Community Experience Distilled

Mastering Go Web Services

Program and deploy fast, scalable web services and create high-performance RESTful APIs using Go

Nathan Kozyra

[PACKT] PUBLISHING open source* community experience distilled

Mastering Go Web Services

Program and deploy fast, scalable web services and create high-performance RESTful APIs using Go

Nathan Kozyra

[PACKT] PUBLISHING open source community experience distilled

BIRMINGHAM - MUMBAI

Mastering Go Web Services

First published: April 2015

Production reference: 1080415

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham B3 2PB, UK.

ISBN 978-1-78398-130-4

`www.packtpub.com`

Credits

Author
Nathan Kozyra

Reviewers
Jiahua Chen
János Fehér
Aleksandar S. Sokolovski
Forrest Y. Yu

Commissioning Editor
Julian Ursell

Acquisition Editor
Kevin Colaco

Content Development Editor
Amey Varangaonkar

Technical Editors
Edwin Moses
Shali Sasidharan

Copy Editors
Jasmine Nadar
Vikrant Phadke

Project Coordinator
Suzanne Coutinho

Proofreaders
Simran Bhogal
Stephen Copestake
Maria Gould
Paul Hindle

Indexer
Monica Ajmera Mehta

Graphics
Sheetal Aute

Production Coordinator
Alwin Roy

Cover Work
Alwin Roy

About the Author

Nathan Kozyra is a veteran developer and software architect with more than a dozen years of experience in developing web applications and large-scale SaaS platforms. He has also authored the book *Mastering Concurrency in Go*, published by Packt Publishing.

I would like to thank my wife, Mary, and my son, Ethan, for their perpetual love and support.

About the Reviewers

Jiahua Chen is a Gopher, web application developer, and signing lecturer. He is the creator of the Gogs project. He is also pursuing his studies. He was the technical reviewer for the video *Building Your First Web Application with Go*, published by Packt Publishing.

János Fehér has been involved in a wide variety of projects since 1996, including technical support for NATO operations and the development of a high-performance computing grid, national TV and radio websites, and web applications for universities and adult learning.

In recent years, he has been heavily involved in distributed and concurrent software architectures. He is currently the head of development for the start-up called Intern Avenue, where his team is working on a matching technology platform to help employers find the best young talent for their business.

I would like to thank my amazing fiancée, Szilvi, for her support and patience during the many long days and nights it has taken me to make this book relevant for both stakeholders and technologists.

Aleksandar S. Sokolovski is a software engineering professional from Europe. He has a a bachelor's degree in computer science from the Ss. Cyril and Methodius University and a master's degree in technology, innovation, and entrepreneurship from the University of Sheffield. He was a member of the organizational committee, participant, and presenter on multiple international research conferences; he is also a published author of many research papers. He has worked as a research and teaching associate at the Faculty of Informatics and Computer Science in Skopje, Macedonia. He is currently working as a research associate and software engineer in the telecom industry. He is a member of IEEE, PMI, and AAAS.

Aleksandar has worked as a reviewer for the book *Mastering Concurrency in GO*, published by Packt Publishing.

I give my regards and blessings to everyone who supported me to complete this project: my family for their support and their help, especially my sister, Sofija, my parents, Stefan and Brankica, and last but not least, my girlfriend.

Forrest Y. Yu is an author of two books on operating systems. He has a wide range of interests and experience in desktop applications, web services, LBS, operating systems, cloud computing, information security, and so on. Recently, he has been working at Amazon, building the next generation information security platform and tools. He is also the reviewer of the book *Scratch Cookbook* and the video *Building Games with Scratch 2.0*, both of which were published by Packt Publishing. He has a blog, `http://forrestyu.com/`, where you can find more information about him.

I want to thank my wife, Danhong, for all her support. I also want to thank my son, Frank, for all the happiness he brings to us.

Table of Contents

Preface

If there's one thing that's said more than anything else about the Go language, it's that "*Go is a server language*."

Certainly, Go was built to be an ideal server language, designed as a next-generation iteration over the expanses and/or over-engineering of C, C++, and Java.

The language has evolved—largely with fervent community support—to go far beyond servers into system tools, graphics, and even compilers for new languages. At its heart, however, Go is made for powerful, concurrent, and easy-to-deploy cross-platform servers. This is what makes the language ideal for this book's topic.

Mastering Web Services in Go is intended to be a guide to building robust web services and APIs that can scale for production, with emphasis on security, scalability, and adherence to RESTful principles.

In this book, we'll build a rudimentary API for a social network, which will allow us to demonstrate and dive deeper into some fundamental concepts, such as connecting Go to other services and keeping your server secure and highly available.

By the end of this book, you should be experienced with all the relevant instances to build a robust, scalable, secure, and production-ready web service.

What this book covers

Chapter 1, Our First API in Go, quickly introduces—or reintroduces—some core concepts related to Go setup and usage as well as the `http` package.

Chapter 2, RESTful Services in Go, focuses on the guiding principles of the REST architecture and translates them into our overall API design infrastructure.

Chapter 3, *Routing and Bootstrapping*, is devoted to applying the RESTful practices from the previous chapter to built-in, third-party, and custom routers for the scaffolding of our API.

Chapter 4, *Designing APIs in Go*, explores overall API design while examining other related concepts, such as utilization of web sockets and HTTP status codes within the REST architecture.

Chapter 5, *Templates and Options in Go*, covers ways to utilize the `OPTIONS` request endpoints, implementing TLS and authentication, and standardizing response formats in our API.

Chapter 6, *Accessing and Using Web Services in Go*, explores ways to integrate other web services for authentication and identity in a secure way.

Chapter 7, *Working with Other Web Technologies*, focuses on bringing in other critical components of application architecture, such as frontend reverse proxy servers and solutions, to keep session data in the memory or datastores for quick access.

Chapter 8, *Responsive Go for the Web*, looks at expressing the values of our API as a consumer might, but utilizing frontend, client-side libraries to parse and present our responses.

Chapter 9, *Deployment*, introduces deployment strategies, including utilization of processes to keep our server running, highly accessible, and interconnected with associated services.

Chapter 10, *Maximizing Performance*, stresses upon various strategies for keeping our API alive, responsive, and fast in production. We look at caching mechanisms that are kept on disk as well as in memory, and explore ways in which we can distribute these mechanisms across multiple machines or images.

Chapter 11, *Security*, focuses more on best practices to ensure that your application and sensitive data are protected. We look at eliminating SQL injection and cross-site scripting attacks.

What you need for this book

To use the examples in this book, you can utilize any one of a Windows, Linux, or OS X machine, though you may find Windows limiting with some of the third-party tools we'll be using.

You'll obviously need to get the Go language platform installed. The easiest way to do this is through a binary, available for OS X or Windows at [URL]. Go is also readily available via multiple Linux package managers, such as yum or aptitude.

The choice of the IDE is largely a personal issue, but we recommend Sublime Text, which has fantastic Go support, among other languages. We'll spend a bit more time detailing some of the pros and cons of the other common IDEs in *Chapter 1*, *Our First API in Go*.

We'll utilize quite a few additional platforms and services, such as MySQL, MongoDB, Nginx, and more. Most should be available across platforms, but if you're running Windows, it's recommended that you consider running a Linux platform—preferably an Ubuntu server—on a virtual machine to ensure maximum compatibility.

Who this book is for

This book is intended for developers who are experienced in both Go and server-side development for web services and APIs. We haven't spent any time on the basics of programming in Go, so if you're shaky on that aspect, it's recommended that you brush up on it prior to diving in.

The target reader is comfortable with web performance at the server level, has some familiarity with REST as a guiding principle for API design, and is at least aware of Go's native server capabilities.

We don't anticipate that you'll be an expert in all the technologies covered, but fundamental understanding of Go's core library is essential, and general understanding of networked server architecture setup and maintenance is ideal.

Conventions

In this book, you will find a number of styles of text that distinguish between different kinds of information. Here are some examples of these styles, and an explanation of their meaning.

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "Now download the `julia-n.m.p-win64.exe` file on a temporary folder."

A block of code is set as follows:

```
package main

import (
  "fmt"
)
func main() {
  fmt.Println("Here be the code")
}
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold as follows:

```
package main
import (
  "fmt"
)

func stringReturn(text string) string {
  return text
}

func main() {
  myText := stringReturn("Here be the code")
  fmt.Println(myText)
}
```

Any command-line input or output is written as follows:

```
curl --head http://localhost:8080/api/user/read/1111
HTTP/1.1 200 OK
Date: Wed, 18 Jun 2014 14:09:30 GMT
Content-Length: 12
Content-Type: text/plain; charset=utf-8
```

New terms and **important words** are shown in bold. Words that you see on the screen, in menus or dialog boxes for example, appear in the text like this: "When a user clicks on **Accept**, we'll be returned to our redirect URL with the code that we're looking for."

Warnings or important notes appear in a box like this.

Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or may have disliked. Reader feedback is important for us to develop titles that you really get the most out of.

To send us general feedback, simply send an e-mail to `feedback@packtpub.com`, and mention the book title via 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 on `www.packtpub.com/authors`.

Customer support

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

Downloading the color images of this book

We also provide you a PDF file that has color images of the screenshots/diagrams used in this book. The color images will help you better understand the changes in the output. You can download this file from: `http://www.packtpub.com/sites/default/files/downloads/1304OS_ColorImages.pdf`.

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 books—maybe a mistake in the text or the code—we would be grateful if you would report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting `http://www.packtpub.com/submit-errata`, selecting your book, 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 on our website, or added to any list of existing errata, under the Errata section of that title. Any existing errata can be viewed by selecting your title from `http://www.packtpub.com/support`.

Piracy

Questions

You can contact us at `questions@packtpub.com` if you are having a problem with any aspect of the book, and we will do our best to address it.

1
Our First API in Go

If you spend any time developing applications on the Web (or off it, for that matter), it won't be long before you find yourself facing the prospect of interacting with a web service or an API.

Whether it's a library that you need or another application's sandbox with which you have to interact, the world of development relies in no small part on the cooperation among dissonant applications, languages, and formats.

That, after all, is why we have APIs to begin with—to allow standardized communication between any two given platforms.

If you spend a long amount of time working on the Web, you'll encounter bad APIs. By *bad* we mean APIs that are not all-inclusive, do not adhere to best practices and standards, are confusing semantically, or lack consistency. You'll encounter APIs that haphazardly use OAuth or simple HTTP authentication in some places and the opposite in others, or more commonly, APIs that ignore the stated purposes of HTTP verbs (we will discuss more on this later in the chapter).

Google's Go language is particularly well suited to servers. With its built-in HTTP serving, a simple method for XML and JSON encoding of data, high availability, and concurrency, it is the ideal platform for your API.

Throughout this book, we'll not only explore a robust and clean API development but also its interaction with other APIs and data sources, and best practices for such development. We'll build one large service and a bunch of smaller ones for individual, self-contained lessons.

Most importantly, by the end, you should be able to interact with any networked API in Go and be able to design and execute a well-polished API suite yourself.

This book requires at least a casual familiarity with the web-based APIs and a beginner's level competency in Go, but we'll do some very brief introductions when we discuss new concepts and steer you to more information if it turns out that you're not entirely versed in this aspect of either Go or APIs.

We will also touch a bit on concurrency in Go, but we won't get too detailed—if you wish to learn more about this, please check out for the book authored by me, *Mastering Concurrency in Go, Packt Publishing*.

We will cover the following topics in this chapter:

- Understanding requirements and dependencies
- Introducing the HTTP package
- Building our first routes
- Setting data via HTTP
- Serving data from the datastore to the client

Understanding requirements and dependencies

Before we get too deep into the weeds in this book, it would be a good idea for us to examine the things that you will need to have installed in order to handle all our examples as we develop, test, and deploy our APIs.

Installing Go

It should go without saying that we will need to have the Go language installed. However, there are a few associated items that you will also need to install in order to do everything we do in this book.

Go is available for Mac OS X, Windows, and most common Linux variants. You can download the binaries at `http://golang.org/doc/install`.

On Linux, you can generally grab Go through your distribution's package manager. For example, you can grab it on Ubuntu with a simple `apt-get install golang` command. Something similar exists for most distributions.

In addition to the core language, we'll also work a bit with the Google App Engine, and the best way to test with the App Engine is to install the **Software Development Kit** (**SDK**). This will allow us to test our applications locally prior to deploying them and simulate a lot of the functionality that is provided only on the App Engine.

The App Engine SDK can be downloaded from `https://developers.google.com/appengine/downloads`.

While we're obviously most interested in the Go SDK, you should also grab the Python SDK as there are some minor dependencies that may not be available solely in the Go SDK.

Installing and using MySQL

We'll be using quite a few different databases and datastores to manage our test and real data, and MySQL will be one of the primary ones.

We will use MySQL as a storage system for our users; their messages and their relationships will be stored in our larger application (we will discuss more about this in a bit).

MySQL can be downloaded from `http://dev.mysql.com/downloads/`.

You can also grab it easily from a package manager on Linux/OS X as follows:

- Ubuntu: `sudo apt-get install mysql-server mysql-client`
- OS X with Homebrew: `brew install  mysql`

Redis

Redis is the first of the two NoSQL datastores that we'll be using for a couple of different demonstrations, including caching data from our databases as well as the API output.

If you're unfamiliar with NoSQL, we'll do some pretty simple introductions to results gathering using both Redis and Couchbase in our examples. If you know MySQL, Redis will at least feel similar, and you won't need the full knowledge base to be able to use the application in the fashion in which we'll use it for our purposes.

Redis can be downloaded from `http://redis.io/download`.
Redis can be downloaded on Linux/OS X using the following:

- Ubuntu: `sudo apt-get install redis-server`
- OS X with Homebrew: `brew install redis`

Couchbase

As mentioned earlier, Couchbase will be our second NoSQL solution that we'll use in various products, primarily to set short-lived or ephemeral key store lookups to avoid bottlenecks and as an experiment with in-memory caching.

Unlike Redis, Couchbase uses simple REST commands to set and receive data, and everything exists in the JSON format.

Couchbase can be downloaded from `http://www.couchbase.com/download`.

- For Ubuntu (`deb`), use the following command to download Couchbase:

