  for i:= 0; i < TESTLENGTH; i++ {
    m := MemoryHog{}
    _ = append(memoryHogs,m)
  }

  return memoryHogs
}


var profile = flag.String("cpuprofile", "", "output pprof data to
  file")


func main() {
  var CPUHogs []CPUHog

  flag.Parse()
    if *profile != "" {
      flag,err := os.Create(*profile)
      if err != nil {
        fmt.Println("Could not create profile",err)
      }
      pprof.StartCPUProfile(flag)
      defer pprof.StopCPUProfile()

    }


  for i := 0; i < TESTLENGTH; i++ {
    hog := CPUHog{}
    hog.mHog = makeMemoryHog()
    _ = append(CPUHogs,hog)
  }
}
```

With this in place, our CPU consumption remains about the same (due to the looping mechanism remaining largely unchanged), but our memory allocation has increased—unsurprisingly—by about 900 percent. It's unlikely that you will precisely duplicate these results, but the general trend of a small change leading to a major difference in resource allocation is reproducible. Note that memory utilization reporting is possible with pprof, but it's not what we're doing here; the memory utilization observations here happened outside of pprof.

If we took the extreme approach suggested previously—to create absurdly large properties for our struct—we could carry that out even further, but let's see what the aggregate impact is on our CPU profile on execution. The impact is shown in the following diagram:



On the left-hand side, we have our new allocation approach, which invokes our larger struct instead of an array of strings. On the right-hand side, we have our initial application.

A pretty dramatic flux, don't you think? While neither of these programs is wrong in design, we can easily toggle our methodologies to see where resources are going and discern how we can reduce their consumption.

Parallelism's and concurrency's impact on I/O pprof

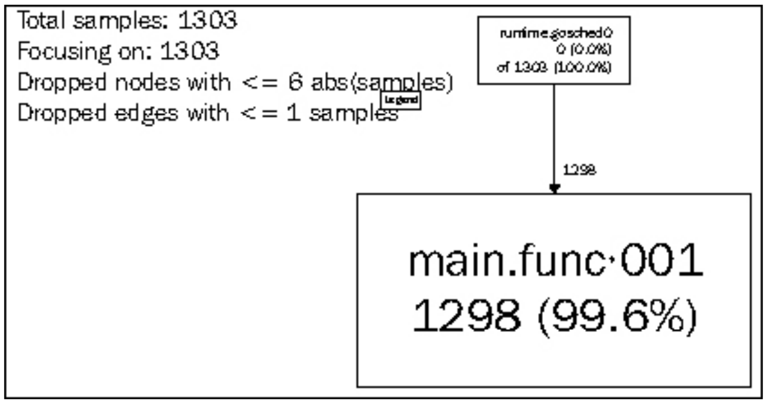
One issue you'll likely run into pretty quickly when using pprof is when you've written a script or application that is especially bound to efficient runtime performance. This happens most frequently when your program executes too quickly to properly profile.

A related issue involves network applications that require connections to profile; in this case, you can simulate traffic either in-program or externally to allow proper profiling.

We can demonstrate this easily by replicating something like the preceding example with goroutines as follows:

```
const TESTLENGTH = 20000

type DataType struct {
  a,b,c,d,e,f,g int64
  longByte []byte
}

func (dt DataType) init() {

}

var profile = flag.String("cpuprofile", "", "output pprof data to
  file")

func main() {

  flag.Parse()
    if *profile != "" {
      flag,err := os.Create(*profile)
      if err != nil {
        fmt.Println("Could not create profile",err)
      }
      pprof.StartCPUProfile(flag)
      defer pprof.StopCPUProfile()
    }
  var wg sync.WaitGroup

  numCPU := runtime.NumCPU()
  runtime.GOMAXPROCS(numCPU)

  wg.Add(TESTLENGTH)

  for i := 0; i < TESTLENGTH; i++ {
    go func() {
      for y := 0; y < TESTLENGTH; y++ {
        dT := DataType{}
        dT.init()
      }
      wg.Done()
    }()
  }

  wg.Wait()

  fmt.Println("Complete.")
}
```

The following diagram shows the pprof output of the preceding code:



It's not nearly as informative, is it?

If we want to get something more valuable about the stack trace of our goroutines, Go—as usual—provides some additional functionality.

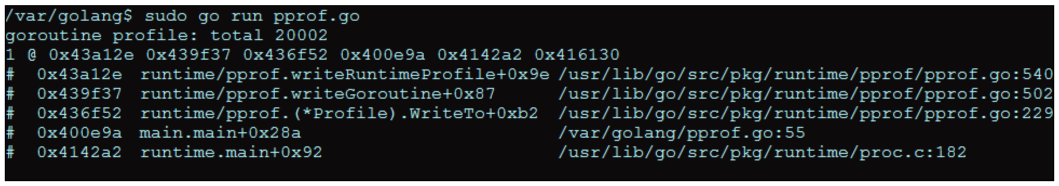
In the runtime package, there is a function and a method that allow us to access and utilize the stack traces of our goroutines:

- `runtime.Lookup`: This function returns a profile based on name
- `runtime.WriteTo`: This method sends the snapshot to the I/O writer

If we add the following line to our program, we won't see the output in the `pprof` Go tool, but we can get a detailed analysis of our goroutines in the console.

```
pprof.Lookup("goroutine").WriteTo(os.Stdout, 1)
```

The previous code line gives us some more of the abstract goroutine memory location information and package detail, which will look something like the following screenshot:



```
/var/golang$ sudo go run pprof.go
goroutine profile: total 20002
1 @ 0x43a12e 0x439f37 0x436f52 0x400e9a 0x4142a2 0x416130
#  0x43a12e  runtime/pprof.writeRuntimeProfile+0x9e /usr/lib/go/src/pkg/runtime/pprof/pprof.go:540
#  0x439f37  runtime/pprof.writeGoroutine+0x87      /usr/lib/go/src/pkg/runtime/pprof/pprof.go:502
#  0x436f52  runtime/pprof.(*Profile).WriteTo+0xb2  /usr/lib/go/src/pkg/runtime/pprof/pprof.go:229
#  0x400e9a  main.main+0x28a                        /var/golang/pprof.go:55
#  0x4142a2  runtime.main+0x92                      /usr/lib/go/src/pkg/runtime/proc.c:182
```

But an even faster way to get this output is by utilizing the `http/pprof` tool, which keeps the results of our application active via a separate server. We've gone with port 6000 here as shown in the following code, though you can modify this as necessary:

```
go func() {
  log.Println(http.ListenAndServe("localhost:6000", nil))
}()
```

While you cannot get an SVG output of the goroutine stack call, you can see it live in your browser by going to `http://localhost:6060/debug/pprof/goroutine?debug=1`.

Using the App Engine

While not right for every project, Google's App Engine can open up a world of scalability when it comes to concurrent applications, without the hassle of VM provisioning, reboots, monitoring, and so on.

The App Engine is not entirely dissimilar to Amazon Web Services, DigitalOcean, and the ilk, except for the fact that you do not need to necessarily involve yourself in the minute details of direct server setup and maintenance. All of them provide a single spot to acquire and utilize virtual computing resources for your applications.

Rather, it can be a more abstract environment within Google's architecture with which to house and run your code in a number of languages, including—no surprise here—the Go language itself.

While large-scale apps will cost you, Google provides a free tier with reasonable quotas for experimentation and small applications.

The benefits as they relate to scalability here are two-fold: you're not responsible for ensuring uptime on the instances as you would be in an AWS or DigitalOcean scenario. Who else but Google will have not only the architecture to support anything you can throw at it, but also have the fastest updates to the Go core itself?

There are some obvious limitations here that coincide with the advantages, of course, including the fact that your core application will be available exclusively via `http` (although it will have access to plenty of other services).

> To deploy apps to the App Engine, you'll need the SDK for Go, available for Mac OS X, Linux, and Windows, at `https://developers.google.com/appengine/downloads#Google_App_Engine_SDK_for_Go`.

Once you've installed the SDK, the changes you'll need to make to your code are minor—the most noteworthy point is that for most cases, your Go tool command will be supplanted by `goapp`, which handles serving your application locally and then deploying it.

Distributed Go

We've certainly covered a lot about concurrent and parallel Go, but one of the biggest infrastructure challenges for developers and system architects today has to do with cooperative computing.

Some of the applications and designs that we've mentioned previously scale from parallelism to distributed computing.

Memcache(d) is a form of in-memory caching, which can be used as a queue among several systems.

Our master-slave and producer-consumer models we presented in *Chapter 4*, *Data Integrity in an Application*, have more to do with distributed computing than single-machine programming in Go, which manages concurrency idiomatically. These models are typical concurrency models in many languages, but can be scaled to help us design distributed systems as well, utilizing not just many cores and vast resources but also redundancy.

The basic premise of distributed computing is to share, spread, and best absorb the various burdens of any given application across many systems. This not only improves performance on aggregate, but provides some sense of redundancy for the system itself.

This all comes at some cost though, which are as follows:

- Potential for network latency
- Creating slowdowns in communication and in application execution
- Overall increase in complexity both in design and in maintenance
- Potential for security issues at various nodes along the distributed route(s)
- Possible added cost due to bandwidth considerations

This is all to say, simply, that while building a distributed system can provide great benefits to a large-scale application that utilizes concurrency and ensures data consistency, it's by no means right for every example.

Types of topologies

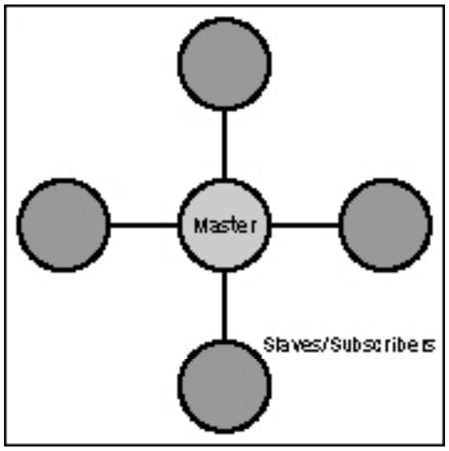
Distributed computing recognizes a slew of logical topologies for distributed design. Topology is an apt metaphor, because the positioning and logic of the systems involved can often represent physical topology.

Out of the box, not all of the accepted topologies apply to Go. When we design concurrent, distributed applications using Go, we'll generally rely on a few of the simpler designs, which are as follows.

Type 1 – star

The star topology (or at least this particular form of it), resembles our master-slave or producer-consumer models as outlined previously.

The primary method of data passing involves using the master as a message-passing conduit; in other words, all requests and commands are coordinated by a single instance, which uses some routing method to pass messages. The following diagram shows the star topology:



We can actually very quickly design a goroutine-based system for this. The following code is solely the master's (or distributed destination's) code and lacks any sort of security considerations, but shows how we can parlay network calls to goroutines:

```
package main

import
(
  "fmt"
  "net"

)
```

Our standard, basic libraries are defined as follows:

```
type Subscriber struct {
  Address net.Addr
  Connection net.Conn
  do chan Task
}

type Task struct {
  name string
}
```

These are the two custom types we'll use here. A `Subscriber` type is any distributed helper that comes into the fray, and a `Task` type represents any given distributable task. We've left that undefined here because it's not the primary goal of demonstration, but you could ostensibly have `Task` do anything by communicating standardized commands across the TCP connection. The `Subscriber` type is defined as follows:

```
var SubscriberCount int
var Subscribers []Subscriber
var CurrentSubscriber int
var taskChannel chan Task

func (sb Subscriber) awaitTask() {
  select {
    case t := <-sb.do:
      fmt.Println(t.name,"assigned")

  }
}

func serverListen (listener net.Listener) {
  for {
    conn,_ := listener.Accept()

    SubscriberCount++

    subscriber := Subscriber{ Address: conn.RemoteAddr(),
      Connection: conn }
    subscriber.do = make(chan Task)
    subscriber.awaitTask()
    _ = append(Subscribers,subscriber)

  }
}

func doTask() {
  for {
    select {
      case task := <-taskChannel:
        fmt.Println(task.name,"invoked")
        Subscribers[CurrentSubscriber].do <- task
        if (CurrentSubscriber+1) > SubscriberCount {
          CurrentSubscriber = 0
        }else {
          CurrentSubscriber++
        }
    }

  }
}

func main() {

  destinationStatus := make(chan int)

  SubscriberCount = 0
  CurrentSubscriber = 0

  taskChannel = make(chan Task)

  listener, err := net.Listen("tcp", ":9000")
  if err != nil {
    fmt.Println ("Could not start server!",err)
  }
  go serverListen(listener)
  go doTask()

  <-destinationStatus
}
```

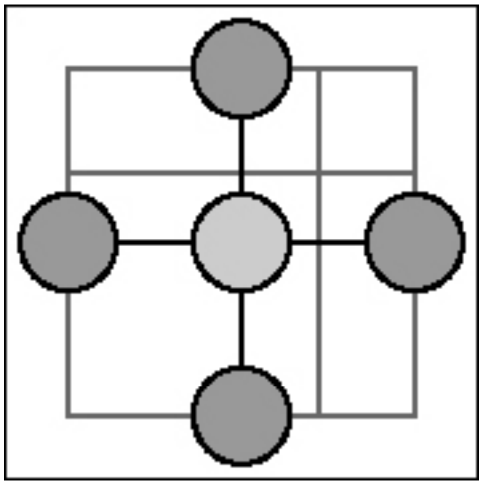
This essentially treats every connection as a new `Subscriber`, which gets its own channel based on its index. This master server then iterates through existing `Subscriber` connections using the following very basic round-robin approach:

```
  if (CurrentSubscriber+1) > SubscriberCount {
    CurrentSubscriber = 0
  }else {
    CurrentSubscriber++
  }
```

As mentioned previously, this lacks any sort of security model, which means that any connection to port 9000 would become a `Subscriber` and could get network messages assigned to it (and ostensibly could invoke new messages too). But you may have noticed an even bigger omission: this distributed application doesn't do anything. Indeed, this is just a model for assignment and management of subscribers. Right now, it doesn't have any path of action, but we'll change that later in this chapter.

Type 2 – mesh

The mesh is very similar to the star with one major difference: each node is able to communicate not just through the master, but also directly with other nodes as well. This is also known as a **complete graph**. The following diagram shows a mesh topology:



For practical purposes, the master must still handle assignments and pass connections back to the various nodes.

This is actually not particularly difficult to add through the following simple modification of our previous server code:

```
func serverListen (listener net.Listener) {
  for {
    conn,_ := listener.Accept()

    SubscriberCount++

    subscriber := Subscriber{ Address: conn.RemoteAddr(),
      Connection: conn }
    subscriber.awaitTask()
    _ = append(Subscribers,subscriber)
    broadcast()
  }
}
```

Then, we add the following corresponding `broadcast` function to share all available connections to all other connections:

```
func broadcast() {
  for i:= range Subscribers {
    for j:= range Subscribers {
      Subscribers[i].Connection.Write
        ([]byte("Subscriber:",Subscriber[j].Address))
    }
  }
}
```

The Publish and Subscribe model

In both the previous topologies, we've replicated a Publish and Subscribe model with a central/master handling delivery. Unlike in a single-system, concurrent pattern, we lack the ability to use channels directly across separate machines (unless we use something like Go's Circuit as described in *Chapter 4, Data Integrity in an Application*).

Without direct programmatic access to send and receive actual commands, we rely on some form of API. In the previous examples, there is no actual task being sent or executed, but how could we do this?

Obviously, to create tasks that can be formalized into non-code transmission, we'll need a form of API. We can do this one of two ways: serialization of commands, ideally via JSONDirect transmission, and execution of code.

As we'll always be dealing with compiled code, the serialization of commands option might seem like you couldn't include Go code itself. This isn't exactly true, but passing full code in any language is fairly high on lists of security concerns.

But let's look at two ways of sending data via API in a task by removing a URL from a slice of URLs for retrieval. We'll first need to initialize that array in our `main` function as shown in the following code:

```
type URL struct {
  URI string
  Status int
  Assigned Subscriber
  SubscriberID int
}
```

Every URL in our array will include the URI, its status, and the subscriber address to which it's been assigned. We'll formalize the status points as 0 for unassigned, 1 for assigned and waiting, and 2 for assigned and complete.

Remember our `CurrentSubscriber` iterator? That represents the next-in-line round robin assignment which will fulfill the `SubscriberID` value for our `URL` struct.

Next, we'll create an arbitrary array of URLs that will represent our overall job here. Some suspension of incredulity may be necessary to assume that the retrieval of four URLs should require any distributed system; in reality, this would introduce significant slowdown by virtue of network transmission. We've handled this in a purely single-system, concurrent application before:

```
  URLs = []URL{ {Status:0,URL:"http://golang.org/"},
    {Status:0,URL:"http://play.golang.org/"},
      {Status:0,URL:"http://golang.org/doc/"},
        {Status:0,URL:"http://blog.golang.org/"} }
```

Serialized data

In our first option in the API, we'll send and receive serialized data in JSON. Our master will be responsible for formalizing its command and associated data. In this case, we'll want to transmit a few things: what to do (in this case, retrieve) with the relevant data, what the response should be when it is complete, and how to address errors.

We can represent this in a custom struct as follows:

```
type Assignment struct {
  command string
  data string
  successResponse string
  errorResponse string
}
...
  asmnt := Assignment{command:"process",
    url:"http://www.golang.org",successResponse:"success",
      errorResponse:"error"}
  json, _ := json.Marshal(asmnt )
  send(string(json))
```

Remote code execution

The remote code execution option is not necessarily separate from serialization of commands, but instead of structured and interpreted formatted responses, the payload could be code that will be run via a system command.

As an example, code from any language could be passed through the network and executed from a shell or from a syscall library in another language, like the following Python example:

```
from subprocess import call
call([remoteCode])
```

The disadvantages to this approach are many: it introduces serious security issues and makes error detection within your client nearly impossible.

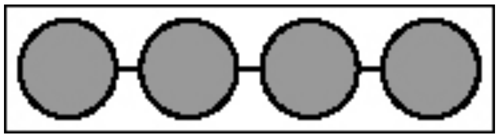
The advantages are you do not need to come up with a specific format and interpreter for responses as well as potential speed improvements. You can also offload the response code to another external process in any number of languages.

In most cases, serialization of commands is far preferable over the remote code execution option.

Other topologies

There exist quite a few topology types that are more complicated to manage as part of a messaging queue.

The following diagram shows the bus topology:

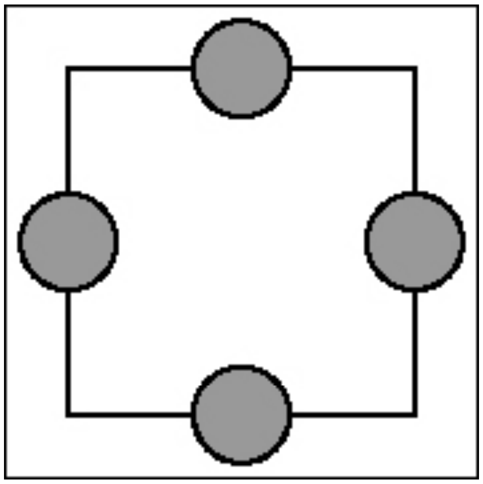


The bus topology network is a unidirectional transmission system. For our purposes, it's neither particularly useful nor easily managed, as each added node needs to announce its availability, accept listener responsibility, and be ready to cede that responsibility when a new node joins.

The advantage of a bus is quick scalability. This comes with serious disadvantages though: lack of redundancy and single point of failure.

Even with a more complex topology, there will always be some issue with potentially losing a valuable cog in the system; at this level of modular redundancy, some additional steps will be necessary to have an always-available system, including automatic double or triple node replication and failovers. That's a bit more than we'll get into here, but it's important to note that the risk will be there in any event, although it would be a little more vulnerable with a topology like the bus.

The following diagram shows the ring topology:



The ring topology looks similar to our mesh topology, but lacks a master. It essentially requires the same communication process (announce and listen) as does a bus. Note one significant difference: instead of a single listener, communication can happen between any node without the master.

This simply means that all nodes must both listen and announce their presence to other nodes.

Message Passing Interface

There exists a slightly more formalized version of what we built previously, called Message Passing Interface. MPI was borne from early 1990s academia as a standard for distributed communication.

Originally written with FORTRAN and C in mind, it is still a protocol, so it's largely language agnostic.

MPI allows the management of topology above and beyond the basic topologies we were able to build for a resource management system, including not only the line and ring but also the common bus topology.

For the most part, MPI is used by the scientific community; it is a highly concurrent and analogous method for building large-scale distributed systems. Point-to-point operations are more rigorously defined with error handling, retries, and dynamic spawning of processes all built in.

Our previous basic examples lend no prioritization to processors, for example, and this is a core effect of MPI.

There is no official implementation of MPI for Go, but as there exists one for both C and C++, it's entirely possible to interface with it through that.

> There is also a simple and incomplete binding written in Go by Marcus Thierfelder that you can experiment with. It is available at `https://github.com/marcusthierfelder/mpi`.
>
> You can read more about and install OpenMPI from `http://www.open-mpi.org/`.
>
> Also you can read more about MPI and MPICH implementations at `http://www.mpich.org/`.

Some helpful libraries

There's little doubt that Go provides some of the best ancillary tools available to any compiled language out there. Compiling to native code on a myriad of systems, deadlock detection, pprof, fmt, and more allow you to not just build high-performance applications, but also test them and format them.

This hasn't stopped the community from developing other tools that can be used for debugging or aiding your concurrent and/or distributed code. We'll take a look at a few great tools that may prove worthy of inclusion in your app, particularly if it's highly visible or performance critical.

Nitro profiler

As you are probably now well aware, Go's pprof is extremely powerful and useful, if not exactly user-friendly.

If you love pprof already, or even if you find it arduous and confusing, you may love Nitro profiler twice as much. Coming from Steve Francia of spf13, Nitro profiler allows you to produce even cleaner analyses of your application and its functions and steps, as well as providing more usable a/b tests of alternate functions.

> Read more about Nitro profiler at `http://spf13.com/project/nitro`.
>
> You can get it via `github.com/spf13/nitro`.

As with pprof, Nitro automatically injects flags into your application, and you'll see them in the results themselves.

Unlike pprof, your application does not need to be compiled to get profile analysis from it. Instead, you can simply append `-stepAnalysis` to the `go run` command.

Heka

Heka is a data pipeline tool that can be used to gather, analyze, and distribute raw data. Available from Mozilla, Heka is more a standalone application rather than a library, but when it comes to acquiring, analyzing, and distributing data such as server logfiles across multiple servers, Heka can prove itself worthy.

Heka is also written in Go, so make sure to check out the source to see how Mozilla utilizes concurrency and Go in real-time data analysis.

> You can visit the Heka home page at `http://heka-docs.readthedocs.org/en/latest/` and the Heka source page at `https://github.com/mozilla-services/heka`.

GoFlow

Finally, there's GoFlow, a flow-based programming paradigm tool that lets you segment your application into distinct components, each capable of being bound to ports, channels, the network, or processes.

While not itself a performance tool, GoFlow might be an appropriate approach to extending concurrency for some applications.

> Visit GoFlow at `https://github.com/trustmaster/goflow`.

Memory preservation

At the time of this writing, Go 1.2.2's compiler utilizes a naive mark/sweep garbage collector, which assigns a reference rank to objects and clears them when they are no longer in use. This is noteworthy only to point out that it is widely considered a relatively poor garbage collection system.

So why does Go use it? As Go has evolved; language features and compiler speed have largely taken precedence over garbage collection. While it's a long-term development timeline for Go, for the time being, this is where we are. The tradeoff is a good one, though: as you well know by now, compiling Go code is light years faster than, say, compiling C or C++ code. Good enough for now is a fair description for the GC. But there are some things you can do to augment and experiment within the garbage collection system.

Garbage collection in Go

To get an idea of how the garbage collector is managing the stack at any time, take a look at the `runtime.MemProfileRecord` object, which keeps track of presently living objects in the active stack trace.

You can call the profile record when necessary and then utilize it against the following methods to get a few interesting pieces of data:

- `InUseBytes()`: This method has the bytes used presently as per the memory profile
- `InUseObjects()`:This method has the number of live objects in use
- `Stack()`: This method has the full stack trace

You can place the following code in a heavy loop in your application to get a peek at all of these:

```
var mem runtime.MemProfileRecord
obj := mem.InUseObjects();
bytes := mem.InUseBytes();
stack := mem.Stack();
fmt.Println(i,obj,bytes)
```

Summary

We can now build some pretty high-performance applications and then utilize some of Go's built-in tools and third-party packages to seek out the most performance in a single instance application as well as across multiple, distributed systems.

In the next chapter, we're going to wrap everything together to design and build a concurrent server application that can work quickly and independently, and easily scale in performance and scope.

8
Concurrent Application Architecture

By now, we've designed small bits of concurrent programs, primarily in a single piece keeping concurrency largely isolated. What we haven't done yet is tie everything together to build something a little more robust, complex, and more daunting to manage from an administrator's perspective.

Simple chat applications and web servers are fine and dandy. However, you will eventually need more complexity and require external software to meet all of the more advanced requirements.

In this case, we'll build something that's satisfied by a few dissonant services: a file manager with revision control that supplies web and shell access. Services such as Dropbox and Google Drive allow users to keep and share files among peers. On the other hand, GitHub and its ilk allow for a similar platform but with the critical added benefit of revision control.

Many organizations face problems with the following sharing and distribution options:

- Limitations on repositories, storage, or number of files
- Potential inaccessibility if the services are down
- Security concerns, particularly for sensitive information

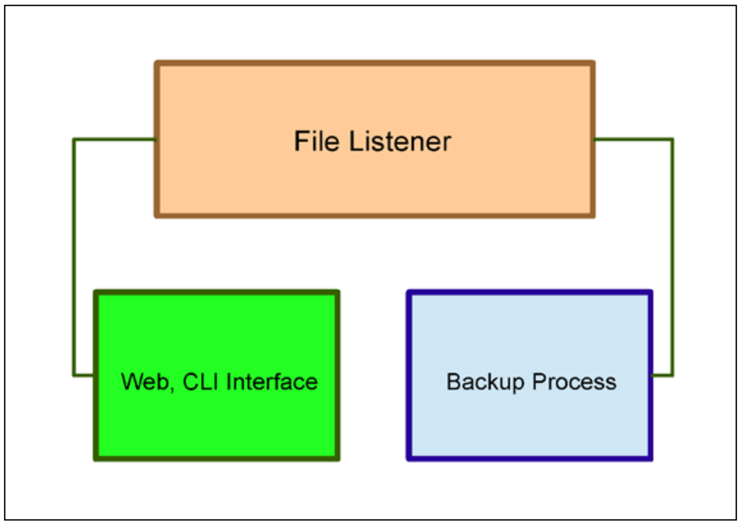
Simple sharing applications such as Dropbox and Google Drive are great at storing data without a large amount of revision control options. GitHub is an excellent collaborative revision control and distribution system, but comes with many costs and the mistakes by developers can lead to large and potentially serious security lapses.

We'll be combining the aims of version control (and the GitHub ideal) with Dropbox's / Google Drive's simplicity and openness. This type of application will be perfect as an intranet replacement—wholly isolated and accessible with custom authentication that doesn't necessarily rely on cloud services. The ability to keep it all in-house removes any potential for network security concerns and allows an administrator to design permanent backup solutions in a way that fits their organization.

File sharing within the organization will allows forking, backups, file locking, and revision control all from the command line but also through a simple web interface.

Designing our concurrent application

When designing a concurrent application, we will have three components running in separate processes. A file listener will be alerted to make changes to files in specified locations. A web-CLI interface will allow users to augment or modify files, and a backup process will be bound to the listener to provide automated copies of new file changes. With that in mind, these three processes will look a bit like what is shown in the following diagram:



Our file listener process will do the following three things:

- Keep an eye on any file changes
- Broadcast to our web/CLI servers and the backup process
- Maintain the state of any given file in our database / data store

The backup process will accept any broadcasts from the file listener (#2) and create a backup file in an iterative design.

Our general server (web and CLI) will report details on individual files and allow versioning forward and backward with a customizable syntax. This part of the application will also have to broadcast back to the file listener when new files are committed or revisions are requested.

Identifying our requirements

The most critical step in our architectural design process is really zooming in on the required features, packages, and technologies that we'll need to implement. For our file management and revision control application, there are a few key points that will stand out:

- A web interface that allows file uploads, downloads, and revisions.
- A command-line interface that allows us to roll back changes and modify files directly.
- A filesystem listener that finds changes made to a shared location.
- A data store system that has strong Go tie-in and allows us to maintain information about files and users in a mostly consistent manner. This system will also maintain user records.
- A concurrent log system that maintains and cycles logs of changed files.

We're somewhat complicating things by allowing the following three different ways to interface with the overall application:

- Via the Web that requires a user and login. This also allows our users to access and modify files even if they happen to be somewhere not connected to the shared drive.
- Via the command line. This is archaic but also extremely valuable anytime a user is traversing a filesystem, particularly power users not in a GUI.
- Via the filesystem that changes itself. This is the shared drive mechanism wherein we assume that any user with access to this will be making valid modifications to any files.

To handle all of this, we can identify a few critical technologies as follows:

- A database or data store to manage revisions to our filesystem. When choosing between transactional, ACID-compliant SQL and fast document stores in NoSQL, the tradeoff is often performance versus consistency. However, since most of our locking mechanism will exist in the application, duplicating locks (even at the row level) will add a level of potential slowness and cruft that we don't need. So, we will utilize a NoSQL solution.
- This solution will need to play well with concurrency.
- We'll be using a web interface, one that brings in powerful and clean routing/muxing and plays well with Go's robust built-in templating system.
- A filesystem notification library that allows us to monitor changes to files as well as backing up revisions.

Any solutions we uncover or build to satisfy these requirements will need to be highly concurrent and non-blocking. We'll want to make sure that we do not allow simultaneous changes to files, including changes to our internal revisions themselves.

With all of this in mind, let's identify our pieces one-by-one and decide how they will play in our application.

We'll also present a few alternatives with options that can be swapped without compromising the functionality or core requirements. This will allow some flexibility in cases where platform or preference makes our primary option unpalatable. Any time we're designing an application, it's a good idea to know
what else is out there in case the software (or terms of its use) change or it is no longer satisfactory to use at a future scale.

Let's start with our data store.

Using NoSQL as a data store in Go

One of the biggest concessions with using NoSQL is, obviously, the lack of standardization when it comes to CRUD operations (create, read, update, and delete). SQL has been standardized since 1986 and is pretty airtight across a number of databases—from MySQL to SQL Server and from Microsoft and Oracle all the way down to PostgreSQL.

You can read more about NoSQL and various NoSQL platforms at `http://nosql-database.org/`.

Martin Fowler has also written a popular introduction to the concept and some use cases in his book *NoSQL Distilled* at `http://martinfowler.com/books/nosql.html`.

Depending on the NoSQL platform, you can also lose ACID compliance and durability. This means that your data is not 100 percent secure—there can be transactional loss if a server crashes, if reads happen on outdated or non-existent data, and so on. The latter of which is known as a dirty read.

This is all noteworthy as it applies to our application and with concurrency specifically because we've talked about one of those big potential third-party bottlenecks in the previous chapters.

For our file-sharing application in Go, we will utilize NoSQL to store metadata about files as well as the users that modify/interact with those files.

We have quite a few options when it comes to a NoSQL data store to use here, and almost all of the big ones have a library or interface in Go. While we're going to go with Couchbase here, we'll briefly talk about some of the other big players in the game as well as the merits of each.

The code snippets in the following sections should also give you some idea of how to switch out Couchbase for any of the others without too much angst. While we don't go deeply into any of them, the code for maintaining the file and modifying information will be as generic as possible to ensure easy exchange.

MongoDB

MongoDB is one of the most popular NoSQL platforms available. Written in 2009, it's also one of the most mature platforms, but comes with a number of tradeoffs that have pushed it somewhat out of favor in the recent years.

Even so, Mongo does what it does in a reliable fashion and with a great deal of speed. Utilizing indices, as is the case with most databases and data stores, improves query speed on reads greatly.

Mongo also allows for some very granular control of guarantees as they apply to reads, writes, and consistency. You can think of this as a very vague analog to any language and/or engine that supports syntactical dirty reads.

Most importantly, Mongo supports concurrency easily within Go and is implicitly designed to work in distributed systems.

> The biggest Go interface for Mongo is `mgo`, which is available at: `http://godoc.org/labix.org/v2/mgo`.

Should you wish to experiment with Mongo in Go, it's a relatively straightforward process to take your data store record and inject it into a custom struct. The following is a quick and dirty example:

```
import
(
    "labix.org/v2/mgo"
    "labix.org/v2/mgo/bson"
)


type User struct {
  name string
}


func main() {
  servers, err := mgo.Dial("localhost")
  defer servers.Close()
  data := servers.DB("test").C("users")
  result := User{}
  err = c.Find(bson.M{"name": "John"}).One(&result)
}
```

One downside to Mongo compared to other NoSQL solutions is that it does not come with any GUI by default. This means we either need to tie in another application or web service, or stick to the command line to manage its data store. For many applications, this isn't a big deal, but we want to keep this project as compartmentalized and provincial as possible to limit points of failure.

Mongo has also gotten a bit of a bad rap as it pertains to fault tolerance and data loss, but this is equally true of many NoSQL solutions. In addition, it's in many ways a feature of a fast data store—so often catastrophe recovery comes at the expense of speed and performance.

It's also fair to say this is a generally overblown critique of Mongo and its peers. Can something bad happen with Mongo? Sure. Can it also happen with a managed Oracle-based system? Absolutely. Mitigating massive failures in this realm is more the responsibility of a systems administrator than the software itself, which can only provide the tools necessary to design such a contingency plan.

All that said, we'll want something with a quick and highly-available management interface, so Mongo is out for our requirements but could easily be plugged into this solution if those are less highly valued.

Redis

Redis is another key/value data store and, as of recently, took the number one spot in terms of total usage and popularity. In an ideal Redis world, an entire dataset is held in memory. Given the size of many datasets, this isn't always possible; however, coupled with Redis' ability to eschew durability, this can result in some very high performance results when used in concurrent applications.

Another useful feature of Redis is the fact that it can inherently hold different data structures. While you can make abstractions of such data by unmarshalling JSON objects/arrays in Mongo (and other data stores), Redis can handle sets, strings, arrays, and hashes.

There are two major accepted libraries for Redis in Go:

- **Radix**: This is a minimalist client that's barebones, quick, and dirty.
  To install Radix, run the following command:
  ```
  go get github.com/fzzy/radix/redis
  ```
- **Redigo**: This more robust and a bit more complex, but provides a lot of the more intricate functionality that we'll probably not need for this project. To install Redigo, run the following command:
  ```
  go get github.com/garyburd/redigo/redis
  ```

We'll now see a quick example of getting a user's name from the data store of `Users` in Redis using Redigo:

```
package main

import
(
    "fmt"
    "github.com/garyburd/redigo/redis"
)
```

```
func main() {

  connection,_ := dial()
  defer connection.Close()


  data, err := redis.Values(connection.Do("SORT", "Users", "BY",
"User:*->name",
    "GET", "User:*->name"))

  if (err) {
    fmt.Println("Error getting values", err)
  }

  for i:= range data {
    var Uname string
    data,err := redis.Scan(data, &Uname)
    if (err) {
      fmt.Println("Error getting value",err)
    }else {
      fmt.Println("Name Uname")
    }
  }
}
```

Looking over this, you might note some non programmatic access syntax, such as the following:

```
  data, err := redis.Values(connection.Do("SORT", "Users", "BY",
"User:*->name",
    "GET", "User:*->name"))
```

This is indeed one of the reasons why Redis in Go will not be our choice for this project—both libraries here provide an almost API-level access to certain features with some more detailed built-ins for direct interaction. The `Do` command passes straight queries directly to Redis, which is fine if you need to use the library, but a somewhat inelegant solution across the board.

Both the libraries play very nicely with the concurrent features of Go, and you'll have no problem making non-blocking networked calls to Redis through either of them.

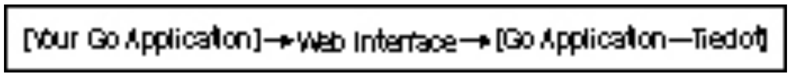
It's worth noting that Redis only supports an experimental build for Windows, so this is mostly for use on *nix platforms. The port that does exist comes from Microsoft and can be found at `https://github.com/MSOpenTech/redis`.

Tiedot

If you've worked a lot with NoSQL, then the preceding engines all likely seemed very familiar to you. Redis, Couch, Mongo, and so on are all virtual stalwarts in what is a relatively young technology.

Tiedot, on the other hand, probably isn't as familiar. We're including it here only because the document store itself is written in Go directly. Document manipulation is handled primarily through a web interface, and it's a JSON document store like several other NoSQL solutions.

As document access and handling is governed via HTTP, there's a somewhat counterintuitive workflow, shown as follows:



As that introduces a potential spot for latency or failure, this keeps from being an ideal solution for our application here. Keep in mind that this is also a feature of a few of the other solutions mentioned earlier, but since Tiedot is written in Go, it would be significantly easier to connect to it and read/modify data using a package. While this book was being written, this did not exist.

Unlike other HTTP- or REST-focused alternatives such as CouchDB, Tiedot relies on URL endpoints to dictate actions, not HTTP methods.

You can see in the following code how we might handle something like this through standard libraries:

```
package main

import
(
  "fmt"
  "json"
  "http"
)

type Collection struct {
  Name string
}
```

This, simply, is a data structure for any record you wish to bring into your Go application via data selects, queries, and so on. You saw this in our previous usage of SQL servers themselves, and this is not any different:

```
func main() {

  Col := Collection{
    Name: ''
  }

  data, err := http.Get("http://localhost:8080/all")
  if (err != nil) {
    fmt.Println("Error accessing tiedot")
  }
  collections,_ = json.Unmarshal(data,&Col)
}
```

While not as robust, powerful, or scalable as many of its peers, Tiedot is certainly worth playing with or, better yet, contributing to.

[You can find Tiedot at `https://github.com/HouzuoGuo/tiedot`.]

CouchDB

CouchDB from Apache Incubator is another one of the big boys in NoSQL big data. As a JSON document store, CouchDB offers a great deal of flexibility when it comes to your data store approach.

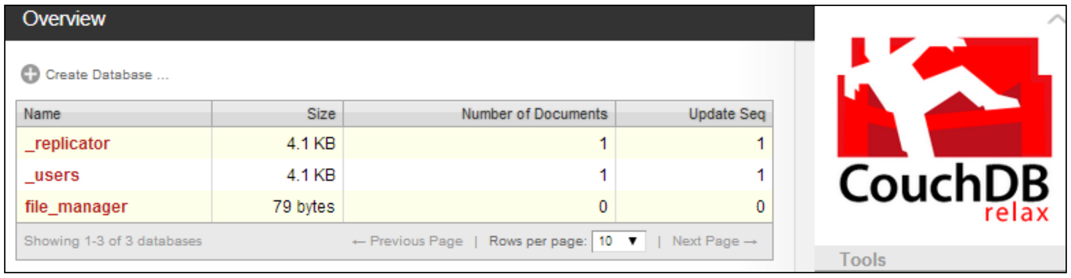
CouchDB supports ACID semantics and can do so concurrently, which provides a great deal of performance benefit if one is bound to those properties. In our application, that reliance on ACID consistency is somewhat flexible. By design, it will be failure tolerant and recoverable, but for many, even the possibility of data loss with recoverability is still considered catastrophic.

Interfacing with CouchDB happens via HTTP, which means there is no need for a direct implementation or Go SQL database hook to use it. Interestingly, CouchDB uses HTTP header syntax to manipulate data, as follows:

- **GET**: This represents read operations
- **PUT**: This represents creation operations
- **DELETE**: This represents deletion and update operations

These are, of course, what the header methods were initially intended in HTTP 1.1, but so much of the Web has focused on GET/POST that these tend to get lost in the fray.

Couch also comes with a convenient web interface for management. When CouchDB is running, you're able to access this at `http://localhost:5984/_utils/`, as shown in the following screenshot:



Overview

Create Database ...

Name	Size	Number of Documents	Update Seq
_replicator	4.1 KB	1	1
_users	4.1 KB	1	1
file_manager	79 bytes	0	0

Showing 1-3 of 3 databases ← Previous Page | Rows per page: 10 | Next Page →

That said, there are a few wrappers that provide a level of abstraction for some of the more complicated and advanced features.

Cassandra

Cassandra, another Apache Foundation project, isn't technically a NoSQL solution but a clustered (or cluster-able) database management platform.

Like many NoSQL applications, there is a limitation in the traditional query methods in Cassandra, for example, subqueries and joins are generally not supported.

We're mentioning it here primarily because of its focus on distributed computing as well as the ability to programmatically tune whether data consistency or performance is more important. Much of that is equally expressed in our solution, Couchbase, but Cassandra has a deeper focus on distributed data stores.

Cassandra does, however, support a subset of SQL that will make it far more familiar to developers who have dabbled in MySQL, PostgreSQL, or the ilk. Cassandra's built-in handling of highly concurrent integrations makes it in many ways ideal for Go, although it is an overkill for this project.

The most noteworthy library to interface with Cassandra is gocql, which focuses on speed and a clean connection to the Cassandra connection. Should you choose to use Cassandra in lieu of Couchbase (or other NoSQL), you'll find a lot of the methods that can be simply replaced.

The following is an example of connecting to a cluster and writing a simple query:

```
package main

import
(
    "github.com/gocql/gocql"
    "log"
)

func main() {

  cass := gocql.NewCluster("127.0.0.1")
  cass.Keyspace = "filemaster"
  cass.Consistency = gocql.LocalQuorum

  session, _ := cass.CreateSession()
  defer session.Close()

  var fileTime int;

  if err := session.Query(`SELECT file_modified_time FROM filemaster
  WHERE filename = ? LIMIT 1`,
  "test.txt").Consistency(gocql.One).Scan(&fileTime); err != nil {
    log.Fatal(err)
  }
  fmt.Println("Last modified",fileTime)
}
```

Cassandra may be an ideal solution if you plan on rapidly scaling this application, distributing it widely, or are far more comfortable with SQL than data store / JSON access.

For our purposes here, SQL is not a requirement and we value speed over anything else, including durability.

Couchbase

Couchbase is a relative newcomer in the field, but it was built by people from both CouchDB and memcached. Written in Erlang, it shares many of the same focuses on concurrency, speed, and non-blocking behavior that we've come to expect from a great deal of our Go applications.

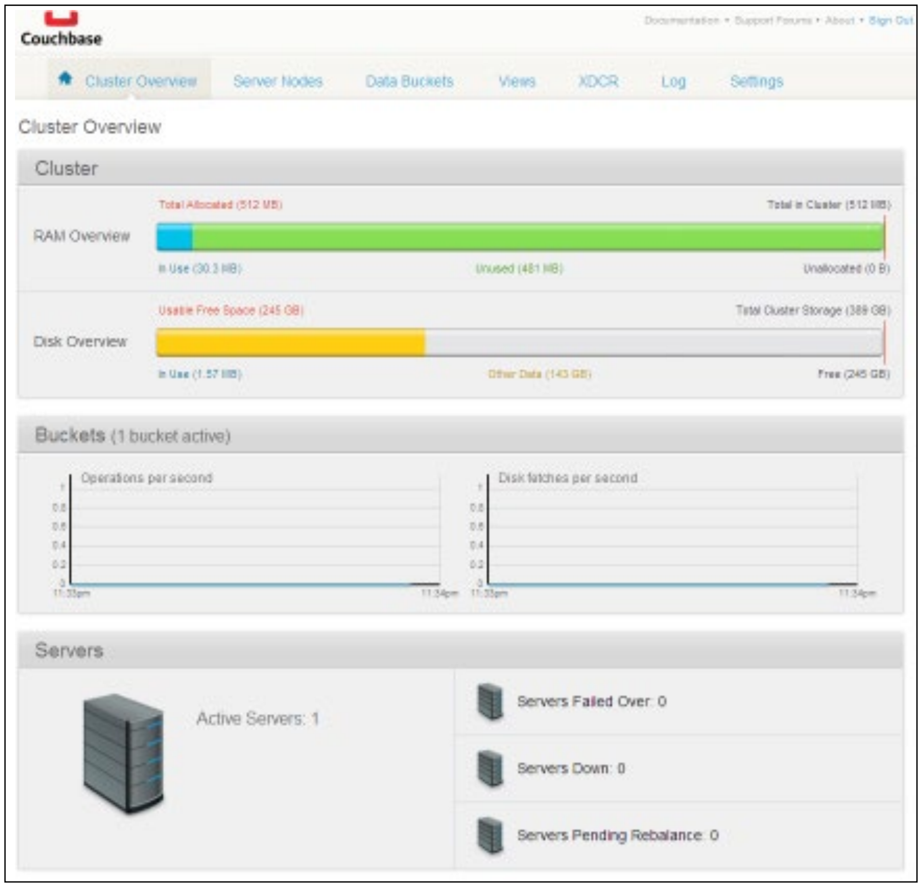
Couchbase also supports a lot of the other features we've discussed in the previous chapters, including easy distribution-based installations, tuneable ACID compliance, and low-resource consumption.

One caveat on Couchbase is it doesn't run well (or at all) on some lower-resourced machines or VMs. Indeed, 64-bit installations require an absolute minimum of 4 GB of memory and four cores, so forget about launching this on tiny, small, or even medium-grade instances or older hardware.

While most NoSQL solutions presented here (or elsewhere) offer performance benefits over their SQL counterparts in general, Couchbase has done very well against its peers in the NoSQL realm itself.

Couchbase, such as CouchDB, comes with a web-based graphical interface that simplifies the process of both setup and maintenance. Among the advanced features that you'll have available to you in the setup include your base bucket storage engine (Couchbase or memcached), your automated backup process (replicas), and the level of read-write concurrency.

In addition to configuration and management tools, it also presents some real-time monitoring in the web dashboard as shown in the following screenshot:



While not a replacement for full-scale server management (what happens when this server goes down and you have no insight), it's incredibly helpful to know exactly where your resources are going without needing a command-line method or an external tool.

The vernacular in Couchbase varies slightly, as it tends to in many of these solutions. The nascent desire to slightly separate NoSQL from stodgy old SQL solutions will pop its head from time to time.

With Couchbase, a database is a data bucket and records are documents. However, views, an old transactional SQL standby, bring a bit of familiarity to the table. The big takeaway here is views allow you to create more complex queries using simple JavaScript, in some cases, replicating otherwise difficult features such as joins, unions, and pagination.

Each view created in Couchbase becomes an HTTP access point. So a view that you name `select_all_files` will be accessible via a URL such as `http://localhost:8092/file_manager/_design/select_all_files/_view/Select%20All%20Files?connection_timeout=60000&limit=10&skip=0`.

The most noteworthy Couchbase interface library is Go Couchbase, which, if nothing else, might save you from some of the redundancy of making HTTP calls in your code to access CouchDB.

Go Couchbase can be found at `https://github.com/couchbaselabs/go-couchbase`.

Go Couchbase makes interfacing with Couchbase through a Go abstraction simple and powerful. The following code connects and grabs information about the various data pools in a lean way that feels native:

```
package main

import
(
  "fmt"
  "github.com/couchbaselabs/go-couchbase"
)

func main() {

    conn, err := couchbase.Connect("http://localhost:8091")
    if err != nil {
      fmt.Println("Error:",err)
    }
```

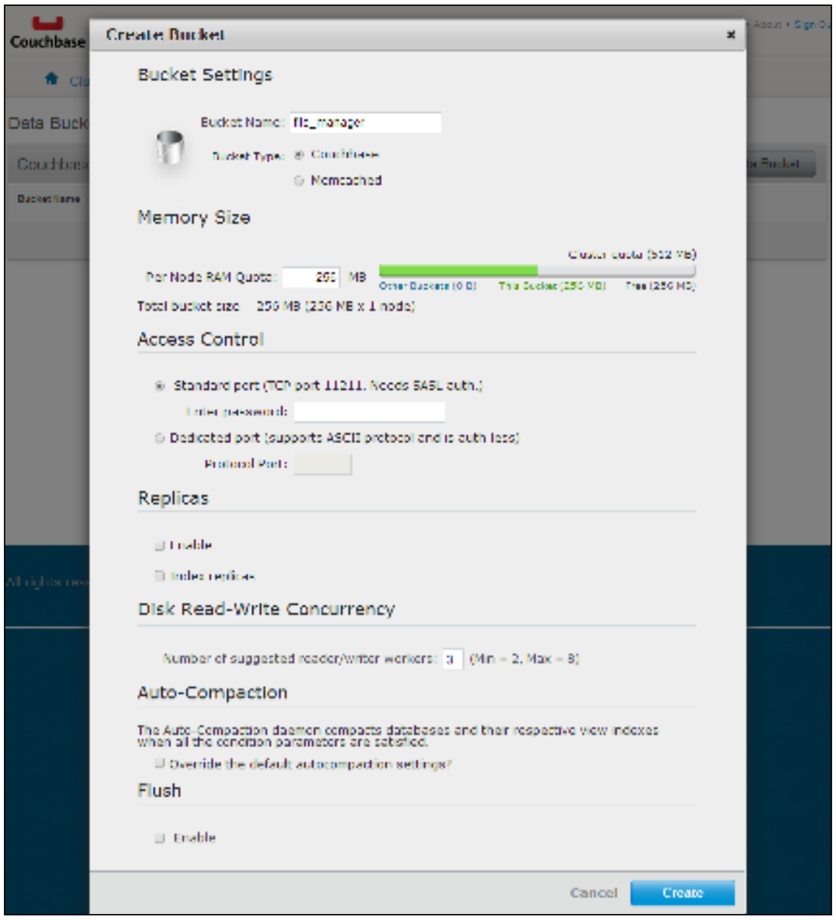
```
        for _, pn := range conn.Info.Pools {
            fmt.Printf("Found pool:  %s -> %s\n", pn.Name, pn.URI)
        }
    }
```

Setting up our data store

After installing Couchbase, you can access its administration panel by default at localhost and port 8091.

You'll be given an opportunity to set up an administrator, other IPs to connect (if you're joining a cluster), and general data store design.

After that, you'll need to set up a bucket, which is what we'll use to store all information about individual files. Here is what the interface for the bucket setup looks like:



Create Bucket

Bucket Settings

Bucket Name: file_manager

Bucket Type: (● Couchbase) ○ Memcached

Memory Size

Per Node RAM Quota: 256 MB

Cluster quota (512 MB)

Other Buckets (0 B) This Bucket (256 MB) Free (256 MB)

Total bucket size 256 MB (256 MB x 1 node)

Access Control

(● Standard port (TCP port 11211. Needs SASL auth.))

Enter password:

○ Dedicated port (supports ASCII protocol and is auth less)

Protocol Port:

Replicas

☐ Enable

☐ Index replicas

Disk Read-Write Concurrency

Number of suggested reader/writer workers: 3 (Min = 2, Max = 8)

Auto-Compaction

The Auto-Compaction daemon compacts databases and their respective view indexes when all the condition parameters are satisfied.

☐ Override the default autocompaction settings?

Flush

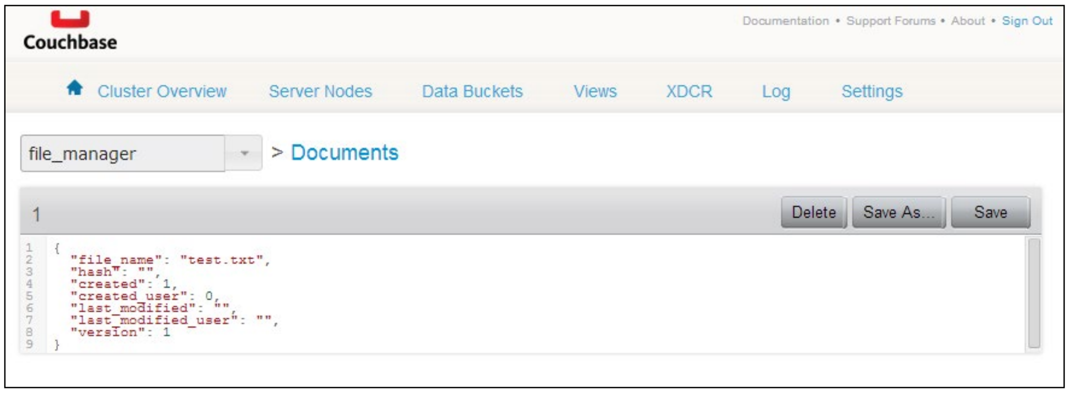
☐ Enable

In our example, we're working on a single machine, so replicas (also known as replication in database vernacular) are not supported. We've named it `file_manager`, but this can obviously be called anything that makes sense.

We're also keeping our data usage pretty low—there's no need for much more than 256 MB of memory when we're storing file operations and logging older ones. In other words, we're not necessarily concerned with keeping the modification history of `test.txt` in memory forever.

We'll also stick with Couchbase for a storage engine equivalent, although you can flip back and forth with memcache(d) without much noticeable change.

Let's start by creating a seed document: one we'll delete later, but that will represent the schema of our data store. We can create this document with an arbitrary JSON structured object, as shown in the following screenshot:



```
{
  "file_name": "test.txt",
  "hash": "",
  "created": 1,
  "created_user": 0,
  "last_modified": "",
  "last_modified_user": "",
  "version": 1
}
```

Since everything stored in this data store should be valid JSON, we can mix and match strings, integers, bools, arrays, and objects. This affords us some flexibility in what data we're using. The following is an example document:

```
{
  "file_name": "test.txt",
  "hash": "",
  "created": 1,
  "created_user": 0,
  "last_modified": "",
  "last_modified_user": "",
  "revisions": [],
  "version": 1
}
```

Monitoring filesystem changes

When it came to NoSQL options, we had a vast variety of solutions at our disposal. This is not the case when it comes to applications that monitor filesystem changes. While Linux flavors have a fairly good built-in solution in inotify, this does restrict the portability of the application.

So it's incredibly helpful that a cross-platform library for handling this exists in Chris Howey's fsnotify.

Fsnotify works on Linux, OSX, and Windows and allows us to detect when files in any given directory are created, deleted, modified, or renamed, which is more than enough for our purposes.

Implementing fsnotify couldn't be easier, either. Best of all it's all non-blocking, so if we throw the listener behind a goroutine, we can have this run as part of the primary server application code.

The following code shows a simple directory listener:

```
package main

import (
    "github.com/howeyc/fsnotify"
  "fmt"
  "log""
)


func main() {

    scriptDone := make(chan bool)
    dirSpy, err := fsnotify.NewWatcher()
    if err != nil {
        log.Fatal(err)
    }

    go func() {
        for {
            select {
            case fileChange := <-dirSpy.Event:
                log.Println("Something happened to a file:",
                  fileChange)
            case err := <-dirSpy.Error:
                log.Println("Error with fsnotify:", err)
            }
        }
    }()

    err = dirSpy.Watch("/mnt/sharedir")
    if err != nil {
      fmt.Println(err)
    }

    <-scriptDone

    dirSpy.Close()
}
```

Managing logfiles

Like many basic features in a developer's toolbox, Go provides a fairly complete solution built-in for logging. It handles many of the basics, such as creating timestamp-marked log items and saving to disk or to console.

One thing the basic package misses out on is built-in formatting and log rotation, which are key requirements for our file manager application.

Remember that key requirements for our application include the ability to work seamlessly in our concurrent environment and be ready to scale to a distributed network if need be. This is where the fine **log4go** application comes in handy. Log4go allows logging to file, console, and memory and handles log rotation inherently.

Log4go can be found at `https://code.google.com/p/log4go/`.
To install Log4go, run the following command:
`go get code.google.com/p/log4go`

Creating a logfile that handles warnings, notices, debug information, and critical errors is simple and appending log rotation to that is similarly simple, as shown in the following code:

```
package main

import
(
  logger "code.google.com/p/log4go"
)
func main() {
  logMech := make(logger.Logger);
  logMech.AddFilter("stdout", logger.DEBUG,
    logger.NewConsoleLogWriter())

  fileLog := logger.NewFileLogWriter("log_manager.log", false)
  fileLog.SetFormat("[%D %T] [%L] (%S) %M")
  fileLog.SetRotate(true)
  fileLog.SetRotateSize(256)
  fileLog.SetRotateLines(20)
  fileLog.SetRotateDaily(true)
  logMech.AddFilter("file", logger.FINE, fileLog)

  logMech.Trace("Received message: %s)", "All is well")
  logMech.Info("Message received: ", "debug!")
  logMech.Error("Oh no!","Something Broke")
}
```

Handling configuration files

When it comes to configuration files and parsing them, you have a lot of options, from simple to complicated.

We could, of course, simply store what we want in JSON, but that format is a little tricky to work directly for humans—it will require escaping characters and so on, which makes it vulnerable to errors.

Instead, we'll keep things simple by using a standard `ini config` file library in gcfg, which handles `gitconfig` files and traditional, old school `.ini` format, as shown in the following code snippet:

```
[revisions]
count = 2
revisionsuffix = .rev
lockfiles = false

[logs]
rotatelength = 86400

[alarms]
emails = sysadmin@example.com,ceo@example.com
```

You can find gcfg at `https://code.google.com/p/gcfg/`.

Essentially, this library takes the values of a config file and pushes them into a struct in Go. An example of how we'll do that is as follows:

```
package main

import
(
  "fmt"
  "code.google.com/p/gcfg"
)

type Configuration struct {
  Revisions struct {
    Count int
    Revisionsuffix string
    Lockfiles bool
  }
  Logs struct {
    Rotatelength int
  }
  Alarms struct {
    Emails string
  }
}

func main() {
  configFile := Configuration{}
  err := gcfg.ReadFileInto(&configFile, "example.ini")
  if err != nil {
    fmt.Println("Error",err)
  }
  fmt.Println("Rotation duration:",configFile.Logs.Rotatelength)
}
```

Detecting file changes

Now we need to focus on our file listener. You may recall this is the part of the application that will accept client connections from our web server and our backup application and announce any changes to files.

The basic flow of this part is as follows:

1. Listen for changes to files in a goroutine.
2. Accept connections and add to the pool in a goroutine.
3. If any changes are detected, announce them to the entire pool.

All three happen concurrently, and the first and the third can happen without any connections in the pool, although we assume there will be a connection that is always on with both our web server and our backup application.

Another critical role the file listener will fulfill is analyzing the directory on first load and reconciling it with our data store in Couchbase. Since the Go Couchbase library handles the get, update, and add operations, we won't need any custom views. In the following code, we'll examine the file listener process and show how we listen on a folder for changes:

```
package main

import
(
  "fmt"
  "github.com/howeyc/fsnotify"
  "net"
  "time"
  "io"
  "io/ioutil"
  "github.com/couchbaselabs/go-couchbase"
  "crypto/md5"
  "encoding/hex"
  "encoding/json"
  "strings"

)

var listenFolder = "mnt/sharedir"

type Client struct {
  ID int
  Connection *net.Conn
}
```

Here, we've declared our shared folder as well as a connecting `Client` struct. In this application, `Client` is either a web listener or a backup listener, and we'll pass messages in one direction using the following JSON-encoded structure:

```
type File struct {
  Hash string "json:hash"
  Name string "json:file_name"
  Created int64 "json:created"
  CreatedUser  int "json:created_user"
  LastModified int64 "json:last_modified"
  LastModifiedUser int "json:last_modified_user"
  Revisions int "json:revisions"
  Version int "json:version"
}
```

If this looks familiar, it could be because it's also the example document format we set up initially.

> If you're not familiar with the syntactical sugar expressed earlier, these are known as struct tags. A tag is just a piece of additional metadata that can be applied to a struct field for key/value lookups via the `reflect` package. In this case, they're used to map our struct fields to JSON fields.

Let's first look at our overall `Message struct`:

```
type Message struct {
  Hash string "json:hash"
  Action string "json:action"
  Location string "json:location"
  Name string "json:name"
  Version int "json:version"
}
```

We compartmentalize our file into a message, which alerts our other two processes of changes:

```
func generateHash(name string) string {

  hash := md5.New()
  io.WriteString(hash,name)
  hashString := hex.EncodeToString(hash.Sum(nil))

  return hashString
}
```

This is a somewhat unreliable method to generate a hash reference to a file and will fail if a filename changes. However, it allows us to keep track of files that are created, deleted, or modified.

Sending changes to clients

Here is the broadcast message that goes to all existing connections. We pass along our JSON-encoded `Message` struct with the current version, the current location, and the hash for reference. Our other servers will then react accordingly:

```
func alertServers(hash string, name string, action string, location
string, version int) {
  msg :=
    Message{Hash:hash,Action:action,Location:location,Name:name,
    Version:version}
  msgJSON,_ := json.Marshal(msg)

  fmt.Println(string(msgJSON))

  for i := range Clients {
    fmt.Println("Sending to clients")
    fmt.Fprintln(*Clients[i].Connection,string(msgJSON))
  }
}
```

Our backup server will create a copy of that file with the `.[VERSION]` extension in the backup folder.

Our web server will simply alert the user via our web interface that the file has changed:

```
func startServer(listener net.Listener) {
  for {
    conn,err := listener.Accept()
    if err != nil {

    }
    currentClient := Client{ ID: 1, Connection: &conn}
    Clients = append(Clients,currentClient)
      for i:= range Clients {
        fmt.Println("Client",Clients[i].ID)
      }
  }

}
```

Does this code look familiar? We've taken almost our exact chat server `Client` handler and brought it over here nearly intact:

```
func removeFile(name string, bucket *couchbase.Bucket) {
  bucket.Delete(generateHash(name))
}
```

The `removeFile` function does one thing only and that's removing the file from our Couchbase data store. As it's reactive, we don't need to do anything on the file-server side because the file is already deleted. Also, there's no need to delete any backups, as this allows us to recover. Next, let's look at our function that updates an existing file:

```
func updateExistingFile(name string, bucket *couchbase.Bucket) int {
  fmt.Println(name,"updated")
  hashString := generateHash(name)

  thisFile := Files[hashString]
  thisFile.Hash = hashString
  thisFile.Name = name
  thisFile.Version = thisFile.Version + 1
  thisFile.LastModified = time.Now().Unix()
  Files[hashString] = thisFile
  bucket.Set(hashString,0,Files[hashString])
  return thisFile.Version
}
```

This function essentially overwrites any values in Couchbase with new ones, copying an existing `File` struct and changing the `LastModified` date:

```
func evalFile(event *fsnotify.FileEvent, bucket *couchbase.Bucket) {
  fmt.Println(event.Name,"changed")
  create := event.IsCreate()
  fileComponents := strings.Split(event.Name,"\\")
  fileComponentSize := len(fileComponents)
  trueFileName := fileComponents[fileComponentSize-1]
  hashString := generateHash(trueFileName)

  if create == true {
    updateFile(trueFileName,bucket)
    alertServers(hashString,event.Name,"CREATE",event.Name,0)
  }
  delete := event.IsDelete()
  if delete == true {
    removeFile(trueFileName,bucket)
    alertServers(hashString,event.Name,"DELETE",event.Name,0)
  }
  modify := event.IsModify()
  if modify == true {
    newVersion := updateExistingFile(trueFileName,bucket)
    fmt.Println(newVersion)
    alertServers(hashString,trueFileName,"MODIFY",event.Name,
      newVersion)
  }
  rename := event.IsRename()
  if rename == true {


  }
}
```

Here, we react to any changes to the filesystem in our watched directory. We aren't reacting to renames, but you can handle those as well. Here's how we'd approach the general `updateFile` function:

```
func updateFile(name string, bucket *couchbase.Bucket) {
  thisFile := File{}
  hashString := generateHash(name)

  thisFile.Hash = hashString
  thisFile.Name = name
  thisFile.Created = time.Now().Unix()
  thisFile.CreatedUser = 0
  thisFile.LastModified = time.Now().Unix()
  thisFile.LastModifiedUser = 0
  thisFile.Revisions = 0
  thisFile.Version = 1

  Files[hashString] = thisFile

  checkFile := File{}
  err := bucket.Get(hashString,&checkFile)
  if err != nil {
    fmt.Println("New File Added",name)
    bucket.Set(hashString,0,thisFile)
  }
}
```

Checking records against Couchbase

When it comes to checking for existing records against Couchbase, we check whether a hash exists in our Couchbase bucket. If it doesn't, we create it. If it does, we do nothing. To handle shutdowns more robustly, we should also ingest existing records into our application. The code for doing this is as follows:

```
var Clients []Client
var Files map[string] File


func main() {
  Files = make(map[string]File)
  endScript := make(chan bool)

  couchbaseClient, err := couchbase.Connect("http://localhost:8091/")
    if err != nil {
      fmt.Println("Error connecting to Couchbase", err)
    }
  pool, err := couchbaseClient.GetPool("default")
    if err != nil {
      fmt.Println("Error getting pool",err)
    }
  bucket, err := pool.GetBucket("file_manager")
    if err != nil {
      fmt.Println("Error getting bucket",err)
    }


  files, _ := ioutil.ReadDir(listenFolder)
  for _, file := range files {
    updateFile(file.Name(),bucket)
  }

    dirSpy, err := fsnotify.NewWatcher()
    defer dirSpy.Close()

  listener, err := net.Listen("tcp", ":9000")
  if err != nil {
    fmt.Println ("Could not start server!",err)
  }

  go func() {
        for {
            select {
            case ev := <-dirSpy.Event:
                evalFile(ev,bucket)
            case err := <-dirSpy.Error:
                fmt.Println("error:", err)
            }
        }
    }()
    err = dirSpy.Watch(listenFolder)
  startServer(listener)

  <-endScript
}
```

Finally, `main()` handles setting up our connections and goroutines, including a file watcher, our TCP server, and connecting to Couchbase.

Now, let's look at another step in the whole process where we will automatically create backups of our modified files.

Backing up our files

Since we're sending our commands on the wire, so to speak, our backup process needs to listen on that wire and respond with any changes. Given that modifications will be sent via localhost, we should have minimal latency on both the network and the file side.

We'll also return some information as to what happened with the file, although at this point we're not doing much with that information. The code for this is as follows:

```
package main

import
(
  "fmt"
  "net"
  "io"
  "os"
  "strconv"
  "encoding/json"
)

var backupFolder = "mnt/backup/"
```

Note that we have a separate folder for backups, in this case, on a Windows machine. If we happen to accidentally use the same directory, we run the risk of infinitely duplicating and backing up files. In the following code snippet, we'll look at the `Message` struct itself and the `backup` function, the core of this part of the application:

```
type Message struct {
  Hash string "json:hash"
  Action string "json:action"
  Location string "json:location"
  Name string "json:name"
  Version int "json:version"
}

func backup (location string, name string, version int) {

  newFileName := backupFolder + name + "." +
    strconv.FormatInt(int64(version),10)
  fmt.Println(newFileName)
  org,_ := os.Open(location)
  defer org.Close()
  cpy,_ := os.Create(newFileName)
  defer cpy.Close()
  io.Copy(cpy,org)
}
```

Here is our basic file operation. Go doesn't have a one-step copy function; instead you need to create a file and then copy the contents of another file into it with `io.Copy`:

```
func listen(conn net.Conn) {
  for {

      messBuff := make([]byte,1024)
    n, err := conn.Read(messBuff)
    if err != nil {

    }



    resultMessage := Message{}
    json.Unmarshal(messBuff[:n],&resultMessage)

    if resultMessage.Action == "MODIFY" {
      fmt.Println("Back up file",resultMessage.Location)
      newVersion := resultMessage.Version + 1
      backup(resultMessage.Location,resultMessage.Name,newVersion)
    }

  }

}
```

This code is nearly verbatim for our chat client's `listen()` function, except that we take the contents of the streamed JSON data, unmarshal it, and convert it to a `Message{}` struct and then a `File{}` struct. Finally, let's look at the `main` function and TCP initialization:

```
func main() {
  endBackup := make(chan bool)
  conn, err := net.Dial("tcp","127.0.0.1:9000")
  if err != nil {
    fmt.Println("Could not connect to File Listener!")
  }
  go listen(conn)

  <- endBackup
}
```

Designing our web interface

To interact with the filesystem, we'll want an interface that displays all of the current files with the version, last modified time, and alerts to changes, and allows drag-and-drop creation/replacement of files.

Getting a list of files will be simple, as we'll grab them directly from our `file_manager` Couchbase bucket. Changes will be sent through our file manager process via TCP, which will trigger an API call, illuminating changes to the file for our web user.

A few of the methods we've used here are duplicates of the ones we used in the backup process and could certainly benefit from some consolidation; still, the following is the code for the web server, which allows uploads and shows notifications for changes:

```
package main

import
(
  "net"
  "net/http"
  "html/template"
  "log"
  "io"
  "os"
  "io/ioutil"
  "github.com/couchbaselabs/go-couchbase"
  "time"
  "fmt"
  "crypto/md5"
  "encoding/hex"
  "encoding/json"
)

type File struct {
  Hash string "json:hash"
  Name string "json:file_name"
  Created int64 "json:created"
  CreatedUser  int "json:created_user"
  LastModified int64 "json:last_modified"
  LastModifiedUser int "json:last_modified_user"
  Revisions int "json:revisions"
  Version int "json:version"
}
```

This, for example, is the same `File` struct we use in both the file listener and the backup process:

```
type Page struct {
  Title string
  Files map[string] File
}
```

Our `Page` struct represents generic web data that gets converted into corresponding variables for our web page's template:

```
type ItemWrapper struct {

  Items []File
  CurrentTime int64
  PreviousTime int64

}

type Message struct {
  Hash string "json:hash"
  Action string "json:action"
  Location string "json:location"
  Name string "json:name"
  Version int "json:version"
}
```

The `ItemWrapper` struct is simply a JSON wrapper that will keep an array that's converted from our `Files` struct. This is essential to loop through the API's JSON on the client side. Our `Message` struct is a duplicate of the same type we saw in our file listener and backup processes. In the following code snippet, we'll dictate some general configuration variables and our hash generation function:

```
var listenFolder = "/wamp/www/shared/"
var Files map[string] File
var webTemplate = template.Must(template.ParseFiles("ch8_html.html"))
var fileChange chan File
var lastChecked int64

func generateHash(name string) string {

  hash := md5.New()
  io.WriteString(hash,name)
  hashString := hex.EncodeToString(hash.Sum(nil))

  return hashString
}
```

Our md5 hashing method is the same for this application as well. It's worth noting that we derive a lastChecked variable that is the Unix-style timestamp from each time we get a signal from our file listener. We use this to compare with file changes on the client side to know whether to alert the user on the Web. Let's now look at the updateFile function for the web interface:

```
func updateFile(name string, bucket *couchbase.Bucket) {
  thisFile := File{}
  hashString := generateHash(name)

  thisFile.Hash = hashString
  thisFile.Name = name
  thisFile.Created = time.Now().Unix()
  thisFile.CreatedUser = 0
  thisFile.LastModified = time.Now().Unix()
  thisFile.LastModifiedUser = 0
  thisFile.Revisions = 0
  thisFile.Version = 1

  Files[hashString] = thisFile

  checkFile := File{}
  err := bucket.Get(hashString,&checkFile)
  if err != nil {
    fmt.Println("New File Added",name)
    bucket.Set(hashString,0,thisFile)
  }else {
    Files[hashString] = checkFile
  }
}
```

This is the same function as our backup process, except that instead of creating a duplicate file, we simply overwrite our internal File struct to allow it represent its updated LastModified value when the /api is next called. And as with our last example, let's check out the listen() function:

```
func listen(conn net.Conn) {
  for {

      messBuff := make([]byte,1024)
    n, err := conn.Read(messBuff)
    if err != nil {
    }
    message := string(messBuff[:n])
    message = message[0:]

    resultMessage := Message{}
    json.Unmarshal(messBuff[:n],&resultMessage)

    updateHash := resultMessage.Hash
    tmp := Files[updateHash]
    tmp.LastModified = time.Now().Unix()
    Files[updateHash] = tmp
  }

}
```

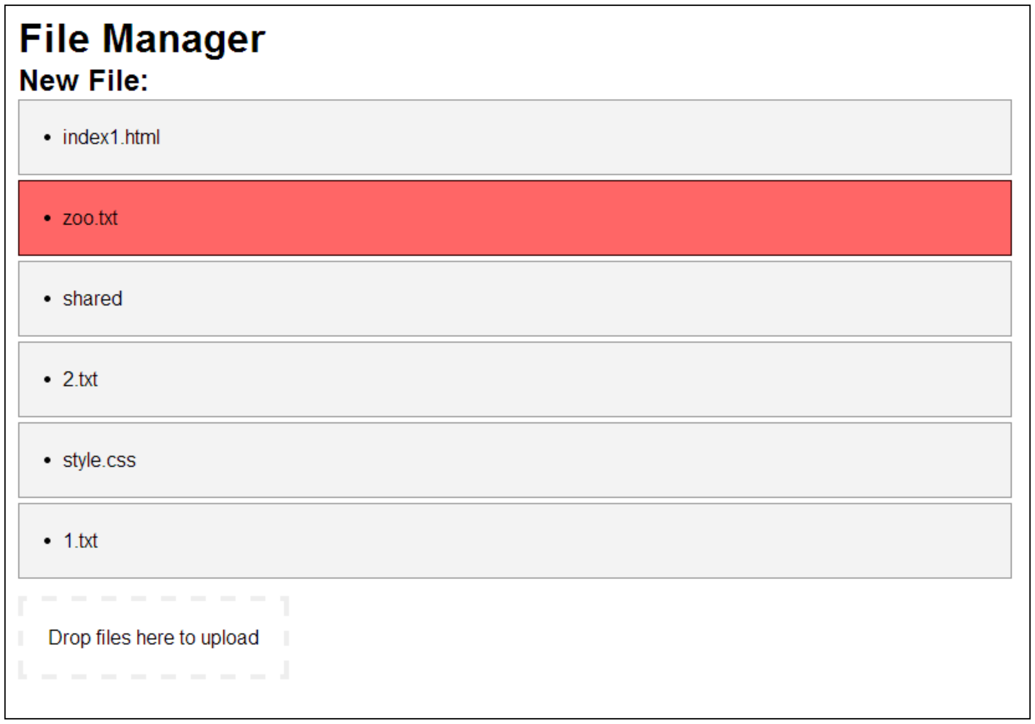
Here is where our message is read, unmarshalled, and set to its hashed map's key. This will create a file if it doesn't exist or update our current one if it does. Next, we'll look at the `main()` function, which sets up our application and the web server:

```
    func main() {
      lastChecked := time.Now().Unix()
      Files = make(map[string]File)
      fileChange = make(chan File)
      couchbaseClient, err := couchbase.Connect("http://localhost:8091/")
        if err != nil {
          fmt.Println("Error connecting to Couchbase", err)
        }
      pool, err := couchbaseClient.GetPool("default")
        if err != nil {
          fmt.Println("Error getting pool",err)
        }
      bucket, err := pool.GetBucket("file_manager")
        if err != nil {
          fmt.Println("Error getting bucket",err)
        }

      files, _ := ioutil.ReadDir(listenFolder)
      for _, file := range files {
        updateFile(file.Name(),bucket)
      }

      conn, err := net.Dial("tcp","127.0.0.1:9000")
      if err != nil {
        fmt.Println("Could not connect to File Listener!")
      }
```

```
  go listen(conn)


  http.HandleFunc("/api", func(w http.ResponseWriter, r
    *http.Request) {
    apiOutput := ItemWrapper{}
    apiOutput.PreviousTime = lastChecked
    lastChecked = time.Now().Unix()
    apiOutput.CurrentTime = lastChecked

    for i:= range Files {
      apiOutput.Items = append(apiOutput.Items,Files[i])
    }
    output,_ := json.Marshal(apiOutput)
    fmt.Fprintln(w,string(output))

  })
  http.HandleFunc("/", func(w http.ResponseWriter, r
    *http.Request) {
    output := Page{Files:Files,Title:"File Manager"}
    tmp, _ := template.ParseFiles("ch8_html.html")
    tmp.Execute(w, output)
  })
  http.HandleFunc("/upload", func(w http.ResponseWriter, r
    *http.Request) {
    err := r.ParseMultipartForm(10000000)
    if err != nil {
      return
    }
    form := r.MultipartForm

    files := form.File["file"]
    for i, _ := range files {
      newFileName := listenFolder + files[i].Filename
      org,_:= files[i].Open()
      defer org.Close()
      cpy,_ := os.Create(newFileName)
      defer cpy.Close()
      io.Copy(cpy,org)
    }
  })

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

}
```

In our web server component, `main()` takes control of setting up the connection to the file listener and Couchbase and creating a web server (with related routing).

If you upload a file by dragging it to the **Drop files here to upload** box, within a few seconds you'll see that the file is noted as changed in the web interface, as shown in the following screenshot:


File Manager
New File:

- index1.html
- zoo.txt
- shared
- 2.txt
- style.css
- 1.txt

Drop files here to upload

We haven't included the code for the client side of the web interface; the key points, though, are retrieval via an API. We used a JavaScript library called `Dropzone.js` that allows a drag-and-drop upload, and jQuery for API access.

Reverting a file's history – command line

The final component we'd like to add to this application suite is a command-line file revision process. We can keep this one fairly simple, as we know where a file is located, where its backups are located, and how to replace the former with the latter. As with before, we have some global configuration variables and a replication of our `generateHash()` function:

```
var liveFolder = "/mnt/sharedir "
var backupFolder = "/mnt/backup

func generateHash(name string) string {

  hash := md5.New()
  io.WriteString(hash,name)
  hashString := hex.EncodeToString(hash.Sum(nil))

  return hashString
}

func main() {
  revision := flag.Int("r",0,"Number of versions back")
  fileName := flag.String("f","","File Name")
  flag.Parse()

  if *fileName == "" {

    fmt.Println("Provide a file name to use!")
    os.Exit(0)
  }


  couchbaseClient, err := couchbase.Connect("http://localhost:8091/")
    if err != nil {
      fmt.Println("Error connecting to Couchbase", err)
    }
  pool, err := couchbaseClient.GetPool("default")
    if err != nil {
      fmt.Println("Error getting pool",err)
    }
  bucket, err := pool.GetBucket("file_manager")
    if err != nil {
      fmt.Println("Error getting bucket",err)
    }

  hashString := generateHash(*fileName)
  checkFile := File{}
  bucketerr := bucket.Get(hashString,&checkFile)
  if bucketerr != nil {
  }else {
    backupLocation := backupFolder + checkFile.Name + "." +
      strconv.FormatInt(int64(checkFile.Version-*revision),10)
    newLocation := liveFolder + checkFile.Name
    fmt.Println(backupLocation)
    org,_ := os.Open(backupLocation)
      defer org.Close()
    cpy,_ := os.Create(newLocation)
      defer cpy.Close()
    io.Copy(cpy,org)
    fmt.Println("Revision complete")
  }

}
```

This application accepts up to two parameters:

- `-f`: This denotes the filename
- `-r`: This denotes the number of versions to revert

Note that this itself creates a new version and thus a backup, so -2 would need to become -3, and then -6, and so on in order to continuously back up recursively.

As an example, if you wished to revert `example.txt` back three versions, you could use the following command:

```
fileversion -f example.txt -r -3
```

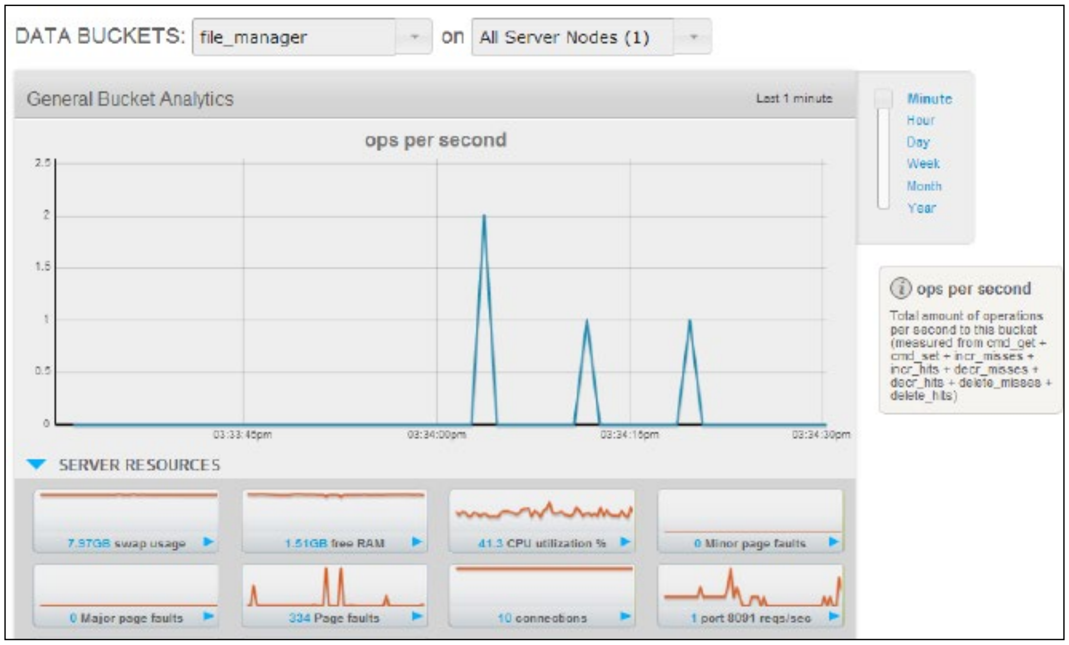
Using Go in daemons and as a service

A minor note on running something like this part of the application—you'll ideally wish to keep these applications as active, restartable services instead of standalone, manually executed background processes. Doing so will allow you to keep the application active and manage its life from external or server processes.

This sort of application suite would be best suited on a Linux box (or boxes) and managed with a daemon manager such as daemontools or Ubuntu's built-in Upstart service. The reason for this is that any long-term downtime can result in lost data and inconsistency. Even storing file data details in the memory (Couchbase and memcached) provides a vulnerability for lost data.

Checking the health of our server

Of the many ways to check general server health, we're in a good position here without having to build our own system, thanks in great part to Couchbase itself. If you visit the Couchbase web admin, under your cluster, server, and bucket views, clicking on any will present some real-time statistics, as shown in the following screenshot:



ops per second

Total amount of operations per second to this bucket (measured from cmd_get + cmd_set + incr_misses + incr_hits + decr_misses + decr_hits + delete_misses + delete_hits)

These areas are also available via REST if you wish to include them in the application to make your logging and error handling more comprehensive.

Summary

We now have a top to bottom application suite that is highly concurrent, ropes in several third-party libraries, and mitigates potential failures with logging and catastrophe recovery.

At this point, you should have no issue constructing a complex package of software with a focus on maintaining concurrency, reliability, and performance in Go. Our file monitoring application can be easily modified to do more, use alternative services, or scale to a robust, distributed environment.

In the next chapter, we'll take a closer look at testing our concurrency and throughput, explore the value of panic and recover, as well as dealing with logging vital information and errors in a safe, concurrent manner in Go.

9
Logging and Testing Concurrency in Go

At this stage, you should be fairly comfortable with concurrency in Go and should be able to implement basic goroutines and concurrent mechanisms with ease.

We have also dabbled in some distributed concurrency patterns that are managed not only through the application itself, but also through third-party data stores for networked applications that operate concurrently in congress.

Earlier in this book, we examined some preliminary and basic testing and logging. We looked at the simpler implementations of Go's internal test tool, performed some race condition testing using the race tool, and performed some rudimentary load and performance testing.

However, there's much more to be looked at here, particularly as it relates to the potential black hole of concurrent code—we've seen unexpected behavior among code that runs in goroutines and is non-blocking.

In this chapter, we'll further investigate load and performance testing, look at unit testing in Go, and experiment with more advanced tests and debugging. We'll also look at best practices for logging and reporting, as well as take a closer look at panicking and recovering.

Lastly, we'll want to see how all of these things can be applied not just to our standalone concurrent code, but also to distributed systems.

Along the way, we'll introduce a couple of frameworks for unit testing in a variety of different styles.

Handling errors and logging

Though we haven't specifically mentioned it, the idiomatic nature of error handling in Go makes debugging naturally easier by mandate.

One good practice for any large-scale function inside Go code is to return an error as a return value—for many smaller methods and functions, this is potentially burdensome and unnecessary. Still, it's a matter for consideration whenever we're building something that involves a lot of moving pieces.

For example, consider a simple `Add()` function:

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

If we wish to follow the general rule of "always return an error value", we may be tempted to convert this function to the following code:

```
package main
import
(
  "fmt"
  "errors"
  "reflect"
)

func Add(x int, y int) (int, error) {
  var err error

  xType := reflect.TypeOf(x).Kind()
  yType := reflect.TypeOf(y).Kind()
  if xType != reflect.Int || yType != reflect.Int {
    fmt.Println(xType)
    err = errors.New("Incorrect type for integer a or b!")
  }
  return x + y, err
}

func main() {

  sum,err := Add("foo",2)
  if err != nil {
    fmt.Println("Error",err)
  }
  fmt.Println(sum)
}
```

You can see that we're (very poorly) reinventing the wheel. Go's internal compiler kills this long before we ever see it. So, we should focus on things that the compiler may not catch and that can cause unexpected behavior in our applications, particularly when it comes to channels and listeners.

The takeaway is to let Go handle the errors that the compiler would handle, unless you wish to handle the exceptions yourself, without causing the compiler specific grief. In the absence of true polymorphism, this is often cumbersome and requires the invocation of interfaces, as shown in the following code:

```
type Alpha struct {

}

type Numeric struct {

}
```

You may recall that creating interfaces and structs allows us to route our function calls separately based on type. This is shown in the following code:

```
func (a Alpha) Add(x string, y string) (string, error) {
  var err error
  xType := reflect.TypeOf(x).Kind()
  yType := reflect.TypeOf(y).Kind()
  if xType != reflect.String || yType != reflect.String {
    err = errors.New("Incorrect type for strings a or b!")
  }
  finalString := x + y
  return finalString, err
}

func (n Numeric) Add(x int, y int) (int, error) {
  var err error

  xType := reflect.TypeOf(x).Kind()
  yType := reflect.TypeOf(y).Kind()
  if xType != reflect.Int || yType != reflect.Int {
    err = errors.New("Incorrect type for integer a or b!")
  }
  return x + y, err
}
```

```
func main() {
  n1 := Numeric{}
  a1 := Alpha{}
  z,err := n1.Add(5,2)
  if err != nil {
    log.Println("Error",err)
  }
  log.Println(z)

  y,err := a1.Add("super","lative")
  if err != nil {
    log.Println("Error",err)
  }
  log.Println(y)
}
```

This still reports what will eventually be caught by the compiler, but also handles some form of error on what the compiler cannot see: external input. We're routing our `Add()` function through an interface, which provides some additional standardization by directing the struct's parameters and methods more explicitly.

If, for example, we take user input for our values and need to evaluate the type of that input, we may wish to report an error in this way as the compiler will never know that our code can accept the wrong type.

Breaking out goroutine logs

One way of handling messaging and logging that keeps a focus on concurrency and isolation is to shackle our goroutine with its own logger that will keep everything separate from the other goroutines.

At this point, we should note that this may not scale—that is, it may at some point become expensive to create thousands or tens of thousands of goroutines that have their own loggers, but at a minimal size, this is totally doable and manageable.

To do this logging individually, we'll want to tie a `Logger` instance to each goroutine, as shown in the following code:

```
package main

import
(
  "log"
  "os"
  "strconv"
)

const totalGoroutines = 5

type Worker struct {
  wLog *log.Logger
  Name string
}
```

We'll create a generic `Worker` struct that will ironically do no work (at least not in this example) other than hold onto its own `Logger` object. The code is as follows:

```
func main() {
  done := make(chan bool)

  for i:=0; i< totalGoroutines; i++ {

    myWorker := Worker{}
    myWorker.Name = "Goroutine " + strconv.FormatInt(int64(i),10) + ""
    myWorker.wLog = log.New(os.Stderr, myWorker.Name, 1)
    go func(w *Worker) {

        w.wLog.Print("Hmm")

        done <- true
    }(&myWorker)
  }
```

Each goroutine is saddled with its own log routine through `Worker`. While we're spitting our output straight to the console, this is largely unnecessary. However, if we want to siphon each to its own logfile, we could do so by using the following code:

```
  log.Println("...")

  <- done
}
```

Using the LiteIDE for richer and easier debugging

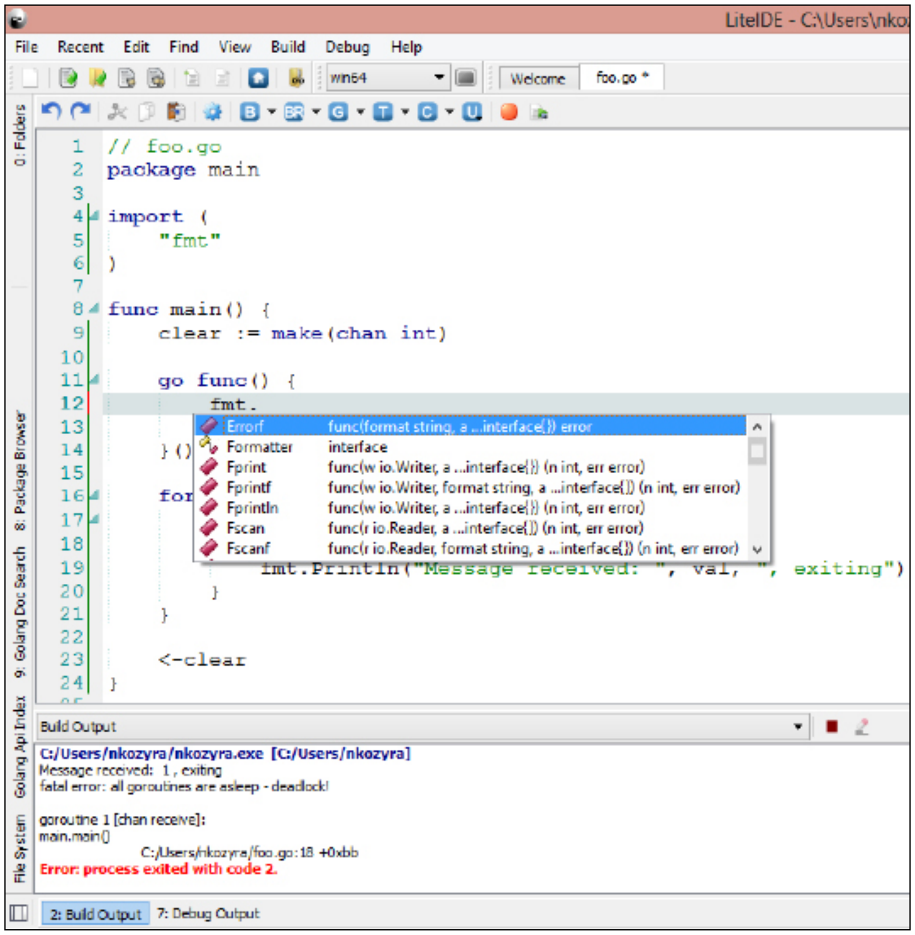
In the earlier chapters of this book, we briefly addressed IDEs and gave a few examples of IDEs that have a tight integration with Go.

As we're examining logging and debugging, there's one IDE we previously and specifically didn't mention before, primarily because it's intended for a very small selection of languages—namely, Go and Lua. However, if you end up working primarily or exclusively in Go, you'll find it absolutely essential, primarily as it relates to debugging, logging, and feedback capabilities.

LiteIDE is cross-platform and works well on OS X, Linux, and Windows. The number of debugging and testing benefits it presents in a GUI form are invaluable, particularly if you're already very comfortable with Go. That last part is important because developers often benefit most from "learning the hard way" before diving in with tools that simplify the programming process. It's almost always better to know how and why something works or doesn't work at the core before being presented with pretty icons, menus, and pop-up windows. Having said that, LiteIDE is a fantastic, free tool for the advanced Go programmer.

By formalizing a lot of the tools and error reporting from Go, we can easily plow through some of the more vexing debugging tasks by seeing them onscreen.

LiteIDE also brings context awareness, code completion, `go fmt`, and more into our workspace. You can imagine how an IDE tuned specifically for Go can help you keep your code clean and bug free. Refer to the following screenshot:



```
// foo.go
package main

import (
    "fmt"
)

func main() {
    clear := make(chan int)

    go func() {
        fmt.

    } ()

    for

            fmt.Println("Message received: ", val, ", exiting")
        }
    }

    <-clear
}
```

```
Errorf      func(format string, a ...interface{}) error
Formatter   interface
Fprint      func(w io.Writer, a ...interface{}) (n int, err error)
Fprintf     func(w io.Writer, format string, a ...interface{}) (n int, err error)
Fprintln    func(w io.Writer, a ...interface{}) (n int, err error)
Fscan       func(r io.Reader, a ...interface{}) (n int, err error)
Fscanf      func(r io.Reader, format string, a ...interface{}) (n int, err error)
```

```
C:/Users/nkozyra/nkozyra.exe  [C:/Users/nkozyra]
Message received:  1 , exiting
fatal error: all goroutines are asleep - deadlock!

goroutine 1 [chan receive]:
main.main()
        C:/Users/nkozyra/foo.go:18 +0xbb
Error: process exited with code 2.
```

LiteIDE showing output and automatic code completion on Windows

[LiteIDE for Linux, OS X, and Windows can be found at `https://code.google.com/p/liteide/`.]

Sending errors to screen

Throughout this book, we have usually handled soft errors, warnings, and general messages with the `fmt.Println` syntax by sending a message to the console.

While this is quick and easy for demonstration purposes, it's probably ideal to use the `log` package to handle these sorts of things. This is because we have more versatility, as `log` relates to where we want our messages to end up.

As for our purposes so far, the messages are ethereal. Switching out a simple `Println` statement to `Logger` is extremely simple.

We've been relaying messages before using the following line of code:

```
fmt.Println("Horrible error:",err)
```

You'll notice the change to `Logger` proves pretty similar:

```
myLogger.Println("Horrible error:", err)
```

This is especially useful for goroutines, as we can create either a global `Logger` interface that can be accessed anywhere or pass the logger's reference to individual goroutines and ensure our logging is handled concurrently.

One consideration for having a single logger for use across our entire application is the possibility that we may want to log individual processes separately for clarity in analysis. We'll talk a bit more about that later in this chapter.

To replicate passing messages to the command line, we can simply use the following line of code:

```
log.Print("Message")
```

With defaults to `stdout` as its `io.writer`—recall that we can set any `io.writer` as the log's destination.

However, we will also want to be able to log to file quickly and easily. After all, any application running in the background or as a daemon will need to have something a little more permanent.

Logging errors to file

There are a lot of ways to send an error to a logfile—we can, after all, handle this with built-in file operation OS calls. In fact, this is what many people do.

However, the `log` package offers some standardization and potential symbiosis between the command-line feedback and more permanent storage of errors, warnings, and general information.

The simplest way to do this is to open a file using the `os.OpenFile()` method (and not the `os.Open()` method) and pass that reference to our log instantiation as `io.Writer`.

Let's take a look at such functionality in the following example:

```
package main

import (
  "log"
  "os"
)

func main() {
  logFile, _ := os.OpenFile("/var/www/test.log", os.O_RDWR, 0755)

  log.SetOutput(logFile)
  log.Println("Sending an entry to log!")

  logFile.Close()
}
```

In our preceding goroutine package, we could assign each goroutine its own file and pass a file reference as an io Writer (we'll need to have write access to the destination folder). The code is as follows:

```
  for i:=0; i< totalGoroutines; i++ {

    myWorker := Worker{}
    myWorker.Name = "Goroutine " + strconv.FormatInt(int64(i),10)
      + ""
    myWorker.FileName = "/var/www/"+strconv.FormatInt(int64(i),10)
      + ".log"
    tmpFile,_ :=   os.OpenFile(myWorker.FileName, os.O_CREATE,
      0755)
    myWorker.File = tmpFile
    myWorker.wLog = log.New(myWorker.File, myWorker.Name, 1)
    go func(w *Worker) {

        w.wLog.Print("Hmm")

        done <- true
    }(&myWorker)
  }
```

Logging errors to memory

When we talk about logging errors to memory, we're really referring to a data store, although there's certainly no reason other than volatility and limited resources to reject logging to memory as a viable option.

While we'll look at a more direct way to handle networked logging through another package in the next section, let's delineate our various application errors in a concurrent, distributed system without a lot of hassle. The idea is to use shared memory (such as Memcached or a shared memory data store) to pass our log messages.

While these will technically still be logfiles (most data stores keep individual records or documents as JSON-encoded hard files), it has a distinctively different feel than traditional logging.

Going back to our old friend from the previous chapter—CouchDB—passing our logging messages to a central server can be done almost effortlessly, and it allows us to track not just individual machines, but their individual concurrent goroutines. The code is as follows:

```
package main

import
(
  "github.com/couchbaselabs/go-couchbase"
  "io"
  "time"
  "fmt"
  "os"
  "net/http"
  "crypto/md5"
  "encoding/hex"
)
type LogItem struct {
  ServerID string "json:server_id"
  Goroutine int "json:goroutine"
  Timestamp time.Time "json:time"
  Message string "json:message"
  Page string "json:page"
}
```

This is what will eventually become our JSON document that will be sent to our Couchbase server. We'll use the `Page`, `Timestamp`, and `ServerID` as a combined, hashed key to allow multiple, concurrent requests to the same document against separate servers to be logged separately, as shown in the following code:

```
var currentGoroutine int

func (li LogItem) logRequest(bucket *couchbase.Bucket) {

  hash := md5.New()
  io.WriteString(hash,li.ServerID+li.Page+li.Timestamp.Format("Jan
    1, 2014 12:00am"))
  hashString := hex.EncodeToString(hash.Sum(nil))
  bucket.Set(hashString,0,li)
  currentGoroutine = 0
}
```

When we reset `currentGoroutine` to `0`, we use an intentional race condition to allow goroutines to report themselves by numeric ID while executing concurrently. This allows us to debug an application that appears to work correctly until it invokes some form of concurrent architecture. Since goroutines will be self-identified by an ID, it allows us to add more granular routing of our messages.

By designating a different log location by goroutine `ID`, `timestamp`, and `serverID`, any concurrency issues that arise can be quickly plucked from logfiles. This is done using the following code:

```
func main() {
  hostName, _ := os.Hostname()
  currentGoroutine = 0

  logClient, err := couchbase.Connect("http://localhost:8091/")
    if err != nil {
      fmt.Println("Error connecting to logging client", err)
    }
  logPool, err := logClient.GetPool("default")
    if err != nil {
      fmt.Println("Error getting pool",err)
    }
  logBucket, err := logPool.GetBucket("logs")
    if err != nil {
      fmt.Println("Error getting bucket",err)
    }
  http.HandleFunc("/", func(w http.ResponseWriter, r
    *http.Request) {
    request := LogItem{}
    request.Goroutine = currentGoroutine
    request.ServerID = hostName
    request.Timestamp = time.Now()
    request.Message = "Request to " + r.URL.Path
    request.Page = r.URL.Path
    go request.logRequest(logBucket)

  })

  http.ListenAndServe(":8080",nil)

}
```

Using the log4go package for robust logging

As with most things in Go, where there's something satisfactory and extensible in the core page, it can be taken to the next level by a third party—Go's wonderful logging package is truly brought to life with **log4go**.

Using log4go greatly simplifies the process of file logging, console logging, and logging via TCP/UDP.

> For more information on log4go, visit `https://code.google.com/p/log4go/`.

Each instance of a `log4go Logger` interface can be configured by an XML configuration file and can have filters applied to it to dictate where messaging goes. Let's look at a simple HTTP server to show how we can direct specific logs to location, as shown in the following code:

```
package main

import (
  "code.google.com/p/log4go"
  "net/http"
  "fmt"
  "github.com/gorilla/mux"
)
```

```
var errorLog log4go.Logger
var errorLogWriter log4go.FileLogWriter

var accessLog log4go.Logger
var accessLogWriter *log4go.FileLogWriter

var screenLog log4go.Logger

var networkLog log4go.Logger
```

In the preceding code, we created four distinct log objects—one that writes errors to a logfile, one that writes accesses (page requests) to a separate file, one that sends directly to console (for important notices), and one that passes a log message across the network.

The last two obviously do not need `FileLogWriter`, although it's entirely possible to replicate the network logging using a shared drive if we can mitigate issues with concurrent access, as shown in the following code:

```
func init() {
  fmt.Println("Web Server Starting")
}

func pageHandler(w http.ResponseWriter, r *http.Request) {
  pageFoundMessage := "Page found: " + r.URL.Path
  accessLog.Info(pageFoundMessage)
  networkLog.Info(pageFoundMessage)
  w.Write([]byte("Valid page"))
}
```

Any request to a valid page goes here, sending the message to the `web-access.log` file `accessLog`.

```
func notFound(w http.ResponseWriter, r *http.Request) {
  pageNotFoundMessage := "Page not found / 404: " + r.URL.Path
  errorLog.Info(pageNotFoundMessage)
  w.Write([]byte("Page not found"))
}
```

As with the `accessLog` file, we'll take any `404 / page not found` request and route it directly to the `notFound()` method, which saves a fairly generic error message along with the invalid / missing URL requested. Let's look at what we'll do with extremely important errors and messages in the following code:

```
func restricted(w http.ResponseWriter, r *http.Request) {
  message := "Restricted directory access attempt!"
```

```
      errorLog.Info(message)
      accessLog.Info(message)
      screenLog.Info(message)
      networkLog.Info(message)
      w.Write([]byte("Restricted!"))

    }
```

The `restricted()` function and corresponding `screenLog` represents a message we deem as *critical* and worthy of going to not only the error and the access logs, but also to screen and passed across the wire as a `networkLog` item. In other words, it's a message so important that everybody gets it.

In this case, we're detecting attempts to get at our `.git` folder, which is a fairly common accidental security vulnerability that people have been known to commit in automatic file uploads and updates. Since we have cleartext passwords represented in files and may expose that to the outside world, we'll catch this on request and pass to our critical and noncritical logging mechanisms.

We might also look at this as a more open-ended bad request notifier—one worthy of immediate attention from a network developer. In the following code, we'll start creating a few loggers:

```
    func main() {

      screenLog = make(log4go.Logger)
      screenLog.AddFilter("stdout", log4go.DEBUG, log4go.
    NewConsoleLogWriter())

      errorLogWriter := log4go.NewFileLogWriter("web-errors.log",
        false)
        errorLogWriter.SetFormat("%d %t - %M (%S)")
        errorLogWriter.SetRotate(false)
        errorLogWriter.SetRotateSize(0)
        errorLogWriter.SetRotateLines(0)
        errorLogWriter.SetRotateDaily(true)
```

Since log4go opens up a bevy of additional logging options, we can play a bit with how our logs rotate and are formatted without having to draw that out specifically with `Sprintf` or something similar.

The options here are simple and expressive:

- `SetFormat`: This allows us to specify how our individual log lines will look.
- `SetRotate`: This permits automatic rotation based on the size of the file and/or the number of lines in `log`. The `SetRotateSize()` option sets rotation on bytes in the message and `SetRotateLines()` sets the maximum number of `lines`. The `SetRotateDaily()` function lets us create new logfiles based on the day regardless of our settings in the previous functions. This is a fairly common logging technique and can generally be burdensome to code by hand.

The output of our logging format ends up looking like the following line of code:

```
04/13/14 10:46 - Page found%!(EXTRA string=/valid)
  (main.pageHandler:24)
```

The `%S` part is the source, and that gives us the line number and our method trace for the part of our application that invoked the log:

```
errorLog = make(log4go.Logger)
errorLog.AddFilter("file", log4go.DEBUG, errorLogWriter)

networkLog = make(log4go.Logger)
networkLog.AddFilter("network", log4go.DEBUG,
  log4go.NewSocketLogWriter("tcp", "localhost:3000"))
```

Our network log sends JSON-encoded messages via TCP to the address we provide. We'll show a very simple handling server for this in the next section of code that translates the log messages into a centralized logfile:

```
accessLogWriter = log4go.NewFileLogWriter("web-access.log",false)
  accessLogWriter.SetFormat("%d %t - %M (%S)")
  accessLogWriter.SetRotate(true)
  accessLogWriter.SetRotateSize(0)
  accessLogWriter.SetRotateLines(500)
  accessLogWriter.SetRotateDaily(false)
```

Our `accessLogWriter` is similar to the `errorLogWriter` except that instead of rotating daily, we rotate it every 500 lines. The idea here is that access logs would of course be more frequently touched than an error log—hopefully. The code is as follows:

```
accessLog = make(log4go.Logger)
accessLog.AddFilter("file",log4go.DEBUG,accessLogWriter)

rtr := mux.NewRouter()
rtr.HandleFunc("/valid", pageHandler)
```

```
    rtr.HandleFunc("/.git/", restricted)
    rtr.NotFoundHandler = http.HandlerFunc(notFound)
```

In the preceding code, we used the Gorilla Mux package for routing. This gives us easier access to the `404` handler, which is less than simplistic to modify in the basic `http` package built directly into Go. The code is as follows:

```
    http.Handle("/", rtr)
    http.ListenAndServe(":8080", nil)
  }
```

Building the receiving end of a network logging system like this is also incredibly simple in Go, as we're building nothing more than another TCP client that can handle the JSON-encoded messages.

We can do this with a receiving server that looks remarkably similar to our TCP chat server from an earlier chapter. The code is as follows:

```
package main

import
(
  "net"
  "fmt"
)

type Connection struct {

}

func (c Connection) Listen(l net.Listener) {
  for {
    conn,_ := l.Accept()
    go c.logListen(conn)
  }
}
```

As with our chat server, we bind our listener to a `Connection` struct, as shown in the following code:

```
func (c *Connection) logListen(conn net.Conn) {
  for {
    buf := make([]byte, 1024)
    n, _ := conn.Read(buf)
    fmt.Println("Log Message",string(n))
  }
}
```

In the preceding code, we receive log messages delivered via JSON. At this point, we're not unmarshalling the JSON, but we've shown how to do that in an earlier chapter.

Any message sent will be pushed into the buffer—for this reason, it may make sense to expand the buffer's size depending on how detailed the information is.

```
func main() {
  serverClosed := make(chan bool)

  listener, err := net.Listen("tcp", ":3000")
  if err != nil {
    fmt.Println ("Could not start server!",err)
  }

  Conn := Connection{}

  go Conn.Listen(listener)

  <-serverClosed
}
```

You can imagine how network logging can be useful, particularly in server clusters where you might have a selection of, say, web servers and you don't want to reconcile individual logfiles into a single log.

Panicking

With all the discussion of capturing errors and logging them, we should probably consider the `panic()` and `recover()` functionality in Go.

As briefly discussed earlier, `panic()` and `recover()` operate as a more basic, immediate, and explicit error detection methodology than, say, `try`/`catch`/`finally` or even Go's built-in error return value convention. As designed, `panic()` unwinds the stack and leads to program exit unless `recover()` is invoked. This means that unless you explicitly recover, your application will end.

So, how is this useful other than for stopping execution? After all, we can catch an error and simply end the application manually through something similar to the following code:

```
package main

import
(
  "fmt"
```

```
    "os"
)

func processNumber(un int) {

  if un < 1 || un > 4 {
    fmt.Println("Now you've done it!")
    os.Exit(1)
  }else {
    fmt.Println("Good, you can read simple instructions.")
  }
}

func main() {
  userNum := 0
  fmt.Println("Enter a number between 1 and 4.")
  _,err := fmt.Scanf("%d",&userNum)
    if err != nil {}

  processNumber(userNum)
}
```

However, while this function does sanity checking and enacts a permanent, irreversible application exit, `panic()` and `recover()` allow us to reflect errors from a specific package and/or method, save those, and then resume gracefully.

This is very useful when we're dealing with methods that are called from other methods that are called from other methods, and so on. The types of deeply embedded or recursive functions that make it hard to discern a specific error are where `panic()` and `recover()` are most advantageous. You can also imagine how well this functionality can play with logging.

Recovering

The `panic()` function on its own is fairly simple, and it really becomes useful when paired with `recover()` and `defer()`.

Take, for example, an application that returns meta information about a file from the command line. The main part of the application will listen for user input, pass this into a function that will open the file, and then pass that file reference to another function that will get the file's details.

Now, we can obviously stack errors as return elements straight through the process, or we can panic along the way, recover back down the steps, and gather our errors at the bottom for logging and/or reporting directly to console.

Avoiding spaghetti code is a welcomed side effect of this approach versus the former one. Think of this in a general sense (this is pseudo code):

```
func getFileDetails(fileName string) error {
  return err
}

func openFile(fileName string) error {
  details,err := getFileDetails(fileName)
  return err
}

func main() {

  file,err := openFile(fileName)

}
```

With a single error, it's entirely manageable to approach our application in this way. However, when each individual function has one or more points of failure, we will require more and more return values and a way of reconciling them all into a single overall error message or messages. Check the following code:

```
package main


import
(
  "os"
  "fmt"
  "strconv"
)

func gatherPanics() {
  if rec := recover(); rec != nil {
    fmt.Println("Critical Error:", rec)
  }
}
```

This is our general recovery function, which is called before every method on which we wish to capture any panic. Let's look at a function to deduce the file's details:

```
func getFileDetails(fileName string) {
  defer gatherPanics()
  finfo,err := os.Stat(fileName)
  if err != nil {
    panic("Cannot access file")
  }else {
    fmt.Println("Size: ", strconv.FormatInt(finfo.Size(),10))
  }
}

func openFile(fileName string) {
  defer gatherPanics()
  if _, err := os.Stat(fileName); err != nil {
    panic("File does not exist")
  }

}
```

The two functions from the preceding code are merely an attempt to open a file and panic if the file does not exist. The second method, `getFileDetails()`, is called from the `main()` function such that it will always execute, regardless of a blocking error in `openFile()`.

In the real world, we will often develop applications where a nonfatal error stops just a portion of the application from working, but will not cause the application as a whole to break. Check the following code:

```
func main() {
  var fileName string
  fmt.Print("Enter filename>")
  _,err := fmt.Scanf("%s",&fileName)
  if err != nil {}
  fmt.Println("Getting info for",fileName)

  openFile(fileName)
  getFileDetails(fileName)

}
```

If we were to remove the `recover()` code from our `gatherPanics()` method, the application would crash if/when the file didn't exist.

This may seem ideal, but imagine a scenario where a user selects a nonexistent file for a directory that they lack the rights to view. When they solve the first problem, they will be presented with the second instead of seeing all potential issues at one time.

The value of expressive errors can't be overstated from a user experience standpoint. Gathering and presenting expressive errors is made easier through this methodology—even a `try`/`catch`/`finally` requires that we (as developers) explicitly do something with the returned error in the catch clause.

Logging our panics

In the preceding code, we can integrate a logging mechanism pretty simply in addition to catching our panics.

One consideration about logging that we haven't discussed is the notion of when to log. As our previous examples illustrate, we can sometimes run into problems that should be logged but may be mitigated by future user action. As such, we can choose to log our errors immediately or save it until the end of execution or a greater function.

The primary benefit of logging immediately is that we're not susceptible to an actual crash preventing our log from being saved. Take the following example:

```
type LogItem struct {
  Message string
  Function string
}

var Logs []LogItem
```

We've created a log `struct` and a slice of `LogItems` using the following code:

```
func SaveLogs() {
  logFile := log4go.NewFileLogWriter("errors.log",false)
    logFile.SetFormat("%d %t - %M (%S)")
    logFile.SetRotate(true)
    logFile.SetRotateSize(0)
    logFile.SetRotateLines(500)
    logFile.SetRotateDaily(false)

  errorLog := make(log4go.Logger)
  errorLog.AddFilter("file",log4go.DEBUG,logFile)
```

```
    for i:= range Logs {
      errorLog.Info(Logs[i].Message + " in " + Logs[i].Function)
    }

  }
```

This, ostensibly, is where all of our captured `LogItems` will be turned into a good collection of line items in a logfile. There is a problem, however, as illustrated in the following code:

```
func registerError(block chan bool) {

  Log := LogItem{ Message:"An Error Has Occurred!", Function:
    "registerError()"}
  Logs = append(Logs,Log)
  block <- true
}
```

Executed in a goroutine, this function is non-blocking and allows the main thread's execution to continue. The problem is with the following code that runs after the goroutine, which causes us to log nothing at all:

```
func separateFunction() {
  panic("Application quitting!")
}
```

Whether invoked manually or by the binary itself, the application quitting prematurely precludes our logfiles from being written, as that method is deferred until the end of the `main()` method. The code is as follows:

```
func main() {
  block := make(chan bool)
  defer SaveLogs()
  go func(block chan bool) {

    registerError(block)

  }(block)

  separateFunction()

}
```

The tradeoff here, however, is performance. If we execute a file operation every time we want to log something, we're potentially introducing a bottleneck into our application. In the preceding code, errors are sent via goroutine but written in blocking code—if we introduce the log writing directly into `registerError()`, it can slow down our final application.

As mentioned previously, one opportunity to mitigate these issues and allow the application to still save all of our log entries is to utilize either memory logging or network logging.

Catching stack traces with concurrent code

In earlier Go releases, the ability to properly execute a stack trace from our source was a daunting task, which is emblematic of some of the many complaints and concerns users had early on about general error handling in Go.

While the Go team has remained vigilant about the *right* way to do this (as they have with several other key language features such as a lack of generics), stack traces and stack info have been tweaked a bit as the language has grown.

Using the runtime package for granular stack traces

In an effort to capture stack traces directly, we can glean some helpful pieces of information from the built-in runtime package.

Specifically, Go provides a couple of tools to give us insight into the invocation and/or breakpoints of a goroutine. The following are the functions within the runtime package:

- `runtime.Caller()`: This returns information about the parent function of a goroutine
- `runtime.Stack()`: This allocates a buffer for the amount of data in a stack trace and then fills that with the trace
- `runtime.NumGoroutine()`: This returns the total number of open goroutines

We can utilize all three preceding tools to better describe the inner workings of any given goroutine and related errors.

Using the following code, we'll spawn some random goroutines doing random things and log not only the goroutine's log message, but also the stack trace and the goroutine's caller:

```
package main

import
(
  "os"
  "fmt"
  "runtime"
  "strconv"
  "code.google.com/p/log4go"
)


type LogItem struct {
  Message string
}

var LogItems []LogItem

func saveLogs() {
  logFile := log4go.NewFileLogWriter("stack.log", false)
    logFile.SetFormat("%d %t - %M (%S)")
    logFile.SetRotate(false)
    logFile.SetRotateSize(0)
    logFile.SetRotateLines(0)
    logFile.SetRotateDaily(true)

  logStack := make(log4go.Logger)
  logStack.AddFilter("file", log4go.DEBUG, logFile)
  for i := range LogItems {
    fmt.Println(LogItems[i].Message)
    logStack.Info(LogItems[i].Message)
  }
}
```

The `saveLogs()` function merely takes our map of `LogItems` and applies them to file per log4go, as we did earlier in the chapter. Next, we'll look at the function that supplies details on our goroutines:

```
func goDetails(done chan bool) {
  i := 0
  for {
    var message string
    stackBuf := make([]byte,1024)
    stack := runtime.Stack(stackBuf, false)
    stack++
    _, callerFile, callerLine, ok := runtime.Caller(0)
    message = "Goroutine from " + string(callerLine) + "" +
      string(callerFile) + " stack:" +     string(stackBuf)
    openGoroutines := runtime.NumGoroutine()

    if (ok == true) {
      message = message + callerFile
    }

    message = message +
      strconv.FormatInt(int64(openGoroutines),10) + " goroutines
        active"

    li := LogItem{ Message: message}

    LogItems = append(LogItems,li)
    if i == 20 {
      done <- true
      break
    }

    i++
  }
}
```

This is where we gather more details about a goroutine. The `runtime.Caller()` function provides a few returned values: its pointer, the filename of the caller, the line of the caller. The last return value indicates whether the caller could be found.

As mentioned previously, `runtime.NumGoroutine()` gives us the number of extant goroutines that have not yet been closed.

Then, in `runtime.Stack(stackBuf, false)`, we fill our buffer with the stack trace. Note that we're not trimming this byte array to length.

All three are passed into `LogItem.Message` for later use. Let's look at the setup in the `main()` function:

```
func main() {
  done := make(chan bool)

  go goDetails(done)
  for i:= 0; i < 10; i++ {
    go goDetails(done)
  }

  for {
    select {
      case d := <-done:
        if d == true {
          saveLogs()
          os.Exit(1)
        }
    }
  }

}
```

Finally, we loop through some goroutines that are doing loops themselves and exit upon completion.

When we examine our logfile, we're given far more verbose details on our goroutines than we have previously, as shown in the following code:

```
04/16/14 23:25 - Goroutine from + /var/log/go/ch9_11_stacktrace.
goch9_11_stacktrace.go stack:goroutine 4 [running]:
main.goDetails(0xc08400b300)
  /var/log/go/ch9_11_stacktrace.goch9_11_stacktrace.go:41 +0x8e
created by main.main
  /var/log/go/ch9_11_stacktrace.goch9_11_stacktrace.go:69 +0x4c

  /var/log/go/ch9_11_stacktrace.goch9_11_stacktrace.go14 goroutines
active (main.saveLogs:31)
```

For more information on the runtime package, go to `http://golang.org/pkg/runtime/`.

Summary

Debugging, testing, and logging concurrent code can be particularly cumbersome, often when concurrent goroutines fail in a seemingly silent fashion or fail to execute whatsoever.

We looked at various methods of logging, from file to console to memory to network logging, and examined how concurrent application pieces can fit into these various implementations.

By now, you should be comfortable and natural in creating robust and expressive logs that rotate automatically, impose no latency or bottlenecks, and assist in debugging your applications.

You should feel comfortable with the basics of the runtime package. We'll dive into the testing package, controlling goroutines more explicitly, and unit testing as we dig deeper in the next chapter.

In addition to further examining the testing and runtime packages, in our final chapter, we'll also broach the topic of more advanced concurrency topics in Go as well as review some overall best practices as they relate to programming in the Go language.

10 Advanced Concurrency and Best Practices

Once you're comfortable with the basic and intermediate usage of concurrency features in Go, you may find that you're able to handle the majority of your development use cases with bidirectional channels and standard concurrency tools.

In *Chapter 2*, *Understanding the Concurrency Model*, and *Chapter 3*, *Developing a Concurrent Strategy*, we looked at the concurrency models, not just of Go but of other languages as well, and compared the way they—and distributed models—can work. In this chapter, we'll touch on those and some higher level concepts with regard to designing and managing your concurrent application.

In particular, we're going to look at central management of goroutines and their associated channels—out of the box you may find goroutines to be a set-it-and-forget-it proposition; however, there are cases where we might want more granular control of a channel's state.

We've also looked quite a bit at testing and benchmarking from a high level, but we'll look at some more detailed and complex methods for testing. We'll also explore a primer on the Google App Engine, which will give us access to some specific testing tools we haven't yet used.

Finally, we'll touch upon some general best practices for Go, which will surely pertain not just to concurrent application design but your future work in general with the language.

Going beyond the basics with channels

We've talked about quite a few different channel implementations—channels of different type (interfaces, functions, structs, and channels)—and touched upon the differences in buffered and unbuffered channels. However, there's still a lot more we can do with the design and flow of our channels and goroutines.

By design, Go wants you to keep things simple. And that's fantastic for 90 percent of what you'll do with Go. But there are other times where you'll need to dig a little deeper for a solution, or when you'll need to save resources by preserving the amount of open goroutine processes, channels, and more.

You may, at some point, want some hands on control of the size and state, and also the control of a running or closed goroutine, so we'll look at doing that.

Just as importantly, designing your goroutines to work in concert with the application design as a whole can be critical to unit testing, which is a topic we'll touch on in this final chapter.

Building workers

Earlier in this book, we talked about concurrency patterns and a bit about workers. We even brought the workers concept into play in the previous chapter, when we were building our logging systems.

Truly speaking, "worker" is a fairly generic and ambiguous concept, not just in Go, but in general programming and development. In some languages, it's an object/instantiated class, and in others it's a concurrent actor. In functional programming languages, worker is a graduated function return passed to another.

If we go back to the preface, we will see that we have literally used the Go gopher as an example of a worker. In short, a worker is something more complex than a single function call or programmatic action that will perform a task one or more times.

So why are we talking about it now? When we build our channels, we are creating a mechanism to do work. When we have a struct or an interface, we're combining methods and values at a single place, and then doing work using that *object* as both a mechanism for the work as well as a place to store information about that work.

This is particularly useful in application design, as we're able to delegate various elements of an application's functionality to distinct and well-defined workers. Consider, for example, a server pinging application that has specific pieces doing specific things in a self-contained, compartmentalized manner.

We'll attempt to check for server availability via the HTTP package, check the status code and errors, and back off if we find problems with any particular server. You can probably see where this is going—this is the most basic approach to load balancing. But an important design consideration is the way in which we manage our channels.

We'll have a master channel, where all important global transactions should be accumulated and evaluated, but each individual server will also have its own channels for handling tasks that are important only to that individual struct.

The design in the following code can be considered as a rudimentary pipeline, which is roughly akin to the producer/consumer model we talked about in the previous chapters:

```
package main

import
(
  "fmt"
  "time"
  "net/http"
)

const INIT_DELAY = 3000
const MAX_DELAY = 60000
const MAX_RETRIES = 4
const DELAY_INCREMENT = 5000
```

The preceding code gives the configuration part of the application, setting scope on how frequently to check servers, the maximum amount of time for backing off, and the maximum amount of retries before giving up entirely.

The `DELAY_INCREMENT` value represents how much time we will add to our server checking process each time we discover a problem. Let's take a look at how to create a server in the following section:

```
var Servers []Server

type Server struct {
  Name string
  URI string
  LastChecked time.Time
  Status bool
  StatusCode int
  Delay int
  Retries int
  Channel chan bool
}
```

Now, we design the basic server (using the following code), which contains its current status, the last time it was checked, the present delay between checks, its own channel for evaluating statuses and establishing the new status, and updated retry delay:

```
func (s *Server) checkServerStatus(sc chan *Server) {
  var previousStatus string

    if s.Status == true {
      previousStatus = "OK"
    }else {
      previousStatus = "down"
    }

    fmt.Println("Checking Server",s.Name)
    fmt.Println("\tServer was",previousStatus,"on last check
      at",s.LastChecked)

    response, err := http.Get(s.URI)
    if err != nil {
      fmt.Println("\tError: ",err)
      s.Status = false
      s.StatusCode = 0
    }else {
      fmt.Println(response.Status)
      s.StatusCode = response.StatusCode
      s.Status = true
    }

    s.LastChecked = time.Now()
    sc <- s
}
```

The `checkServerStatus()` method is the meat and potatoes of our application here. We pass all of our servers through this method in the `main()` function to our `cycleServers()` loop, after which it becomes self-fulfilling.

If our `Status` is set to `true`, we send the state to the console as `OK` (otherwise `down`) and set our `Server` status code with `s.StatusCode` as either the HTTP code or `0` if there was a network or other error.

Finally, set the last-checked time of `Server` to `Now()` and pass `Server` through the `serverChan` channel. In the following code, we'll demonstrate how we'll rotate through our available servers:

```
func cycleServers(sc chan *Server) {

  for i := 0; i < len(Servers); i++ {
    Servers[i].Channel = make(chan bool)
    go Servers[i].updateDelay(sc)
    go Servers[i].checkServerStatus(sc)
  }

}
```

This is our initial loop, called from main. It simply loops through our available servers and initializes its listening goroutine as well as sending the first `checkServerStatus` request.

It's worth noting two things here: first, the channel invoked by `Server` will never actually die, but instead the application will stop checking the server. That's fine for all practical purposes here, but if we have thousands and thousands of servers to check, we're wasting resources on what essentially amounts to an unclosed channel and a map element that has not been removed. Later, we'll broach the concept of manually killing goroutines, something we've only been able to do through abstraction by stopping the communication channel. Let's now take a look at the following code that controls a server's status and its next steps:

```
func (s *Server) updateDelay(sc chan *Server) {
  for {
    select {
      case msg := <- s.Channel:

        if msg == false {
          s.Delay = s.Delay + DELAY_INCREMENT
          s.Retries++
          if s.Delay > MAX_DELAY {
            s.Delay = MAX_DELAY
          }
        }else {
          s.Delay = INIT_DELAY
        }
        newDuration := time.Duration(s.Delay)

        if s.Retries <= MAX_RETRIES {
          fmt.Println("\tWill check server again")
          time.Sleep(newDuration * time.Millisecond)
          s.checkServerStatus(sc)

        }else {
          fmt.Println("\tServer not reachable
            after",MAX_RETRIES,"retries")
        }

      default:
    }
  }
}
```

This is where each `Server` will listen for changes in its status, as reported by `checkServerStatus()`. When any given `Server` struct receives a message that a change in status has been reported via our initial loop, it will evaluate that message and act accordingly.

If the `Status` is set to `false`, we know that the server was inaccessible for some reason. The `Server` reference itself will then add a delay to the next time it's checked. If it's set to `true`, the server was accessible and the delay will either be set or reset to the default retry value of `INIT_DELAY`.

It finally sets a sleep mode on that goroutine before reinitializing the `checkServerStatus()` method on itself, passing the `serverChan` reference along in the initial goroutine loop in the `main()` function:

```
func main() {

  endChan := make(chan bool)
  serverChan := make(chan *Server)

Servers = []Server{ {Name: "Google", URI: "http://www.google.com",
Status: true, Delay: INIT_DELAY}, {Name: "Yahoo", URI: "http://www.
yahoo.com", Status: true, Delay: INIT_DELAY}, {Name: "Bad Amazon",
URI: "http://amazon.zom", Status: true, Delay: INIT_DELAY} }
```

One quick note here—in our slice of `Servers`, we intentionally introduced a typo in the last element. You'll notice `amazon.zom`, which will provoke an HTTP error in the `checkServerStatus()` method. The following is the function to cycle through servers to find an appropriate match:

```
      go cycleServers(serverChan)

      for {
        select {
          case currentServer := <- serverChan:
            currentServer.Channel <- false
          default:

        }
      }

      <- endChan

    }
```

The following is an example of the output with the typo included:

```
Checking Server Google
        Server was OK on last check at 0001-01-01 00:00:00 +0000 UTC
        200 OK
        Will check server again
Checking Server Yahoo
        Server was OK on last check at 0001-01-01 00:00:00 +0000 UTC
        200 OK
        Will check server again
Checking Server Amazon
        Server was OK on last check at 0001-01-01 00:00:00 +0000 UTC
        Error:  Get http://amazon.zom: dial tcp: GetAddrInfoW: No such
host is known.
        Will check server again
Checking Server Google
        Server was OK on last check at 2014-04-23 12:49:45.6575639 -0400
EDT
```

We'll be taking the preceding code for one last spin through some concurrency patterns later in this chapter, turning it into something a bit more practical.

Implementing nil channel blocks

One of the bigger problems in designing something like a pipeline or producer/consumer model is there's somewhat of a black hole when it comes to the state of any given goroutine at any given time.

Consider the following loop, wherein a producer channel creates an arbitrary set of consumer channels and expects each to do one and only one thing:

```
package main

import (
  "fmt"
  "time"
)

const CONSUMERS = 5

func main() {

  Producer := make(chan (chan int))

  for i := 0; i < CONSUMERS; i++ {
    go func() {
      time.Sleep(1000 * time.Microsecond)
      conChan := make(chan int)

      go func() {
        for {
          select {
          case _,ok := <-conChan:
            if ok  {
              Producer <- conChan
            }else {
              return
            }
          default:
          }
        }
      }()

      conChan <- 1
      close(conChan)
    }()
  }
```

Given a random amount of consumers to produce, we attach a channel to each and pass a message upstream to the `Producer` via that consumer's channel. We send just a single message (which we could handle with a buffered channel), but we simply close the channel after.

Whether in a multithreaded application, a distributed application, or a highly concurrent application, an essential attribute of a producer-consumer model is the ability for data to move across a queue/channel in a steady, reliable fashion. This requires some modicum of mutual knowledge to be shared between both the producer and consumers.

Unlike environments that are distributed (or multicore), we do possess some inherent awareness of the status on both ends of that arrangement. We'll next look at a listening loop for producer messages:

```
    for {
      select {
      case consumer, ok := <-Producer:
        if ok == false {
          fmt.Println("Goroutine closed?")
          close(Producer)
        } else {
          log.Println(consumer)
          // consumer <- 1
        }
        fmt.Println("Got message from secondary channel")
      default:
      }
    }
  }
```

The primary issue is that one of the `Producer` channel doesn't know much about any given `Consumer`, including when it's actively running. If we uncommented the `// consumer <- 1` line, we'll get a panic, because we're attempting to send a message on a closed channel.

As a message is passed across a secondary goroutine's channel, upstream to the channel of the `Producer`, we get an appropriate reception, but cannot detect when the downstream goroutine is closed.

Knowing when a goroutine has terminated is in many cases inconsequential, but consider an application that spawns new goroutines when a certain number of tasks are complete, effectively breaking a task into mini tasks. Perhaps each chunk is dependent on the total completion of the last chunk, and a broadcaster must know the status of the current goroutines before moving on.

Using nil channels

In the earlier versions of Go, you could communicate across uninitialized, thus nil or 0-value channels without a panic (although your results would be unpredictable). Starting from Go Version 1, communication across nil channels produced a consistent but sometimes confusing effect.

It's vital to note that within a select switch, transmission on a nil channel on its own will still cause a deadlock and panic. This is something that will most often creep up when utilizing global channels and not ever properly initializing them. The following is an example of such transmission on a nil channel:

```
func main() {

  var channel chan int

    channel <- 1

  for {
    select {
      case <- channel:

      default:
    }
  }

}
```

As the channel is set to its `0` value (nil, in this case), it blocks perpetually and the Go compiler will detect this, at least in more recent versions. You can also duplicate this outside of a `select` statement, as shown in the following code:

```
  var done chan int
  defer close(done)
  defer log.Println("End of script")
  go func() {
    time.Sleep(time.Second * 5)
    done <- 1
  }()

  for {
    select {
      case <- done:
        log.Println("Got transmission")
        return
      default:
    }
  }
```

The preceding code will block forever without the panic, due to the default in the `select` statement keeping the main loop active while waiting for communication on the channel. If we initialize the channel, however, the application runs as expected.

With these two fringe cases—closed channels and nil channels—we need a way for a master channel to understand the state of a goroutine.

Implementing more granular control over goroutines with tomb

As with many such problems—both niche and common—there exists a third-party utility for grabbing your goroutines by the horns.

Tomb is a library that provides diagnostics to go along with any goroutine and channel—it can tell a master channel if another goroutine is dead or dying.

In addition, it allows you to explicitly kill a goroutine, which is a bit more nuanced than simply closing the channel it is attached to. As previously mentioned, closing the channel is effectively neutering a goroutine, although it could ultimately still be active.

You are about to find a simple fetch-and-grab body script that takes a slice of URL structs (with status and URI) and attempts to grab the HTTP response for each and apply it to the struct. But instead of just reporting information from the goroutines, we'll have the ability to send "kill messages" to each of a "master" struct's child goroutines.

In this example, we'll run the script for 10 seconds, and if any of the goroutines fail to do their job in that allotted time, it will respond that it was unable to get the URL's body due to a kill send from the master struct that invoked it:

```
package main

import (
  "fmt"
  "io/ioutil"
  "launchpad.net/tomb"
  "net/http"
  "strconv"
  "sync"
  "time"
)

var URLS []URL

type GoTomb struct {
  tomb tomb.Tomb
}
```

This is the minimum necessary structure required to create a parent or a master struct for all of your spawned goroutines. The `tomb.Tomb` struct is simply a mutex, two channels (one for dead and dying), and a reason error struct. The structure of the `URL` struct looks like the following code:

```
type URL struct {
  Status bool
  URI    string
  Body   string
}
```

Our `URL` struct is fairly basic—`Status`, set to `false` by default and `true` when the body has been retrieved. It consists of the `URI` variable—which is the reference to the URL—and the `Body` variable for storing the retrieved data. The following function allows us to execute a "kill" on a `GoTomb` struct:

```
func (gt GoTomb) Kill() {

  gt.tomb.Kill(nil)

}
```

The preceding method invokes `tomb.Kill` on our `GoTomb` struct. Here, we have set the sole parameter to `nil`, but this can easily be changed to a more descriptive error, such as `errors.New("Time to die, goroutine")`. Here, we'll show the listener for the `GoTomb` struct:

```
func (gt *GoTomb) TombListen(i int) {

  for {
    select {
    case <-gt.tomb.Dying():
      fmt.Println("Got kill command from tomb!")
      if URLS[i].Status == false {
        fmt.Println("Never got data for", URLS[i].URI)
      }
      return
    }
  }
}
```

We invoke `TombListen` attached to our `GoTomb`, which sets a select that listens for the `Dying()` channel, as shown in the following code:

```
func (gt *GoTomb) Fetch() {
  for i := range URLS {
    go gt.TombListen(i)

    go func(ii int) {

      timeDelay := 5 * ii
      fmt.Println("Waiting ", strconv.FormatInt(int64(timeDelay),
        10), " seconds to get", URLS[ii].URI)
      time.Sleep(time.Duration(timeDelay) * time.Second)
      response, _ := http.Get(URLS[ii].URI)
      URLS[ii].Status = true
      fmt.Println("Got body for ", URLS[ii].URI)
      responseBody, _ := ioutil.ReadAll(response.Body)
      URLS[ii].Body = string(responseBody)
    }(i)
  }
}
```

When we invoke `Fetch()`, we also set the tomb to `TombListen()`, which receives those "master" messages across all spawned goroutines. We impose an intentionally long wait to ensure that our last few attempts to `Fetch()` will come after the `Kill()` command. Finally, our `main()` function, which handles the overall setup:

```
func main() {

  done := make(chan int)