  ```
  dpkg -i couchbase-server version.deb
  ```

- For OS X with Homebrew use the following command to download Couchbase:

  ```
  brew install
    https://github.com/couchbase/homebrew/raw/
      stable/Library/Formula/libcouchbase.rb
  ```

Nginx

Although Go comes with everything you need to run a highly concurrent, performant web server, we're going to experiment with wrapping a reverse proxy around our results. We'll do this primarily as a response to the real-world issues regarding availability and speed. *Nginx is not available natively for Windows*.

- For Ubuntu, use the following command to download Nginx:

  ```
  apt-get install nginx
  ```

- For OS X with Homebrew, use the following command to download Nginx:

  ```
  brew install nginx
  ```

Apache JMeter

We'll utilize JMeter for benchmarking and tuning our API for performance. You have a bit of a choice here, as there are several stress-testing applications for simulating traffic. The two we'll touch on are **JMeter** and Apache's built-in **Apache Benchmark** (**AB**) platform. The latter is a stalwart in benchmarking but is a bit limited in what you can throw at your API, so JMeter is preferred.

One of the things that we'll need to consider when building an API is its ability to stand up to heavy traffic (and introduce some mitigating actions when it cannot), so we'll need to know what our limits are.

[Apache JMeter can be downloaded from `http://jmeter.apache.org/download_jmeter.cgi`.]

Using predefined datasets

While it's not entirely necessary to have our dummy dataset throughout the course of this book, you can save a lot of time as we build our social network by bringing it in because it is full of users, posts, and images.

By using this dataset, you can skip creating this data to test certain aspects of the API and API creation.

[Our dummy dataset can be downloaded at `https://github.com/nkozyra/masteringwebservices`.]

Choosing an IDE

A choice of **Integrated Development Environment** (**IDE**) is one of the most personal choices a developer can make, and it's rare to find a developer who is not steadfastly passionate about their favorite.

Nothing in this book will require one IDE over another; indeed, most of Go's strength in terms of compiling, formatting, and testing lies at the command-line level. That said, we'd like to at least explore some of the more popular choices for editors and IDEs that exist for Go.

Eclipse

As one of the most popular and expansive IDEs available for any language, Eclipse is an obvious first mention. Most languages get their support in the form of an Eclipse plugin and Go is no exception.

There are some downsides to this monolithic piece of software; it is occasionally buggy on some languages, notoriously slow for some autocompletion functions, and is a bit heavier than most of the other available options.

However, the pluses are myriad. Eclipse is very mature and has a gigantic community from which you can seek support when issues arise. Also, it's free to use.

- Eclipse can be downloaded from `http://eclipse.org/`
- Get the Goclipse plugin at `http://goclipse.github.io/`

Sublime Text

Sublime Text is our particular favorite, but it comes with a large caveat—it is the only one listed here that is not free.

This one feels more like a complete code/text editor than a heavy IDE, but it includes code completion options and the ability to integrate the Go compilers (or other languages' compilers) directly into the interface.

Although Sublime Text's license costs $70, many developers find its elegance and speed to be well worth it. You can try out the software indefinitely to see if it's right for you; it operates as nagware unless and until you purchase a license.

Sublime Text can be downloaded from `http://www.sublimetext.com/2`.

LiteIDE

LiteIDE is a much younger IDE than the others mentioned here, but it is noteworthy because it has a focus on the Go language.

It's cross-platform and does a lot of Go's command-line magic in the background, making it truly integrated. LiteIDE also handles code autocompletion, `go fmt`, build, run, and test directly in the IDE and a robust package browser.

It's free and totally worth a shot if you want something lean and targeted directly for the Go language.

LiteIDE can be downloaded from `https://code.google.com/p/golangide/`.

IntelliJ IDEA

Right up there with Eclipse is the JetBrains family of IDE, which has spanned approximately the same number of languages as Eclipse. Ultimately, both are primarily built with Java in mind, which means that sometimes other language support can feel secondary.

The Go integration here, however, seems fairly robust and complete, so it's worth a shot if you have a license. If you do not have a license, you can try the Community Edition, which is free.

- You can download IntelliJ IDEA at `http://www.jetbrains.com/idea/download/`
- The Go language support plugin is available at `http://plugins.jetbrains.com/plugin/?idea&id=5047`

Some client-side tools

Although the vast majority of what we'll be covering will focus on Go and API services, we will be doing some visualization of client-side interactions with our API.

In doing so, we'll primarily focus on straight HTML and JavaScript, but for our more interactive points, we'll also rope in jQuery and AngularJS.

Most of what we do for client-side demonstrations will be available at this book's GitHub repository at `https://github.com/nkozyra/goweb` under client.

Both jQuery and AngularJS can be loaded dynamically from Google's CDN, which will prevent you from having to download and store them locally. The examples hosted on GitHub call these dynamically.

To load AngularJS dynamically, use the following code:

```
<script src="//ajax.googleapis.com/ajax/libs/
angularjs/1.2.18/angular.min.js"></script>
```

To load jQuery dynamically, use the following code:

```
<script src="//ajax.googleapis.com/ajax/libs/
jquery/1.11.1/jquery.min.js"></script>
```

Looking at our application

Throughout this book, we'll be building myriad small applications to demonstrate points, functions, libraries, and other techniques. However, we'll also focus on a larger project that mimics a social network wherein we create and return to users, statuses, and so on, via the API.

Though we'll be working towards the larger application as a way to demonstrate each piece of the puzzle, we'll also build and test self-contained applications, APIs, and interfaces.

The latter group will be prefaced with a quick hitter to let you know that it's not part of our larger application.

Setting up our database

As mentioned earlier, we'll be designing a social network that operates almost entirely at the API level (at least at first) as our *master* project in this book.

When we think of the major social networks (from the past and in the present), there are a few omnipresent concepts endemic among them, which are as follows:

- The ability to create a user and maintain a user profile
- The ability to share messages or statuses and have conversations based on them
- The ability to express pleasure or displeasure on the said statuses/messages to dictate the worthiness of any given message

There are a few other features that we'll be building here, but let's start with the basics. Let's create our database in MySQL as follows:

```
create database social_network;
```

This will be the basis of our social network product in this book. For now, we'll just need a `users` table to store our individual users and their most basic information. We'll amend this to include more features as we go along:

```
CREATE TABLE users (
  user_id INT(10) UNSIGNED NOT NULL AUTO_INCREMENT,
  user_nickname VARCHAR(32) NOT NULL,
  user_first VARCHAR(32) NOT NULL,
  user_last VARCHAR(32) NOT NULL,
  user_email VARCHAR(128) NOT NULL,
  PRIMARY KEY (user_id),
  UNIQUE INDEX user_nickname (user_nickname)
)
```

We won't need to do too much in this chapter, so this should suffice. We'll have a user's most basic information—name, nickname, and e-mail, and not much else.

Introducing the HTTP package

The vast majority of our API work will be handled through REST, so you should become pretty familiar with Go's `http` package.

In addition to serving via HTTP, the `http` package comprises of a number of other very useful utilities that we'll look at in detail. These include cookie jars, setting up clients, reverse proxies, and more.

The primary entity about which we're interested right now, though, is the `http.Server` struct, which provides the very basis of all of our server's actions and parameters. Within the server, we can set our TCP address, HTTP multiplexing for routing specific requests, timeouts, and header information.

Go also provides some shortcuts for invoking a server without directly initializing the struct. For example, if you have a lot of default properties, you could use the following code:

```
Server := Server {
  Addr: ":8080",
  Handler: urlHandler,
  ReadTimeout: 1000 * time.MicroSecond,
  WriteTimeout: 1000 * time.MicroSecond,
  MaxHeaderBytes: 0,
  TLSConfig: nil
}
```

You can simply execute using the following code:

```
http.ListenAndServe(":8080", nil)
```

This will invoke a server struct for you and set only the `Addr` and `Handler` properties within.

There will be times, of course, when we'll want more granular control over our server, but for the time being, this will do just fine. Let's take this concept and output some JSON data via HTTP for the first time.

Quick hitter – saying Hello, World via API

As mentioned earlier in this chapter, we'll go off course and do some work that we'll preface with **quick hitter** to denote that it's unrelated to our larger project.

In this case, we just want to rev up our `http` package and deliver some JSON to the browser. Unsurprisingly, we'll be merely outputting the uninspiring `Hello, world` message to, well, the world.

Let's set this up with our required package and imports:

```
package main

import
(
  "net/http"
  "encoding/json"
  "fmt"
)
```

This is the bare minimum that we need to output a simple string in JSON via HTTP. Marshalling JSON data can be a bit more complex than what we'll look at here, so if the struct for our message doesn't immediately make sense, don't worry.

This is our response struct, which contains all of the data that we wish to send to the client after grabbing it from our API:

```
type API struct {
  Message string "json:message"
}
```

There is not a lot here yet, obviously. All we're setting is a single message string in the obviously-named `Message` variable.

Finally, we need to set up our main function (as follows) to respond to a route and deliver a marshaled JSON response:

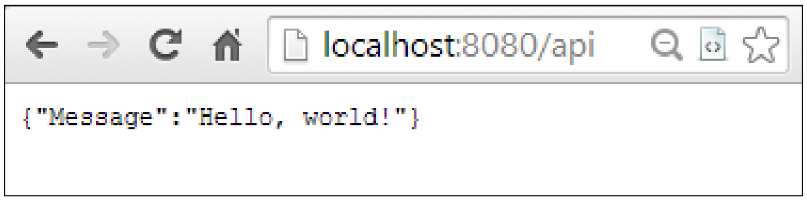
```
func main() {

  http.HandleFunc("/api", func(w http.ResponseWriter, r
    *http.Request) {

    message := API{"Hello, world!"}

    output, err := json.Marshal(message)

    if err != nil {
      fmt.Println("Something went wrong!")
    }

    fmt.Fprintf(w, string(output))

  })

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

Upon entering `main()`, we set a route handling function to respond to requests at `/api` that initializes an API struct with `Hello, world!` We then marshal this to a JSON byte array, `output`, and after sending this message to our `iowriter` class (in this case, an `http.ResponseWriter` value), we cast that to a string.