  URLS = []URL{{Status: false, URI: "http://www.google.com", Body:
""}, {Status: false, URI: "http://www.amazon.com", Body: ""}, {Status:
false, URI: "http://www.ubuntu.com", Body: ""}}

  var MasterChannel GoTomb
  MasterChannel.Fetch()

  go func() {

    time.Sleep(10 * time.Second)
    MasterChannel.Kill()
    done <- 1
  }()
```

```
    for {
      select {
      case <-done:
        fmt.Println("")
        return
      default:
      }
    }
  }
```

By setting `time.Sleep` to `10` seconds and then killing our goroutines, we guarantee that the 5 second delays between `Fetch()` prevent the last of our goroutines from successfully finishing before being killed.

For the tomb package, go to `http://godoc.org/launchpad.net/tomb` and install it using the `go get launchpad.net/tomb` command.

Timing out with channels

One somewhat critical point with channels and `select` loops that we haven't examined particularly closely is the ability—and often necessity—to kill a `select` loop after a certain timeout.

Many of the applications we've written so far are long-running or perpetually-running, but there are times when we'll want to put a finite time limit on how long goroutines can operate.

The `for { select { } }` switch we've used so far will either live perpetually (with a default case) or wait to be broken from one or more of the cases.

There are two ways to manage interval-based tasks—both as part of the time package, unsurprisingly.

The `time.Ticker` struct allows for any given operation after the specified period of time. It provides C, a blocking channel that can be used to detect activity sent after that period of time; refer to the following code:

```
package main

import (
  "log"
  "time"
)
```

```
func main() {

  timeout := time.NewTimer(5 * time.Second)
  defer log.Println("Timed out!")

  for {
    select {
    case <-timeout.C:
      return
    default:
    }
  }

}
```

We can extend this to end channels and concurrent execution after a certain amount of time. Take a look at the following modifications:

```
package main

import (
  "fmt"
  "time"
)

func main() {

  myChan := make(chan int)

  go func() {
    time.Sleep(6 * time.Second)
    myChan <- 1
  }()

  for {
    select {
      case <-time.After(5 * time.Second):
        fmt.Println("This took too long!")
        return
      case <-myChan:
        fmt.Println("Too little, too late")
    }
  }
}
```

Building a load balancer with concurrent patterns

When we built our server pinging application earlier in this chapter, it was probably pretty easy to imagine taking this to a more usable and valuable space.

Pinging a server is often the first step in a health check for a load balancer. Just as Go provides a usable out-of-the-box web server solution, it also presents a very clean `Proxy` and `ReverseProxy` struct and methods, which makes creating a load balancer rather simple.

Of course, a round-robin load balancer will need a lot of background work, specifically on checking and rechecking as it changes the `ReverseProxy` location between requests. We'll handle these with the goroutines triggered with each request.

Finally, note that we have some dummy URLs at the bottom in the configuration—changing those to production URLs should immediately turn the server that runs this into a working load balancer. Let's look at the main setup for the application:

```
package main

import (
  "fmt"
  "log"
  "net/http"
  "net/http/httputil"
  "net/url"
  "strconv"
  "time"
)

const MAX_SERVER_FAILURES = 10
const DEFAULT_TIMEOUT_SECONDS = 5
const MAX_TIMEOUT_SECONDS = 60
const TIMEOUT_INCREMENT = 5
const MAX_RETRIES = 5
```

In the previous code, we defined our constants, much like we did previously. We have a MAX_RETRIES, which limits how many failures we can have, MAX_TIMEOUT_SECONDS, which defines the longest amount of time we'll wait before trying again, and our TIMEOUT_INCREMENT for changing that value between failures. Next, let's look at the basic construction of our Server struct:

```
type Server struct {
  Name        string
  Failures    int
  InService   bool
  Status      bool
  StatusCode  int
  Addr        string
  Timeout     int
  LastChecked time.Time
  Recheck     chan bool
}
```

As we can see in the previous code, we have a generic Server struct that maintains the present state, the last status code, and information on the last time the server was checked.

Note that we also have a Recheck channel that triggers the delayed attempt to check the Server again for availability. Each Boolean passed across this channel will either remove the server from the available pool or reannounce that it is still in service:

```
func (s *Server) serverListen(serverChan chan bool) {
  for {
    select {
    case msg := <-s.Recheck:
      var statusText string
      if msg == false {
        statusText = "NOT in service"
        s.Failures++
        s.Timeout = s.Timeout + TIMEOUT_INCREMENT
        if s.Timeout > MAX_TIMEOUT_SECONDS {
          s.Timeout = MAX_TIMEOUT_SECONDS
        }
      } else {
        if ServersAvailable == false {
          ServersAvailable = true
          serverChan <- true
        }
        statusText = "in service"
        s.Timeout = DEFAULT_TIMEOUT_SECONDS
      }

      if s.Failures >= MAX_SERVER_FAILURES {
        s.InService = false
        fmt.Println("\tServer", s.Name, "failed too many times.")
      } else {
        timeString := strconv.FormatInt(int64(s.Timeout), 10)
        fmt.Println("\tServer", s.Name, statusText, "will check
          again in", timeString, "seconds")
        s.InService = true
        time.Sleep(time.Second * time.Duration(s.Timeout))
        go s.checkStatus()
      }

    }
  }
}
```

This is the instantiated method that listens on each server for messages delivered on the availability of a server at any given time. While running a goroutine, we keep a perpetually listening channel open to listen to Boolean responses from `checkStatus()`. If the server is available, the next delay is set to default; otherwise, `TIMEOUT_INCREMENT` is added to the delay. If the server has failed too many times, it's taken out of rotation by setting its `InService` property to `false` and no longer invoking the `checkStatus()` method. Let's next look at the method for checking the present status of `Server`:

```
func (s *Server) checkStatus() {
  previousStatus := "Unknown"
  if s.Status == true {
    previousStatus = "OK"
  } else {
    previousStatus = "down"
  }
  fmt.Println("Checking Server", s.Name)
  fmt.Println("\tServer was", previousStatus, "on last check at",
    s.LastChecked)
  response, err := http.Get(s.Addr)
  if err != nil {
    fmt.Println("\tError: ", err)
    s.Status = false
    s.StatusCode = 0
  } else {
    s.StatusCode = response.StatusCode
    s.Status = true
  }

  s.LastChecked = time.Now()
  s.Recheck <- s.Status
}
```

Our `checkStatus()` method should look pretty familiar based on the server ping example. We look for the server; if it is available, we pass `true` to our `Recheck` channel; otherwise `false`, as shown in the following code:

```
  func healthCheck(sc chan bool) {
    fmt.Println("Running initial health check")
    for i := range Servers {
      Servers[i].Recheck = make(chan bool)
      go Servers[i].serverListen(sc)
      go Servers[i].checkStatus()
    }
  }
```

Our `healthCheck` function simply kicks off the loop of each server checking (and re-checking) its status. It's run only one time, and initializes the `Recheck` channel via the `make` statement:

```
  func roundRobin() Server {
    var AvailableServer Server

    if nextServerIndex > (len(Servers) - 1) {
      nextServerIndex = 0
    }

    if Servers[nextServerIndex].InService == true {
      AvailableServer = Servers[nextServerIndex]
    } else {
      serverReady := false
      for serverReady == false {
        for i := range Servers {
          if Servers[i].InService == true {
            AvailableServer = Servers[i]
            serverReady = true
          }
        }

      }
    }
    nextServerIndex++
    return AvailableServer
  }
```

The `roundRobin` function first checks the next available `Server` in the queue—if that server happens to be down, it loops through the remaining to find the first available `Server`. If it loops through all, it will reset to `0`. Let's look at the global configuration variables:

```
var Servers []Server
var nextServerIndex int
var ServersAvailable bool
var ServerChan chan bool
var Proxy *httputil.ReverseProxy
var ResetProxy chan bool
```

These are our global variables—our `Servers` slice of `Server` structs, the `nextServerIndex` variable, which serves to increment the next `Server` to be returned, `ServersAvailable` and `ServerChan`, which start the load balancer only after a viable server is available, and then our `Proxy` variables, which tell our `http` handler where to go. This requires a `ReverseProxy` method, which we'll look at now in the following code:

```
func handler(p *httputil.ReverseProxy) func(http.ResponseWriter,
*http.Request) {
  Proxy = setProxy()
  return func(w http.ResponseWriter, r *http.Request) {

    r.URL.Path = "/"

    p.ServeHTTP(w, r)

  }
}
```

Note that we're operating on a `ReverseProxy` struct here, which is different from our previous forays into serving webpages. Our next function executes the round robin and gets our next available server:

```
func setProxy() *httputil.ReverseProxy {

  nextServer := roundRobin()
  nextURL, _ := url.Parse(nextServer.Addr)
  log.Println("Next proxy source:", nextServer.Addr)
  prox := httputil.NewSingleHostReverseProxy(nextURL)

  return prox
}
```

The `setProxy` function is called after every request, and you can see it as the first line in our handler. Next we have the general listening function that looks out for requests we'll be reverse proxying:

```
func startListening() {
  http.HandleFunc("/index.html", handler(Proxy))
  _ = http.ListenAndServe(":8080", nil)

}

func main() {
  nextServerIndex = 0
  ServersAvailable = false
  ServerChan := make(chan bool)
  done := make(chan bool)

  fmt.Println("Starting load balancer")
  Servers = []Server{{Name: "Web Server 01", Addr: "http://www.google.
com", Status: false, InService: false}, {Name: "Web Server 02", Addr:
"http://www.amazon.com", Status: false, InService: false}, {Name: "Web
Server 03", Addr: "http://www.apple.zom", Status: false, InService:
false}}

  go healthCheck(ServerChan)

  for {
    select {
    case <-ServerChan:
      Proxy = setProxy()
```

```
        startListening()
        return

      }
    }

    <-done
  }
```

With this application, we have a simple but extensible load balancer that works with the common, core components in Go. Its concurrency features keep it lean and fast, and we wrote it in a very small amount of code using exclusively standard Go.

Choosing unidirectional and bidirectional channels

For the purpose of simplicity, we've designed most of our applications and sample code with bidirectional channels, but of course any channel can be set unidirectionally. This essentially turns a channel into a "read-only" or "write-only" channel.

If you're wondering why you should bother limiting the direction of a channel when it doesn't save any resources or guarantee an issue, the reason boils down to simplicity of code and limiting the potential for panics.

By now we know that sending data on a closed channel results in a panic, so if we have a write-only channel, we'll never accidentally run into that problem in the wild. Much of this can also be mitigated with `WaitGroups`, but in this case that's a sledgehammer being used on a nail. Consider the following loop:

```
const TOTAL_RANDOMS = 100

func concurrentNumbers(ch chan int) {
  for i := 0; i < TOTAL_RANDOMS; i++ {
    ch <- i
  }
}

func main() {

  ch := make(chan int)
```

```
    go concurrentNumbers(ch)

    for {
      select {
        case num := <- ch:
          fmt.Println(num)
          if num == 98 {
            close(ch)
          }
        default:
      }
    }
  }
```

Since we're abruptly closing our `ch` channel one digit before the goroutine can finish, any writes to it cause a runtime error.

In this case, we are invoking a read-only command, but it's in the `select` loop. We can safeguard this a bit more by allowing only specific actions to be sent on unidirectional channels. This application will always work up to the point where in the channel is closed prematurely, one shy of the `TOTAL_RANDOMS` constant.

Using receive-only or send-only channels

When we limit the direction or the read/write capability of our channels, we also reduce the potential for closed channel deadlocks if one or more of our processes inadvertently sends on such a channel.

So the short answer to the question "When is it appropriate to use a unidirectional channel?" is "Whenever you can."

Don't force the issue, but if you can set a channel to read/write only, it may preempt issues down the road.

Using an indeterminate channel type

One trick that can often come in handy, and we haven't yet addressed, is the ability to have what is effectively a typeless channel.

If you're wondering why that might be useful, the short answer is concise code and application design thrift. Often this is a discouraged tactic, but you may find it useful from time to time, especially when you need to communicate one or more disparate concepts across a single channel. The following is an example of an indeterminate channel type:

```
package main

import (

  "fmt"
  "time"
)

func main() {

  acceptingChannel := make(chan interface{})

  go func() {

    acceptingChannel <- "A text message"
    time.Sleep(3 * time.Second)
    acceptingChannel <- false
  }()

  for {
    select {
      case msg := <- acceptingChannel:
        switch typ := msg.(type) {
          case string:
            fmt.Println("Got text message",typ)
          case bool:
            fmt.Println("Got boolean message",typ)
            if typ == false {
              return
            }
          default:
          fmt.Println("Some other type of message")
        }

      default:

    }

  }

  <- acceptingChannel
}
```

Using Go with unit testing

As with many of the basic and intermediate development and deployment requirements you may have, Go comes with a built-in application for handling unit tests.

The basic premise behind testing is that you create your package and then create a testing package to run against the initial application. The following is a very basic example:

```
mathematics.go
package mathematics

func Square(x int) int {

  return x * 3
}
mathematics_test.go
package mathematics

import
(
  "testing"
)

func Test_Square_1(t *testing.T) {
  if Square(2) != 4 {
    t.Error("Square function failed one test")
  }
}
```

A simple Go test in that subdirectory will give you the response you're looking for. While this was admittedly simple—and purposefully flawed—you can probably see how easy it is to break apart your code and test it incrementally. This is enough to do very basic unit tests out of the box.

Correcting this would then be fairly simple—the same test would pass on the following code:

```
func Square(x int) int {

  return x * x
}
```

The testing package is somewhat limited; however, as it provides basic pass/fails without the ability to do assertions. There are two third-party packages that can step in and help in this regard, and we'll explore them in the following sections.

GoCheck

GoCheck extends the basic testing package primarily by augmenting it with assertions and verifications. You'll also get some basic benchmarking utility out of it that works a little more fundamentally than anything you'd need to engineer using Go.

For more details on GoCheck visit `http://labix.org/gocheck` and install it using `go get gopkg.in/check.v1`.

Ginkgo and Gomega

Unlike GoCheck, Ginkgo (and its dependency Gomega) takes a different approach to testing, utilizing the **behavior-driven development** (**BDD**) model. Behavior-driven development is a general model for making sure your application does what it should at every step, and Ginkgo formalizes that into some easily parseable properties.

BDD tends to complement test-driven development (for example, unit testing) rather than replacement. It seeks to answer a few critical questions about the way people (or other systems) will interact with your application. In that sense, we'll generally describe a process and what we expect from that process in fairly human-friendly terms. The following is a short snippet of such an example:

```
Describe("receive new remote TCP connection", func() {
    Context("user enters a number", func() {
        It("should be an integer", func() {
        })
    })
})
```

This allows testing to be as granular as unit testing, but also expands the way we handle application usage in verbose and explicit behaviors.

If BDD is something you or your organization is interested in, this is a fantastic, mature package for implementing deeper unit testing.

For more information on Ginkgo go to `https://github.com/onsi/ginkgo` and install it using `go get github.com/onsi/ginkgo/ginkgo`.

For more information on dependency, refer to `go get github.com/onsi/gomega`.

Using Google App Engine

If you're unfamiliar with Google App Engine, the short version is it's a cloud environment that allows for simple building and deployment of **Platform-As-A-Service** (**paas**) solutions.

Compared to a lot of similar solutions, Google App Engine allows you to build and test your applications in a very simple and straightforward way. Google App Engine allows you to write and deploy in Python, Java, PHP, and of course, Go.

For the most part, Google App Engine provides a standard Go installation that makes it easy to dovetail off of the `http` package. But it also gives you a few noteworthy additional packages that are unique to Google App Engine itself:

Package	Description
`appengine/memcache`	This provides a distributed memcache installation unique to Google App Engine
`appengine/mail`	This allows you to send e-mails through an SMTP-esque platform
`appengine/log`	Given your storage may be more ephemeral here, it formalizes a cloud version of the log
`appengine/user`	This opens both identity and OAuth capabilities
`appengine/search`	This gives your application the power of Google search on your own data via datastore
`appengine/xmpp`	This provides Google Chat-like capabilities
`appengine/urlfetch`	This is a crawler functionality
`appengine/aetest`	This extends unit testing for Google App Engine

While Go is still considered beta for Google App Engine, you can expect that if anyone was able to competently deploy it in a cloud environment, it would be Google.

Utilizing best practices

The wonderful thing with Go when it comes to best practices is that even if you don't necessarily do everything right, either Go will yell at you or provide you with the tools necessary to fix it.

If you attempt to include code and not use it, or if you attempt to initialize a variable and not use it, Go will stop you. If you want to clean up your code's formatting, Go enables it with `go fmt`.

Structuring your code

One of the easiest things you can do when building a package from scratch is to structure your code directories in an idiomatic way. The standard for a new package would look something like the following code:

```
/projects/
  thisproject/
    bin/
    pkg/
    src/
      package/
        mypackage.go
```

Setting up your Go code like this is not just helpful for your own organization, but allows you to distribute your package more easily.

Documenting your code

For anyone who has worked in a corporate or collaborative coding environment, documentation is sacrosanct. As you may recall earlier, using the `godoc` command allows you to quickly get information about a package at the command line or via an ad hoc localhost server. The following are the two basic ways you may use `godoc`:

Using godoc	Description
`godoc fmt`	This brings `fmt` documentation to the screen
`godoc -http=:3000`	This hosts the documentation on port `:3030`

Go makes it super easy to document your code, and you absolutely should. By simply adding single-line comments above each identifier (package, type, or function), you'll append that to the contextual documentation, as shown in the following code:

```
// A demo documentation package
package documentation

// The documentation struct object
// Chapter int represents a document's chapter
// Content represents the text of the documentation
type Documentation struct {
  Chapter int
  Content string
}
```

```
//  Display() outputs the content of any given Document by chapter
func (d Documentation) Display() {

}
```

When installed, this will allow anyone to run the `godoc` documentation on your package and get as much detailed information as you're willing to supply.

You'll often see more robust examples of this in the Go core code itself, and it's worth reviewing that to compare your style of documentation to Google's and the Go community's.

Making your code available via go get

Assuming you've kept your code in a manner consistent with the organizational techniques as listed previously, making your code available via code repositories and hosts should be a cinch.

Using GitHub as the standard, here's how we might design our third-party application:

1. Make sure you stick to the previous structural format.
2. Keep your source files under the directory structures they'll live in remotely. In other words, expect that your local structure will reflect the remote structure.
3. Perhaps obviously, commit only the files you wish to share in the remote repository.

Assuming your repository is public, anyone should be able to get (`go get`) and then install (`go install`) your package.

Keeping concurrency out of your packages

One last point that might seem somewhat out of place given the context of the book—if you're building separate packages that will be imported, avoid including concurrent code whenever possible.

This is not a hard-and-fast rule, but when you consider potential usage, it makes sense—let the main application handle the concurrency unless your package absolutely needs it. Doing so will prevent a lot of hidden and difficult-to-debug behavior that may make your library less appealing.

Summary

It is my sincere hope that you've been able to explore, understand, and utilize the depths of Go's powerful abilities with concurrency through this book.

We've gone over a lot, from the most basic, channel-free concurrent goroutines to complex channel types, parallelism, and distributed computing, and we've brought some example code along at every step.

By now, you should be fully equipped to build anything your heart desires in code, in a manner that is highly concurrent, fast, and error-free. Beyond that, you should be able to produce well-formed, properly-structured, and documented code that can be used by you, your organization, or others to implement concurrency where it is best utilized.

Concurrency itself is a vague concept; it's one that means slightly different things to different people (and across multiple languages), but the core goal is always fast, efficient, and reliable code that can provide performance boosts to any application.

Armed with a full understanding of both the implementation of concurrency in Go as well as its inner workings, I hope you continue your Go journey as the language evolves and grows, and similarly implore you to consider contributing to the Go project itself as it develops.

Bibliography

This book is a blend of text and projects, all packaged up keeping your journey in mind. It includes content from the following Packt books:

- Learning Go Web Development, Nathan Kozyra
- Go Programming Blueprints, Mat Ryer
- Mastering Concurrency in Go, Nathan Kozyra