The last step is a kind of quick-and-dirty approach for sending our byte array through a function that expects a string, but there's not much that could go wrong in doing so.

Go handles typecasting pretty simply by applying the type as a function that flanks the target variable. In other words, we can cast an `int64` value to an integer by simply surrounding it with the `int(OurInt64)` function. There are some exceptions to this—types that cannot be directly cast and some other pitfalls, but that's the general idea. Among the possible exceptions, some types cannot be directly cast to others and some require a package like `strconv` to manage typecasting.

If we head over to our browser and call `localhost:8080/api` (as shown in the following screenshot), you should get exactly what we expect, assuming everything went correctly:



```
{"Message":"Hello, world!"}
```

Building our first route

When we talk about routing in Go nomenclature, we're more accurately discussing a multiplexer or `mux`. In this case, the multiplexer refers to taking URLs or URL patterns and translating them into internal functions.

You can think of this as a simple mapping from a request to a function (or a handler). You might draw up something like this:

```
/api/user  func apiUser
/api/message  func apiMessage
/api/status  func apiStatus
```

There are some limitations with the built-in mux/router provided by the `net/http` package. You cannot, for example, supply a wildcard or a regular expression to a route.

You might expect to be able to do something as discussed in the following code snippet:

```
http.HandleFunc("/api/user/\d+", func(w http.ResponseWriter, r
  *http.Request) {

  // react dynamically to an ID as supplied in the URL

})
```

However, this results in a parsing error.

If you've spent any serious time in any mature web API, you'll know that this won't do. We need to be able to react to dynamic and unpredictable requests. By this we mean that anticipating every numerical user is untenable as it relates to mapping to a function. We need to be able to accept and use patterns.

There are a few solutions for this problem. The first is to use a third-party platform that has this kind of robust routing built in. There are a few very good platforms to choose from, so we'll quickly look at these now.

Gorilla

Gorilla is an all-inclusive web framework, and one that we'll use quite a bit in this book. It has precisely the kind of URL routing package that we need (in its `gorilla/mux` package), and it also supplies some other very useful tools, such as JSON-RPC, secure cookies, and global session data.

Gorilla's `mux` package lets us use regular expressions, but it also has some shorthand expressions that let us define the kind of request string we expect without having to write out full expressions.

For example, if we have a request like `/api/users/309`, we can simple route it as follows in Gorilla:

```
gorillaRoute := mux.NewRouter()
gorillaRoute.HandleFunc("/api/{user}", UserHandler)
```

However, there is a clear risk in doing so—by leaving this so open-ended, we have the potential to get some data validation issues. If this function accepts anything as a parameter and we expect digits or text only, this will cause problems in our underlying application.

So, Gorilla allows us to clarify this with regular expressions, which are as follows:

```
r := mux.NewRouter()
r.HandleFunc("/products/{user:\d+}", ProductHandler)
```

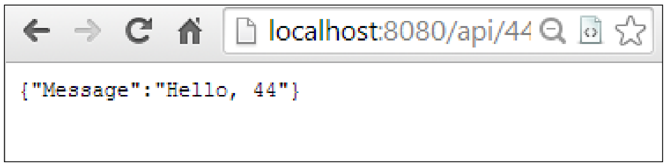
And now, we will only get what we expect—digit-based request parameters. Let's modify our previous example with this concept to demonstrate this:

```
package main

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

type API struct {
  Message string "json:message"
}

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

  urlParams := mux.Vars(r)
  name := urlParams["user"]
  HelloMessage := "Hello, " + name

  message := API{HelloMessage}
  output, err := json.Marshal(message)

  if err != nil {
    fmt.Println("Something went wrong!")
  }

  fmt.Fprintf(w, string(output))

}

func main() {

  gorillaRoute := mux.NewRouter()
  gorillaRoute.HandleFunc("/api/{user:[0-9]+}", Hello)
  http.Handle("/", gorillaRoute)
  http.ListenAndServe(":8080", nil)
}
```

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.

With this code, we have some validation at the routing level. A valid request to `/api/44` will give us a proper response, as shown in the following screenshot:



```
{"Message":"Hello, 44"}
```

An invalid request to something like `/api/nkozyra` will give us a 404 response.

- You can download the Gorilla web toolkit from `http://www.gorillatoolkit.org/`
- The documentation on its URL multiplexer can be found at `http://www.gorillatoolkit.org/pkg/mux`

Routes

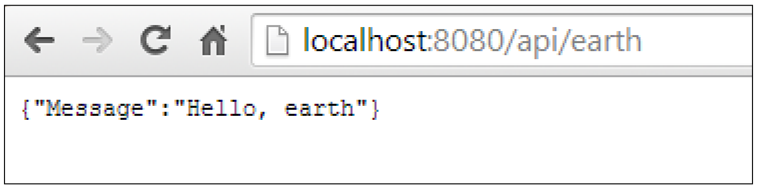
Routes, from `drone.io`, is explicitly and solely a routing package for Go. This makes it much more focused than the Gorilla web toolkit.

For the most part, URL routing will not be a bottleneck in a smaller application, but it's something that should be considered as your application scales. For our purpose, the differences in speed between, say, Gorilla and Routes is negligible.

Defining your `mux` package in routes is very clean and simple. Here is a variation on our `Hello world` message that responds to URL parameters:

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

  urlParams := r.URL.Query()
  name := urlParams.Get(":name")
  HelloMessage := "Hello, " + name
  message := API{HelloMessage}
  output, err := json.Marshal(message)

  if err != nil {
    fmt.Println("Something went wrong!")
  }

  fmt.Fprintf(w, string(output))

}

func main() {

  mux := routes.New()
  mux.Get("/api/:name", Hello)
  http.Handle("/", mux)
  http.ListenAndServe(":8080", nil)
}
```

The primary difference here (as with Gorilla) is that we're passing our `routes` multiplexer to `http` instead of using the internal one. And as with Gorilla, we can now use variable URL patterns to change our output, as follows:



```
{"Message":"Hello, earth"}
```

> You can read more about routes and how to install them at: `https://github.com/drone/routes`.
>
> Run the following command to install routes:
>
> **`go get github.com/drone/routes`**

Setting data via HTTP

Now that we've examined how we're going to handle routing, let's take a stab at injecting data into our database directly from a REST endpoint.

In this case, we'll be looking exclusively at the `POST` request methods because in most cases when large amounts of data could be transferred, you want to avoid the length limitations that the `GET` requests impose.

> Technically, a `PUT` request is the semantically correct method to use for requests that are made to create data in the **create-read-update-delete** (**CRUD**) concept, but years of disuse have largely relegated `PUT` to a historical footnote. Recently, some support for restoring `PUT` (and `DELETE`) to their proper place has taken hold. Go (and Gorilla) will gladly allow you to relegate requests to either and as we go forward, we'll move towards more protocol-valid semantics.

Connecting to MySQL

Go has a largely built-in agnostic database connection facility, and most third-party database connectivity packages yield to it. Go's default SQL package is `database/sql`, and it allows more general database connectivity with some standardization.

However, rather than rolling our own MySQL connection (for now, at least), we'll yield to a third-party add-on library. There are a couple of these libraries that are available, but we'll go with `Go-MySQL-Driver`.

You can install `Go-MySQL-Driver` using the following command (it requires Git):

`go get github.com/go-sql-driver/mysql`

For the purpose of this example, we'll assume that you have MySQL running with a localhost on the `3306` standard port. If it is not running, then please make the necessary adjustments accordingly in the examples. The examples here will also use a passwordless root account for the sake of clarity.

Our imports will remain largely the same but with two obvious additions: the `sql` package (`database/sql`) and the aforementioned MySQL driver that is imported solely for side effects by prepending it with an underscore:

```
package main

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

We'll set a new endpoint using Gorilla. You may recall that when we intend to set or create data, we'll generally push for a `PUT` or `POST` verb, but for the purposes of this demonstration, appending URL parameters is the easiest way to push data. Here is how we'd set up this new route:

```
  routes := mux.NewRouter()
  routes.HandleFunc("/api/user/create", CreateUser).Methods("GET")
```

Note that we're specifying the verbs that we'll accept for this request. In real usage, this is recommended for the `GET` requests.

Our `CreateUser` function will accept several parameters—`user`, `email`, `first`, and `last`. `User` represents a short user name and the rest should be self-explanatory. We'll precede our code with the definition of a `User` struct as follows:

```
type User struct {
  ID int "json:id"
  Name  string "json:username"
  Email string "json:email"
```

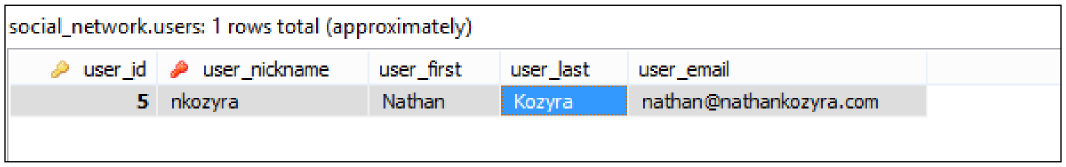
```
    First string "json:first"
    Last  string "json:last"
  }
```

And now let's take a look at the `CreateUser` function itself:

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

    NewUser := User{}
    NewUser.Name = r.FormValue("user")
    NewUser.Email = r.FormValue("email")
    NewUser.First = r.FormValue("first")
    NewUser.Last = r.FormValue("last")
    output, err := json.Marshal(NewUser)
    fmt.Println(string(output))
    if err != nil {
      fmt.Println("Something went wrong!")
    }

    sql := "INSERT INTO users set user_nickname='" + NewUser.Name +
      "', user_first='" + NewUser.First + "', user_last='" +
        NewUser.Last + "', user_email='" + NewUser.Email + "'"
    q, err := database.Exec(sql)
    if err != nil {
      fmt.Println(err)
    }
    fmt.Println(q)
  }
```

When we run this, our routed API endpoint should be available at `localhost:8080/api/user/create`. Though if you look at the call itself, you'll note that we need to pass URL parameters to create a user. We're not yet doing any sanity checking on our input, nor are we making certain it's clean/escaped, but we'll hit the URL as follows: `http://localhost:8080/api/user/create?user=nkozyra&first=Nathan&last=Kozyra&email=nathan@nathankozyra.com`.

And then, we'll end up with a user created in our `users` table, as follows:


social_network.users: 1 rows total (approximately)

user_id	user_nickname	user_first	user_last	user_email
5	nkozyra	Nathan	Kozyra	nathan@nathankozyra.com

Serving data from the datastore to the client

Obviously, if we start setting data via API endpoint—albeit crudely—we'll also want to retrieve the data via another API endpoint. We can easily amend our current call using the following code to include a new route that provides the data back via a request:

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

  urlParams   := mux.Vars(r)
  id       := urlParams["id"]
  ReadUser := User{}
  err := database.QueryRow("select * from users where
    user_id=?",id).Scan(&ReadUser.ID, &ReadUser.Name,
      &ReadUser.First, &ReadUser.Last, &ReadUser.Email )
  switch {
      case err == sql.ErrNoRows:
              fmt.Fprintf(w,"No such user")
      case err != nil:
              log.Fatal(err)
  fmt.Fprintf(w, "Error")
      default:
        output, _ := json.Marshal(ReadUser)
        fmt.Fprintf(w,string(output))
  }
}
```

We're doing a couple of new and noteworthy things here. First, we're using a `QueryRow()` method instead of `Exec()`. Go's default database interface offers a couple of different querying mechanisms that do slightly different things. These are as follows:

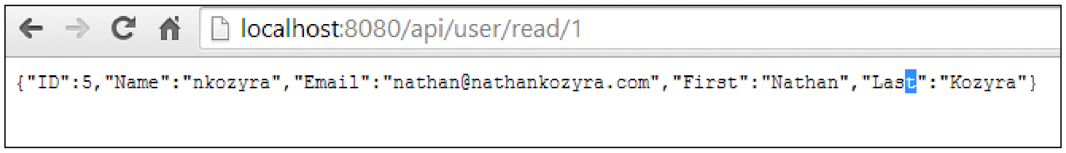
- `Exec()`: This method is used for queries (`INSERT`, `UPDATE`, and `DELETE` primarily) that will not return rows.
- `Query()`: This method is used for queries that will return one or more rows. This is usually designated for the `SELECT` queries.
- `QueryRow()`: This method is like `Query()`, but it expects just one result. This is typically a row-based request similar to the one we had in our previous example. We can then run the `Scan()` method on that row to inject the returned values into our struct's properties.

Since we're scanning the returned data into our struct, we don't get a return value. With the `err` value, we run a switch to determine how to convey a response to the user or the application that's using our API.

If we have no rows, it is likely that there is an error in the request and we'll let the recipient know that an error exists.

However, if there is a SQL error, then we'll stay quiet for now. It's a bad practice to expose internal errors to the public. However, we should respond that something went wrong without being too specific.

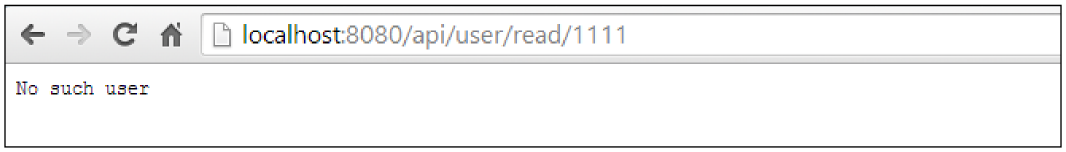
Finally, if the request is valid and we get a record, we will marshal that into a JSON response and cast it to a string before returning it. Our following result looks like what we'd expect for a valid request:



localhost:8080/api/user/read/1

```
{"ID":5,"Name":"nkozyra","Email":"nathan@nathankozyra.com","First":"Nathan","Last":"Kozyra"}
```

And then, it appropriately returns an error (as shown in the following screenshot) if we request a particular record from our users' table that does not actually exist:



localhost:8080/api/user/read/1111

```
No such user
```

Setting headers to add detail for clients

Something that will come up a bit more as we go on is the idea of using HTTP headers to convey important information about the data that we're sending or accepting via the API.

We can quickly look at the headers that are being sent through our API now by running a `curl` request against it. When we do, we'll see something like this:

```
curl --head http://localhost:8080/api/user/read/1111
HTTP/1.1 200 OK
Date: Wed, 18 Jun 2014 14:09:30 GMT
Content-Length: 12
Content-Type: text/plain; charset=utf-8
```

This is a pretty small set of headers that is sent by Go, by default. As we go forward, we may wish to append more informative headers that tell a recipient service how to handle or cache data.

Let's very briefly try to set some headers and apply them to our request using the `http` package. We'll start with one of the more basic response headers and set a Pragma. This is a `no-cache` Pragma on our result set to tell users or services that ingest our API to always request a fresh version from our database.

Ultimately, given the data we're working with, this is unnecessary in this case, but is the simplest way to demonstrate this behavior. We may find going forward that endpoint caching helps with performance, but it may not provide us with the freshest data.

The `http` package itself has a pretty simple method for both setting response headers and getting request headers. Let's modify our `GetUser` function to tell other services that they should not cache this data:

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

  w.Header().Set("Pragma","no-cache")
```

The `Header()` method returns the `Header` struct of `iowriter`, which we can then add directly using `Set()` or get by using values using `Get()`.

Now that we've done that, let's see how our output has changed:

```
curl --head http://localhost:8080/api/user/read/1111
HTTP/1.1 200 OK
Pragma: no-cache
Date: Wed, 18 Jun 2014 14:15:35 GMT
Content-Length: 12
Content-Type: text/plain; charset=utf-8
```

As we'd expect, we now see our value directly in CURL's header information and it properly returns that this result should not be cached.

There are, of course, far more valuable response headers that we can send with web services and APIs, but this is a good start. As we move forward, we'll utilize more of these, including `Content-Encoding`, `Access-Control-Allow-Origin`, and more headers that allow us to specify what our data is, who can access it, and what they should expect in terms of formatting and encoding.

Summary

We've touched on the very basics of developing a simple web service interface in Go. Admittedly, this particular version is extremely limited and vulnerable to attack, but it shows the basic mechanisms that we can employ to produce usable, formalized output that can be ingested by other services.

At this point, you should have the basic tools at your disposal that are necessary to start refining this process and our application as a whole. We'll move forward with applying a fuller design to our API as we push forward, as two randomly chosen API endpoints will obviously not do much for us.

In our next chapter, we'll dive in deeper with API planning and design, the nitty-gritty of RESTful services, and look at how we can separate our logic from our output. We'll briefly touch on some logic/view separation concepts and move toward more robust endpoints and methods in *Chapter 3*, *Routing and Bootstrapping*.

2
RESTful Services in Go

When people typically design APIs and web services, they're making them as an afterthought or at least as the final step in a large-scale application.

There's good logic behind this—the application comes first and catering to developers when there's no product on the table doesn't make a lot of sense. So, typically when an application or website is created, that's the core product and any additional resources for APIs come second to it.

As the Web has changed in recent years, this system has changed a little bit. Now, it's not entirely uncommon to write the API or web service first and then the application. Most often, this happens with highly responsive, single-page applications or mobile applications where the structure and data are more important than the presentation layer.

Our overarching project—a social network—will demonstrate the nature of a data-and-architecture-first application. We'll have a functional social network that can be traversed and manipulated exclusively at API endpoints. However, later in this book, we will have some fun with a presentation layer.

While the concept behind this could be viewed as entirely demonstrative, the reality is that this method is behind a lot of emerging services and applications today. It's extremely common for a new site or service to launch with an API, and sometimes with nothing but an API.

In this chapter, we will examine the following topics:

- Strategies for designing an API for our application
- The basics of REST
- Other web service architectures and methods

- Encoding data and choosing data formats
- REST actions and what they do
- Creating endpoints with Gorilla's mux
- Approaches to versioning your application

Designing our application

When we set out to build our larger social network application, we have a general idea about our datasets and relationships. When we extend these to a web service, we have to translate not just data types to API endpoints, but relationships and actions as well.

For example, if we wish to find a user, we'll assume that the data is kept in a database called `users` and we'd expect to be able to retrieve that data using the `/api/users` endpoint. This is fair enough. But, what if we wish to get a specific user? What if we wish to see if two users are connected? What if we wish to edit a user's comment on another user's photo?, and so on.

These are the things that we should consider, not just in our application but also in the web services that we build around it (or in this case, the other way around, as our web services comes first).

At this point, we have a relatively simplistic dataset for our application, so let's flush it out in such a way that we can create, retrieve, update, and delete users as well as create, retrieve, update, and delete relationships between the users. We can think of this as *friending* or *following* someone on traditional social networks.

First, let's do a little maintenance on our `users` table. Presently, we have a unique index on just the `user_nickname` variable, but let's create an index for `user_email`. This is a pretty common and logical security point, considering that, theoretically, one person is bound to any one given e-mail address. Type the following into your MySQL console:

```
ALTER TABLE `users`
  ADD UNIQUE INDEX `user_email` (`user_email`);
```

We can now only have one user per e-mail address. This makes sense, right?

Next, let's go ahead and create the basis for user relationships. These will encompass not just the friending/following concept but also the ability to block. So, let's create a table for these relationships. Again, type the following code into your console:

```
CREATE TABLE `users_relationships` (
  `users_relationship_id` INT(13) NOT NULL,
  `from_user_id` INT(10) NOT NULL,
  `to_user_id` INT(10) unsigned NOT NULL,
  `users_relationship_type` VARCHAR(10) NOT NULL,
  `users_relationship_timestamp` DATETIME NOT NULL DEFAULT
    CURRENT_TIMESTAMP,
  PRIMARY KEY (`users_relationship_id`),
  INDEX `from_user_id` (`from_user_id`),
  INDEX `to_user_id` (`to_user_id`),
  INDEX `from_user_id_to_user_id` (`from_user_id`, `to_user_id`),

  INDEX `from_user_id_to_user_id_users_relationship_type` (`from_user_
id`, `to_user_id`, `users_relationship_type`)
)
```

What we've done here is created a table for all of our relationships that include keys on the various users as well as the timestamp field to tell us when the relationships were created.

So, where are we? Well, right now, we have the capability to create, retrieve, update, and delete both user information as well relationships between the users. Our next step would be to conceptualize some API endpoints that will allow consumers of our web service to do this.

In the previous chapter, we created our first endpoints, `/api/user/create` and `/api/user/read`. However, if we want to be able to fully control the data we just discussed, we'll need more than that.

Before that though, let's talk a little bit about the most important concepts that relate to web services, particularly those utilizing REST.

Looking at REST

So, what is REST exactly, and where did it come from? To start with, REST stands for **Representational state transfer**. This is important because the representation of data (and its metadata) is the critical part of data transfer.

The **state** aspect of the acronym is slightly misleading because statelessness is actually a core component of the architecture.

In short, REST presents a simple, stateless mechanism for presenting data over HTTP (and some other protocols) that is uniform and includes a control mechanism such as caching directives.

The architecture initially arose as part of Roy Fielding's dissertation at UC Irvine. Since then, it has become codified and standardized by **World Wide Web Consortium** (**W3C**).

A RESTful application or API will require several important components, and we'll outline these now.

Making a representation in an API

The most important component of the API is the data we'll pass along as part of our web service. Usually, it's formatted text in the format of JSON, RSS/XML, or even binary data.

For the purpose of designing a web service, it's a good practice to make sure that your format matches your data. For example, if you've created a web service for passing image data, it's tempting to jam that sort of data into a text format. It's not unusual to see binary data translated into Base64 encoding and sent via JSON.

However, an important consideration with APIs is thrift, in terms of data size. If we take our earlier example and encode our image data in Base64, we end up with an API payload that will be nearly 40 percent larger. By doing this, we will increase latency in our service and introduce a potential annoyance. There is no reason to do this if we can reliably transfer the data as it exists.

The representation in the model should also serve an important role—to satisfy all requirements for the client to update, remove, or retrieve such a particular resource.

Self-description

When we say self-description, we can also describe this as self-contained to encompass two core components of REST—that a response should include everything necessary for the client per request and that it should include (either explicitly or implicitly) the information on how to handle the information.

The second part refers to cache rules, which we very briefly touched on in *Chapter 1, Our First API in Go*.

It may go without saying but providing valuable caching information about a resource contained by an API request is important. It eliminates redundant or unnecessary requests down the road.

This also brings in the concept of the stateless nature of REST. By this we mean that each request exists on its own. As mentioned earlier, any single request should include everything necessary to satisfy that request.

More than anything, this means dropping the idea of a normal web architecture where you can set cookies or session variables. This is inherently not RESTful. For one, it's unlikely that our clients would support cookies or continuous sessions. But more importantly, it reduces the comprehensive and explicit nature of responses expected from any given API endpoint.

[Automated processes and scripts can, of course, handle sessions and they could handle them as the initial proposal of REST. This is more a matter of demonstration than a reason why REST rejects a persistent state as part of its ethos.]

The importance of a URI

For reasons that we'll touch on later in this chapter, the URI or URL is one of the most critical factors in a good API design. There are several reasons for this:

- The URI should be informative. We should have information on not just the data endpoints but also on what data we might expect to see in return. Some of this is idiomatic to programmers. For example, `/api/users` would imply that we're looking for a set of users, whereas `/api/users/12345` would indicate that we're expecting to get information about a specific user.
- The URI should not break in the future. Soon, we'll talk about versioning, but this is just one place where the expectation of a stable resource endpoint is incredibly important. If the consumers of your service find missing or broken links in their applications over time without warning, this would result in a very poor user experience.
- No matter how much foresight you have in developing your API or web service, things will change. With this in mind, we should react to changes by utilizing HTTP status codes to indicate new locations or errors with present URIs rather than allowing them to simply break.

HATEOAS

HATEOAS stands for **Hypermedia as the Engine of Application State**, and it's a primary constraint of URIs in a REST architecture. The core principles behind it require that APIs should not reference fixed resource names or the actual hierarchies themselves, but they should rather focus on describing the media requested and/or define the application state.

[You can read more about REST and its requirements as defined by its original author by visiting Roy Fielding's blog at `http://roy.gbiv.com/untangled/`.]

Other API architectures

Beyond REST, we'll look at and implement a few other common architectures for APIs and web services in this book.

For the most part, we'll focus on REST APIs but we will also go into SOAP protocols and APIs for XML ingestion as well as newer asynchronous and web socket based services that allow persistence.

RPC

Remote procedure calls, or **RPC,** is a communication method that has existed for a long time and makes up the bones of what later became REST. While there is some merit for using RPC still—in particular JSON-RPC—we're not going to put much effort into accommodating it in this book.

If you're unfamiliar with RPC in general, its core difference as compared to REST is that there is a single endpoint and the requests themselves define the behaviors of the web service.

[To read more about JSON-RPC, go to `http://json-rpc.org/`.]

Choosing formats

The matter of formats used to be a much trickier subject than it is today. Where we once had myriad formats that were specific to individual languages and developers, the API world has caused this breadth of formats to shrink a bit.

The rise of Node and JavaScript as a lingua franca among data transmission formats has allowed most APIs to think of JSON first. JSON is a relatively tight format that has support in almost every major language now, and Go is no exception.

JSON

The following is a quick and simple example of how simply Go can send and receive JSON data using the core packages:

```
package main

import
(
  "encoding/json"
  "net/http"
  "fmt"
)

type User struct {
  Name string `json:"name"`
  Email string `json:"email"`
  ID int `json:"int"`
}

func userRouter(w http.ResponseWriter, r *http.Request) {
  ourUser := User{}
  ourUser.Name = "Bill Smith"
  ourUser.Email = "bill.smith@example.com"
  ourUser.ID = 100

  output,_ := json.Marshal(&ourUser)
  fmt.Fprintln(w, string(output))
}

func main() {

  fmt.Println("Starting JSON server")
  http.HandleFunc("/user", userRouter)
  http.ListenAndServe(":8080",nil)

}
```

One thing to note here are the JSON representations of our variables in the `User` struct. Anytime you see data within the grave accent (`) characters, this represents a rune. Although a string is represented in double quotes and a char in single, the accent represents Unicode data that should remain constant. Technically, this content is held in an `int32` value.

In a struct, a third parameter in a variable/type declaration is called a tag. These are noteworthy for encoding because they have direct translations to JSON variables or XML tags.

Without a tag, we'll get our variable names returned directly.

XML

As mentioned earlier, XML was once the format of choice for developers. And although it's taken a step back, almost all APIs today still present XML as an option. And of course, RSS is still the number one syndication format.

As we saw earlier in our SOAP example, marshaling data into XML is simple. Let's take the data structure that we used in the earlier JSON response and similarly marshal it into the XML data in the following example.

Our `User` struct is as follows:

```
type User struct {
  Name string `xml:"name"`
  Email string `xml:"email"`
  ID int `xml:"id"`
}
```

And our obtained output is as follows:

```
  ourUser := User{}
  ourUser.Name = "Bill Smith"
  ourUser.Email = "bill.smith@example.com"
  ourUser.ID = 100

  output,_ := xml.Marshal(&ourUser)
  fmt.Fprintln(w, string(output))
```

YAML

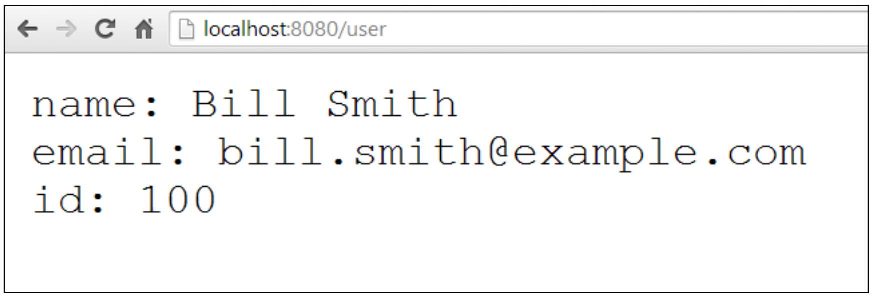
YAML was an earlier attempt to make a human-readable serialized format similar to JSON. There does exist a Go-friendly implementation of YAML in a third-party plugin called `goyaml`.

You can read more about `goyaml` at `https://godoc.org/launchpad.net/goyaml`. To install `goyaml`, we'll call a `go get launchpad.net/goyaml` command.

As with the default XML and JSON methods built into Go, we can also call `Marshal` and `Unmarshal` on YAML data. Using our preceding example, we can generate a YAML document fairly easily, as follows:

```
package main

import
(
  "fmt"
  "net/http"
  "launchpad.net/goyaml"
)

type User struct {
  Name string
  Email string
  ID int
}

func userRouter(w http.ResponseWriter, r *http.Request) {
  ourUser := User{}
  ourUser.Name = "Bill Smith"
  ourUser.Email = "bill.smith@example.com"
  ourUser.ID = 100

  output,_ := goyaml.Marshal(&ourUser)
  fmt.Fprintln(w, string(output))
}

func main() {
    fmt.Println("Starting YAML server")
  http.HandleFunc("/user", userRouter)
  http.ListenAndServe(":8080",nil)

}
```

The obtained output is as shown in the following screenshot:



```
name: Bill Smith
email: bill.smith@example.com
id: 100
```

CSV

The **Comma Separated Values** (**CSV**) format is another stalwart that's fallen somewhat out of favor, but it still persists as a possibility in some APIs, particularly legacy APIs.

Normally, we wouldn't recommend using the CSV format in this day and age, but it may be particularly useful for business applications. More importantly, it's another encoding format that's built right into Go.

Coercing your data into CSV is fundamentally no different than marshaling it into JSON or XML in Go because the `encoding/csv` package operates with the same methods as these subpackages.

Comparing the HTTP actions and methods

An important aspect to the ethos of REST is that data access and manipulation should be restricted by verb/method.

For example, the `GET` requests should not allow the user to modify, update, or create the data within. This makes sense. `DELETE` is fairly straightforward as well. So, what about creating and updating? However, no such directly translated verbs exist in the HTTP nomenclature.

There is some debate on this matter, but the generally accepted method for handling this is to use `PUT` to update a resource and `POST` to create it.

Here is the relevant information on this as per the W3C protocol for HTTP 1.1:

The fundamental difference between the `POST` and `PUT` requests is reflected in the different meaning of the Request-URI. The URI in a `POST` request identifies the resource that will handle the enclosed entity. This resource might be a data-accepting process, a gateway to some other protocol, or a separate entity that accepts annotations. In contrast, the URI in a `PUT` request identifies the entity enclosed with the request—the user agent knows which URI is intended and the server *MUST NOT* attempt to apply the request to some other resource. If the server desires that the request to be applied to a different URI, it MUST send a 301 (Moved Permanently) response; the user agent MAY then make its own decision regarding whether or not to redirect the request.

So, if we follow this, we can assume that the following actions will translate to the following HTTP verbs:

Actions	**HTTP verbs**
Retrieving data	`GET`
Creating data	`POST`
Updating data	`PUT`
Deleting data	`DELETE`

Thus, a `PUT` request to, say, `/api/users/1234` will tell our web service that we're accepting data that will update or overwrite the user resource data for our user with the ID `1234`.

A `POST` request to `/api/users/1234` will tell us that we'll be creating a new user resource based on the data within.

It is very common to see the update and create methods switched, such that `POST` is used to update and `PUT` is used for creation. On the one hand, it's easy enough to do it either way without too much complication. On the other hand, the W3C protocol is fairly clear.

The PATCH method versus the PUT method

So, you might think after going through the last section that everything is wrapped up, right? Cut and dry? Well, as always, there are hitches and unexpected behaviors and conflicting rules.

In 2010, there was a proposed change to HTTP that would include a `PATCH` method. The difference between `PATCH` and `PUT` is somewhat subtle, but, the shortest possible explanation is that `PATCH` is intended to supply only partial changes to a resource, whereas `PUT` is expected to supply a complete representation of a resource.

The `PATCH` method also provides the potential to essentially *copy* a resource into another resource given with the modified data.

For now, we'll focus just on `PUT`, but we'll look at `PATCH` later on, particularly when we go into depth about the `OPTIONS` method on the server side of our API.

Bringing in CRUD

The acronym **CRUD** simply stands for **Create, Read (or Retrieve), Update, and Delete**. These verbs might seem noteworthy because they closely resemble the HTTP verbs that we wish to employ to manage data within our application.

As we discussed in the last section, most of these verbs have seemingly direct translations to HTTP methods. We say "seemingly" because there are some points in REST that keep it from being entirely analogous. We will cover this a bit more in later chapters.

`CREATE` obviously takes the role of the `POST` method, `RETRIEVE` takes the place of `GET`, `UPDATE` takes the place of `PUT`/`PATCH`, and `DELETE` takes the place of, well, `DELETE`.

If we want to be fastidious about these translations, we must clarify that `PUT` and `POST` are not direct analogs to `UPDATE` and `CREATE`. In some ways this relates to the confusion behind which actions `PUT` and `POST` should provide. This all relies on the critical concept of idempotence, which means that any given operation should respond in the same way if it is called an indefinite number of times.

Idempotence is the property of certain operations in mathematics and computer science that can be applied multiple times without changing the result beyond the initial application.

For now, we'll stick with our preceding translations and come back to the nitty-gritty of `PUT` versus `POST` later in the book.

Adding more endpoints

Given that we now have a way of elegantly handling our API versions, let's take a step back and revisit user creation. Earlier in this chapter, we created some new datasets and were ready to create the corresponding endpoints.

Knowing what you know now about HTTP verbs, we should restrict access to user creation through the POST method. The example we built in the first chapter did not work exclusively with the POST request (or with POST requests at all). Good API design would dictate that we have a single URI for creating, retrieving, updating, and deleting any given resource.

With all of this in mind, let's lay out our endpoints and what they should allow a user to accomplish:

Endpoint	**Method**	**Purpose**
`/api`	`OPTIONS`	To outline the available actions within the API
`/api/users`	`GET`	To return users with optional filtering parameters
`/api/users`	`POST`	To create a user
`/api/user/123`	`PUT`	To update a user with the ID `123`
`/api/user/123`	`DELETE`	To delete a user with the ID `123`

For now, let's just do a quick modification of our initial API from *Chapter 1*, *Our First API in Go*, so that we allow user creation solely through the `POST` method.

Remember that we've used **Gorilla web toolkit** to do routing. This is helpful for handling patterns and regular expressions in requests, but it is also helpful now because it allows you to delineate based on the HTTP verb/method.

In our example, we created the `/api/user/create` and `/api/user/read` endpoints, but we now know that this is not the best practice in REST. So, our goal now is to change any resource requests for a user to `/api/users`, and to restrict creation to `POST` requests and retrievals to `GET` requests.

In our main function, we'll change our handlers to include a method as well as update our endpoint:

```
routes := mux.NewRouter()
routes.HandleFunc("/api/users", UserCreate).Methods("POST")
routes.HandleFunc("/api/users", UsersRetrieve).Methods("GET")
```

You'll note that we also changed our function names to `UserCreate` and `UsersRetrieve`. As we expand our API, we'll need methods that are easy to understand and can relate directly to our resources.

Let's take a look at how our application changes:

```
package main

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

var database *sql.DB
```

Up to this point everything is the same—we require the same imports and connections to the database. However, the following code is the change:

```
type Users struct {
  Users []User `json:"users"`
}
```

We're creating a struct for a group of users to represent our generic `GET` request to `/api/users`. This supplies a slice of the `User{}` struct:

```
type User struct {
  ID int "json:id"
  Name  string "json:username"
  Email string "json:email"
  First string "json:first"
  Last  string "json:last"
}

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

  NewUser := User{}
  NewUser.Name = r.FormValue("user")
  NewUser.Email = r.FormValue("email")
  NewUser.First = r.FormValue("first")
  NewUser.Last = r.FormValue("last")
  output, err := json.Marshal(NewUser)
  fmt.Println(string(output))
  if err != nil {
    fmt.Println("Something went wrong!")
  }
  sql := "INSERT INTO users set user_nickname='" + NewUser.Name +
    "', user_first='" + NewUser.First + "', user_last='" +
      NewUser.Last + "', user_email='" + NewUser.Email + "'"
  q, err := database.Exec(sql)
  if err != nil {
    fmt.Println(err)
  }
  fmt.Println(q)
}
```

Not much has changed with our actual user creation function, at least for now. Next, we'll look at the user data retrieval method.

```
  func UsersRetrieve(w http.ResponseWriter, r *http.Request) {
    w.Header().Set("Pragma","no-cache")
    rows,_ := database.Query("select * from users LIMIT 10")
    Response       := Users{}
    for rows.Next() {
    user := User{}
      rows.Scan(&user.ID, &user.Name, &user.First, &user.Last,
        &user.Email )
    Response.Users = append(Response.Users, user)
    }
     output,_ := json.Marshal(Response)
    fmt.Fprintln(w,string(output))
  }
```

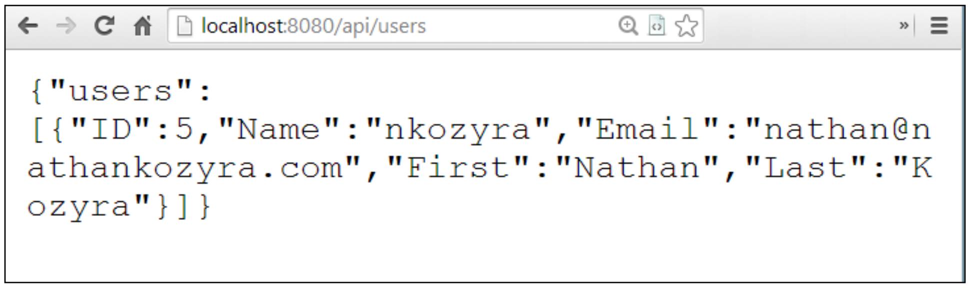
On the `UsersRetrieve()` function, we're now grabbing a set of users and scanning them into our `Users{}` struct. At this point, there isn't yet a header giving us further details nor is there any way to accept a starting point or a result count. We'll do that in the next chapter.

And finally we have our basic routes and MySQL connection in the main function:

```
  func main() {
    db, err := sql.Open("mysql", "root@/social_network")
    if err != nil {

    }
    database = db
    routes := mux.NewRouter()
    routes.HandleFunc("/api/users", UserCreate).Methods("POST")
    routes.HandleFunc("/api/users", UsersRetrieve).Methods("GET")
    http.Handle("/", routes)
    http.ListenAndServe(":8080", nil)
}
```

As mentioned earlier, the biggest differences in `main` are that we've renamed our functions and are now relegating certain actions using the `HTTP` method. So, even though the endpoints are the same, we're able to direct the service depending on whether we use the `POST` or `GET` verb in our requests.

When we visit `http://localhost:8080/api/users` (by default, a `GET` request) now in our browser, we'll get a list of our users (although we still just have one technically), as shown in the following screenshot:



```
{"users":
[{"ID":5,"Name":"nkozyra","Email":"nathan@n
athankozyra.com","First":"Nathan","Last":"K
ozyra"}]}
```

Handling API versions

Before we go nay further with our API, it's worth making a point about versioning our API.

One of the all-too-common problems that companies face when updating an API is changing the version without breaking the previous version. This isn't simply a matter of valid URLs, but it is also about the best practices in REST and graceful upgrades.

Take our current API for example. We have an introductory `GET` verb to access data, such as `/api/users` endpoint. However, what this really should be is a clone of a versioned API. In other words, `/api/users` should be the same as `/api/{current-version}/users`. This way, if we move to another version, our older version will still be supported but not at the `{current-version}` address.

So, how do we tell users that we've upgraded? One possibility is to dictate these changes via HTTP status codes. This will allow consumers to continue accessing our API using older versions such as `/api/2.0/users`. Requests here will also let the consumer know that there is a new version.

We'll create a new version of our API in *Chapter 3, Routing and Bootstrapping*.

Allowing pagination with the link header

Here's another REST point that can sometimes be difficult to handle when it comes to statelessness: how do you pass the request for the next set of results?

You might think it would make sense to do this as a data element. For example:

```
{ "payload": [ "item","item 2"], "next":
  "http://yourdomain.com/api/users?page=2" }
```

Although this may work, it violates some principles of REST. First, unless we're explicitly returning hypertext, it is likely that we will not be supplying a direct URL. For this reason, we may not want to include this value in the body of our response.

Secondly, we should be able to do even more generic requests and get information about the other actions and available endpoints.

In other words, if we hit our API simply at `http://localhost:8080/api`, our application should return some basic information to consumers about potential next steps and all the available endpoints.

One way to do this is with the link header. A **link** header is simply another header key/value that you set along with your response.

JSON responses are often not considered RESTful because they are not in a hypermedia format. You'll see APIs that embed `self`, `rel`, and `next` link headers directly in the response in unreliable formats.

JSON's primary shortcoming is its inability to support hyperlinks inherently. This is a problem that is solved by JSON-LD, which provides, among other things, linked documents and a stateless context.

Hypertext Application Language (**HAL**) attempts to do the same thing. The former is endorsed by W3C but both have their supporters. Both formats extend JSON, and while we won't go too deep into either, you can modify responses to produce either format.

Here's how we could do it in our `/api/users` `GET` request:

```
func UsersRetrieve(w http.ResponseWriter, r *http.Request) {
  log.Println("starting retrieval")
  start := 0
  limit := 10

  next := start + limit

  w.Header().Set("Pragma","no-cache")
  w.Header().Set("Link","<http://localhost:8080/api/
users?start="+string(next)+"; rel=\"next\"")

  rows,_ := database.Query("select * from users LIMIT 10")
  Response := Users{}

  for rows.Next() {

    user := User{}
    rows.Scan(&user.ID, &user.Name, &user.First, &user.Last,
      &user.Email )

    Response.Users = append(Response.Users, user)
  }

  output,_ := json.Marshal(Response)
  fmt.Fprintln(w,string(output))
}
```

This tells the client where to go for further pagination. As we modify this code further, we'll include forward and backward pagination and respond to user parameters.

Summary

At this point, you should be well-versed not only with the basic ideas of creating an API web service in REST and a few other protocols, but also in the guiding principles of the formats and protocols.

We dabbled in a few things in this chapter that we'll explore in more depth over the next few chapters, particularly MVC with the various template implementations in the Go language itself.

In the next chapter, we'll build the rest of our initial endpoints and explore more advanced routing and URL muxing.

3
Routing and Bootstrapping

After the last two chapters, you should be comfortable with creating an API endpoint, the backend database to store your most pertinent information, and mechanisms necessary to route and output your data via HTTP requests.

For the last point, other than our most basic example, we've yielded to a library for handling our URL multiplexers. This is the Gorilla web toolkit. As fantastic as this library (and its related frameworks) is, it's worth getting to know how to handle requests directly in Go, particularly to create more robust API endpoints that involve conditional and regular expressions.

While we've briefly touched on the importance of header information for the web service consumer, including status codes, we'll start digging into some important ones as we continue to scale our application.

The importance of controlling and dictating state is critical for a web service, especially (and paradoxically) in stateless systems such as REST. We say this is a paradox because while the server should provide little information about the state of the application and each request, it's important to allow the client to understand this based on the absolute minimal and standard mechanisms that we're afforded.

For example, while we may not give a page number in a list or a GET request, we want to make sure that the consumer knows how to navigate to get more or previous result sets from our application.

Similarly, we may not provide a hard error message although it exists, but our web services should be bound to some standardization as it relates to feedback that we can provide in our headers.

In this chapter, we'll cover the following topics:

- Extending Go's multiplexer to handle more complex requests
- Looking at more advanced requests in Gorilla
- Introducing RPC and web sockets in Gorilla
- Handling errors in our application and requests
- Dealing with binary data

We'll also create a couple of consumer-friendly interfaces for our web application, which will allow us to interact with our social network API for requests that require `PUT`/`POST`/`DELETE`, and later on, `OPTIONS`.

By the end of this chapter, you should be comfortable with writing routers in Go as well as extending them to allow more complex requests.

Writing custom routers in Go

As mentioned earlier, until this point, we've focused on using the Gorilla Web Toolkit for handling URL routing and multiplexers, and we've done that primarily due to the simplicity of the `mux` package within Go itself.

By simplicity, we mean that pattern matching is explicit and doesn't allow for wildcards or regular expressions using the `http.ServeMux` struct.

By looking directly into the following setup of the `http.ServeMux` code, you can see how this can use a little more nuance:

```
// Find a handler on a handler map given a path string
// Most-specific (longest) pattern wins
func (mux *ServeMux) match(path string) (h Handler, pattern
  string) {
  var n = 0
    for k, v := range mux.m {
      if !pathMatch(k, path) {
        continue
      }
      if h == nil || len(k) > n {
        n = len(k)
        h = v.h
        pattern = v.pattern
      }
    }
    return
}
```

The key part here is the `!pathMatch` function, which calls another method that specifically checks whether a path literally matches a member of a `muxEntry` map:

```
func pathMatch(pattern, path string) bool {
  if len(pattern) == 0 {
   // should not happen
    return false
  }

  n := len(pattern)
  if pattern[n-1] != '/' {
   return pattern == path
  }
  return len(path) >= n && path[0:n] == pattern
}
```

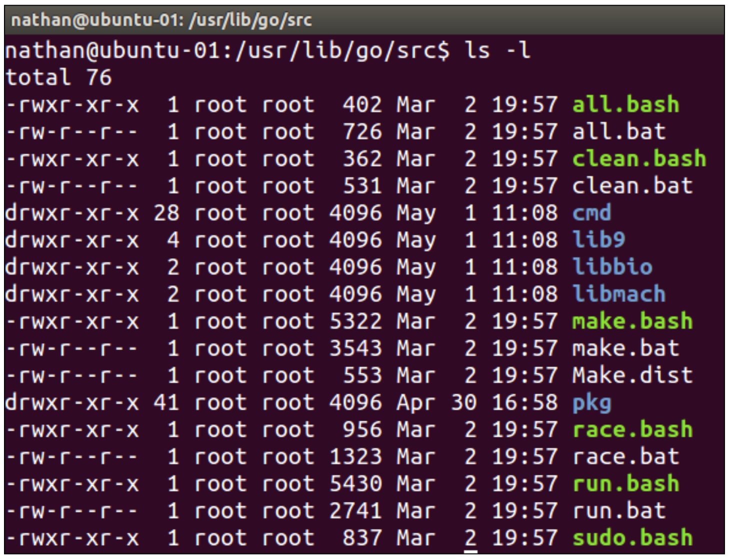
Of course, one of the best things about having access to this code is that it is almost inconsequential to take it and expand upon it.

There are two ways of doing this. The first is to write your own package, which will serve almost like an extended package. The second is to modify the code directly in your `src` directory. This option comes with the caveat that things could be replaced and subsequently broken on upgrade. So, this is an option that will fundamentally break the Go language.

With this in mind, we'll go with the first option. So, how can we extend the `http` package? The short answer is that you really can't without going into the code directly, so we'll need to create our own that inherits the most important methods associated with the various `http` structs with which we'll be dealing.

To start this, we'll need to create a new package. This should be placed in your Golang `src` directory under the domain-specific folder. In this case, we mean domain in the traditional sense, but by convention also in the web directory sense.

If you've ever executed a `go get` command to grab a third-party package, you should be familiar with these conventions. You should see something like the following screenshot in the `src` folder:



```
nathan@ubuntu-01:/usr/lib/go/src$ ls -l
total 76
-rwxr-xr-x  1 root root  402 Mar  2 19:57 all.bash
-rw-r--r--  1 root root  726 Mar  2 19:57 all.bat
-rwxr-xr-x  1 root root  362 Mar  2 19:57 clean.bash
-rw-r--r--  1 root root  531 Mar  2 19:57 clean.bat
drwxr-xr-x 28 root root 4096 May  1 11:08 cmd
drwxr-xr-x  4 root root 4096 May  1 11:08 lib9
drwxr-xr-x  2 root root 4096 May  1 11:08 libbio
drwxr-xr-x  2 root root 4096 May  1 11:08 libmach
-rwxr-xr-x  1 root root 5322 Mar  2 19:57 make.bash
-rw-r--r--  1 root root 3543 Mar  2 19:57 make.bat
-rw-r--r--  1 root root  553 Mar  2 19:57 Make.dist
drwxr-xr-x 41 root root 4096 Apr 30 16:58 pkg
-rwxr-xr-x  1 root root  956 Mar  2 19:57 race.bash
-rw-r--r--  1 root root 1323 Mar  2 19:57 race.bat
-rwxr-xr-x  1 root root 5430 Mar  2 19:57 run.bash
-rw-r--r--  1 root root 2741 Mar  2 19:57 run.bat
-rwxr-xr-x  1 root root  837 Mar  2 19:57 sudo.bash
```

In our case, we'll simply create a domain-specific folder that will hold our packages. Alternatively, you can create projects in your code repository of choice, such as GitHub, and import the packages directly from there via `go get`.

For now though, we'll simply create a subfolder under that directory, in my case `nathankozyra.com`, and then a folder called `httpex` (a portmanteau of `http` and `regex`) for the `http` extension.

Depending on your installation and operating system, your import directory may not be immediately apparent. To quickly see where your import packages should be, run the `go env` internal tool. You will find the directory under the `GOPATH` variable.

If you find your `go get` commands return the `GOPATH not set` error, you'll need to export that `GOPATH` variable. To do so, simply enter `export GOPATH=/your/directory` (for Linux or OS X). On Windows, you'll need to set an environment variable.

One final caveat is that if you're using OS X and have difficulty in getting packages via `go get`, you may need to include the `-E` flag after your `sudo` call to ensure that you're using your local user's variables and not those of the root.

For the sake of saving space, we won't include all of the code here that is necessary to retrofit the `http` package that allows regular expressions. To do so, it's important to copy all of the `ServeMux` structs, methods, and variables into your `httpex.go` file. For the most part, we'll replicate everything as is. You'll need a few important imported packages; this is what your file should look like:

```
  package httpex

import
(
  "net/http"
  "sync"
  "sync/atomic"
  "net/url"
  "path"
  "regexp"
)

type ServeMux struct {
  mu    sync.RWMutex
  m     map[string]muxEntry
  hosts bool // whether any patterns contain hostnames
}
```

The critical change happens with the `pathMatch()` function, which previously required a literal match of the longest possible string. Now, we will change any `==` equality comparisons to regular expressions:

```
// Does path match pattern?
func pathMatch(pattern, path string) bool {
  if len(pattern) == 0 {
    // should not happen
    return false
  }
  n := len(pattern)
  if pattern[n-1] != '/' {
    match,_ := regexp.MatchString(pattern,path)
    return match
  }
  fullMatch,_ := regexp.MatchString(pattern,string(path[0:n]))
  return len(path) >= n && fullMatch
}
```

If all of this seems like reinventing the wheel, the important takeaway is that—as with many things in Go—the core packages provide a great starting point for the most part, but you shouldn't hesitate to augment them when you find that certain functionality is lacking.

There is one other quick and dirty way of rolling your own `ServeMux` router, and that's by intercepting all requests and running a regular expression test on them. Like the last example, this isn't ideal (unless you wish to introduce some unaddressed efficiencies), but this can be used in a pinch. The following code illustrates a very basic example:

```
package main

import
(
  "fmt"
  "net/http"
  "regexp"
)
```

Again, we include the `regexp` package so that we can do regular expression tests:

```
func main() {

    http.HandleFunc("/", func(w http.ResponseWriter, r
      *http.Request) {

      path := r.URL.Path
      message := "You have triggered nothing"


      testMatch,_ := regexp.MatchString("/testing[0-9]{3}",path);

      if (testMatch == true) {
        // helper functions
        message = "You hit the test!"
      }


      fmt.Fprintln(w,message)
    })
```

Here, instead of giving each match a specific handler, we test within a single handler for the `testing[3 digits]` matches and then react accordingly.

In this case, we tell the client that there's nothing unless they match the pattern. This pattern will obviously work for a `/testing123` request and fail for anything that doesn't match this pattern:

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

And finally, we start our web server.

Using more advanced routers in Gorilla

Now that we've played around a bit with extending the multiplexing of the built-in package, let's see what else Gorilla has to offer.

In addition to simple expressions, we can take a URL parameter and apply it to a variable to be used later. We did this in our earlier examples without providing a lot of explanation of what we were producing.

Here's an example of how we might parlay an expression into a variable for use in an `httpHandler` function:

```
/api/users/3
/api/users/nkozyra
```

Both could be approached as `GET` requests for a specific entity within our `users` table. We could handle either with the following code:

```
mux := mux.NewRouter()
mux.HandleFunc("/api/users/[\w+\d+]", UserRetrieve)
```

However, we need to preserve the last value for use in our query. To do so, Gorilla allows us to set that expression to a key in a map. In this case, we'd address this with the following code:

```
mux.HandleFunc("/api/users/{key}", UserRetrieve)
```

This would allow us to extract that value in our handler via the following code:

```
variables := mux.Vars(r)
key := variables["key"]
```

You'll note that we used `"key"` here instead of an expression. You can do both here, which allows you to set a regular expression to a key. For example, if our user key variable consisted of letters, numbers, and dashes, we could set it like this:

```
r.HandleFunc("/api/users/{key:[A-Za-z0-9\-]}",UserRetrieve
```

And, in our `UserRetrieve` function, we'd be able to pull that key (or any other that we added to the `mux` package) directly:

```
func UserRetrieve(w http.ResponseWriter, r *http.Request) {
  urlParams := mux.Vars(r)
  key := vars["key"]
}
```

Using Gorilla for JSON-RPC

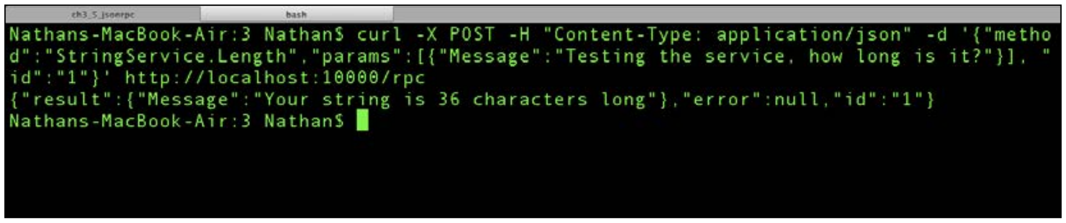
You may recall from *Chapter 2*, *RESTful Services in Go*, that we touched on RPC briefly with the promise of returning to it.

With REST as our primary method for delivery of the web service, we'll continue to limit our knowledge of RPC and JSON-RPC. However, this is a good time to demonstrate how we can create RPC services very quickly with the Gorilla toolkit.

For this example, we'll accept a string and return the number of total characters in the string via an RPC message:

```
package main

import (
  "github.com/gorilla/rpc"
  "github.com/gorilla/rpc/json"
  "net/http"
  "fmt"
  "strconv"
  "unicode/utf8"
)

type RPCAPIArguments struct {
  Message string
}

type RPCAPIResponse struct {
  Message string
}



type StringService struct{}

func (h *StringService) Length(r *http.Request, arguments
  *RPCAPIArguments, reply *RPCAPIResponse) error {
    reply.Message = "Your string is " + fmt.Sprintf("Your string is
      %d chars long", utf8.RuneCountInString(arguments.Message)) + "
      characters long"
    return nil
  }

  func main() {
    fmt.Println("Starting service")
    s := rpc.NewServer()
    s.RegisterCodec(json.NewCodec(), "application/json")
    s.RegisterService(new(StringService), "")
    http.Handle("/rpc", s)
    http.ListenAndServe(":10000", nil)
  }
```

One important note about the RPC method is that it needs to be exported, which means that a function/method must start with a capital letter. This is how Go treats a concept that is vaguely analogous to `public`/`private`. If the RPC method starts with a capital letter, it is exported outside of that package's scope, otherwise it's essentially `private`.



```
Nathans-MacBook-Air:3 Nathan$ curl -X POST -H "Content-Type: application/json" -d '{"metho
d":"StringService.Length","params":[{"Message":"Testing the service, how long is it?"}], "
id":"1"}' http://localhost:10000/rpc
{"result":{"Message":"Your string is 36 characters long"},"error":null,"id":"1"}
Nathans-MacBook-Air:3 Nathan$
```

In this case, if you called the method `stringService` instead of `StringService`, you'd get the response **can't find service stringService**.

Using services for API access

One of the issues we'll quickly encounter when it comes to building and testing our web service is handling the `POST`/`PUT`/`DELETE` requests directly to ensure that our method-specific requests do what we expect them to.

There are a few ways that exist for handling this easily without having to move to another machine or build something elaborate.

The first is our old friend cURL. By far the most popular method for making networked requests over a variety of protocols, cURL is simple and supported by almost any language you can think of.

There is no single built-in cURL component in Go. However, this largely follows the ethos of slim, integrated language design that Go's developers seem to be most interested in.

There are, however, a few third-party solutions you can look at:

- `go-curl`, a binding by ShuYu Wang is available at `https://github.com/andelf/go-curl`.
- `go-av`, a simpler method with `http` bindings is available at `https://github.com/go-av/curl`.

For the purpose of testing things out though, we can use cURL very simply and directly from the command line. It's simple enough, so constructing requests should be neither difficult nor arduous.

Here's an example call we can make to our create method at `/api/users` with a `POST` `http` method:

```
curl http://localhost:8080/api/users --data
    "name=nkozyra&email=nkozyra@gmail.com&first=nathan&last=nathan"
```

Keeping in mind that we already have this user in our database and it's a unique database field, we return an error by simply modifying our `UserCreate` function. Note that in the following code, we change our response to a new `CreateResponse` struct, which for now includes only an error string:

```
type CreateResponse struct {
  Error string "json:error"
}
```

And now, we call it. If we get an error from our database, we'll include it in our response, at least for now; shortly, we'll look at translations. Otherwise, it will be blank and we can (for now) assume that the user was successfully created. We say *for now* because we'll need to provide more information to our client depending on whether our request succeeds or fails:

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

  NewUser := User{}
  NewUser.Name = r.FormValue("user")
  NewUser.Email = r.FormValue("email")
  NewUser.First = r.FormValue("first")
  NewUser.Last = r.FormValue("last")
  output, err := json.Marshal(NewUser)
  fmt.Println(string(output))
  if err != nil {
    fmt.Println("Something went wrong!")
    }

    Response := CreateResponse{}
    sql := "INSERT INTO users SET user_nickname='" + NewUser.Name
      + "', user_first='" + NewUser.First + "', user_last='" +
        NewUser.Last + "', user_email='" + NewUser.Email + "'"
    q, err := database.Exec(sql)
    if err != nil {
      Response.Error = err.Error()
    }
    fmt.Println(q)
    createOutput,_ := json.Marshal(Response)
    fmt.Fprintln(w,string(createOutput))
  }
```

Here is how it looks if we try to create a duplicate user via a cURL request:

```
> curl http://localhost:8080/api/users -data
  "name=nkozyra&email=nkozyra@gmail.com&first=nathan&last=nathan"
{"Error": "Error 1062: Duplicate entry '' for key 'user nickname'"}
```

Using a simple interface for API access

Another way in which we can swiftly implement an interface for hitting our API is through a simple web page with a form. This is, of course, how many APIs are accessed—directly by the client instead of being handled server-side.

And although we're not suggesting this is the way our social network application should work in practice, it provides us an easy way to visualize the application:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>API Interface</title>
    <script src="http://ajax.googleapis.com/ajax/
      libs/jquery/1.11.1/jquery.min.js"></script>
    <link href="http://maxcdn.bootstrapcdn.com/
      bootstrap/3.2.0/css/bootstrap.min.css" rel="stylesheet">
    <script src="http://maxcdn.bootstrapcdn.com/
      bootstrap/3.2.0/js/bootstrap.min.js"></xscript>
    <link rel="stylesheet" href="style.css">
    <script src="script.js"></script>
  </head>
```

```
<body>

<div class="container">
    <div class="row">
<div class="col-12-lg">
      <h1>API Interface</h1>
  <div class="alert alert-warning" id="api-messages"
    role="alert"></div>

  <ul class="nav nav-tabs" role="tablist">
    <li class="active"><a href="#create" role="tab" data-
      toggle="tab">Create User</a></li>
  </ul>


  <div class="tab-content">
    <div class="tab-pane active" id="create">

    <div class="form-group">
    <label for="createEmail">Email</label>
    <input type="text" class="form-control" id="createEmail"
      placeholder="Enter email">
    </div>
    <div class="form-group">
    <label for="createUsername">Username</label>
    <input type="text" class="form-control"
      id="createUsername" placeholder="Enter username">
    </div>
    <div class="form-group">
          <label for="createFirst">First Name</label>
    <input type="text" class="form-control" id="createFirst"
      placeholder="First Name">
    </div>
    <div class="form-group">
    <label for="createLast">Last Name</label>
    <input type="text" class="form-control" id="createLast"
      placeholder="Last Name">
    </div>

    <button type="submit" onclick="userCreate();" class="btn
      btn-success">Create</button>

    </div>

  </div>
```

```
    </div>
    </div>

    </div>

    <script>

    function userCreate() {
      action = "http://localhost:8080/api/users";
      postData = {};
      postData.email  = $('#createEmail').val();
      postData.user  = $('#createUsername').val();
      postData.first  = $('#createFirst').val();
      postData.last = $('#createLast').val();


      $.post(action,postData,function(data) {
        if (data.error) {
          $('.alert').html(data.error);
          $('.alert').alert();
        }
      },'jsonp');
    }

    $(document).ready(function() {
      $('.alert').alert('close');



    });
    </script>
    </body>
  </html>
```

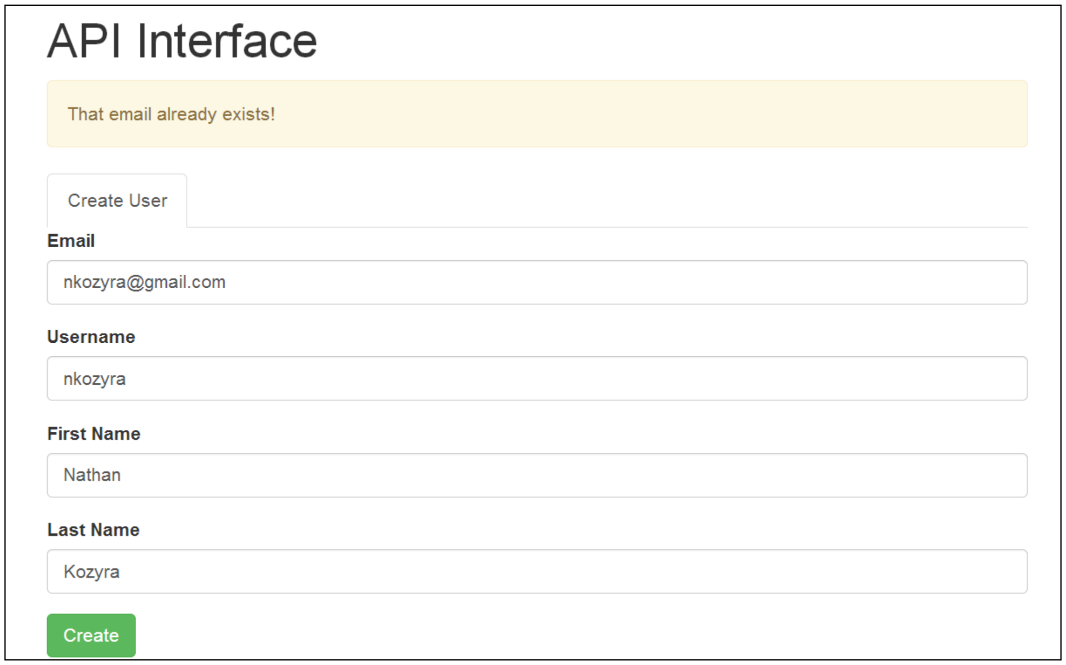
When this is rendered, we'll have a quick basic visual form for getting data into our API as well as returning valuable error information and feedback.

Due to cross-domain restrictions, you may wish to either run this from the same port and domain as our API server, or include this header with every request from the server file itself:

```
w.Header().Set("Access-Control-Allow-
    Origin","http://localhost:9000")
```

Here, `http://localhost:9000` represents the originating server for the request.

Here's what our rendered HTML presentation looks like:


API Interface

That email already exists!

Email: nkozyra@gmail.com

Username: nkozyra

First Name: Nathan

Last Name: Kozyra

Returning valuable error information

When we returned errors in our last request, we simply proxied the MySQL error and passed it along. This isn't always helpful though, because it seems to require at least some familiarity with MySQL to be valuable information for the client.

Granted, MySQL itself has a fairly clean and straightforward error messaging system, but the point is it's specific to MySQL and not our application.

What if your client doesn't understand what a "duplicate entry" means? What if they don't speak English? Will you translate the message or will you tell all of your dependencies what language to return with each request? Now you can see why this might get arduous.

Most APIs have their own system for error reporting, if for no other reason than to have control over messaging. And while it's ideal to return the language based on the request header's language, if you can't, then it's helpful to return an error code so that you (or another party) can provide a translation down the road.

And then there are the most critical errors which are returned via HTTP status codes. By default, we're producing a few of these with Go's `http` package, as any request to an invalid resource will provide a standard 404 **not found** message.

However, there are also REST-specific error codes that we'll get into shortly. For now, there's one that's relevant to our error: 409.

As per W3C's RFC 2616 protocol specification, we can send a 409 code that indicates a conflict. Here's what the spec states:

The request could not be completed due to a conflict with the current state of the resource. This code is only allowed in situations where it is expected that the user might be able to resolve the conflict and resubmit the request. The response body SHOULD include enough information for the user to recognize the source of the conflict. Ideally, the response entity would include enough information for the user or user agent to fix the problem; however, that might not be possible and is not required.

Conflicts are most likely to occur in response to a `PUT` request. For example, if versioning were being used and the entity being `PUT` included changes to a resource which conflict with those made by an earlier (third-party) request, the server might use the 409 response to indicate that it can't complete the request. In this case, the response entity would likely contain a list of the differences between the two versions in a format defined by the response `Content-Type`.

With that in mind, let's first detect an error that indicates an existing record and prevents the creation of a new record.

Unfortunately, Go does not return a specific database error code along with the error, but at least with MySQL it's easy enough to extract the error if we know the pattern used.

Using the following code, we'll construct a parser that will split a MySQL error string into its two components and return an integer error code:

```
func dbErrorParse(err string) (string, int64) {
  Parts := strings.Split(err, ":")
  errorMessage := Parts[1]
  Code := strings.Split(Parts[0],"Error ")
  errorCode,_ := strconv.ParseInt(Code[1],10,32)
  return errorMessage, errorCode
}
```

We'll also augment our `CreateResponse` struct with an error status code, represented as follows:

```
type CreateResponse struct {
  Error string "json:error"
  ErrorCode int "json:code"
}
```

We'll also take our MySQL response and message it into a `CreateResponse` struct by changing our error response behavior in the `UsersCreate` function:

```
  if err != nil {
    errorMessage, errorCode := dbErrorParse( err.Error() )
    fmt.Println(errorMessage)
    error, httpCode, msg := ErrorMessages(errorCode)
    Response.Error = msg
    Response.ErrorCode = error
    fmt.Println(httpCode)
  }
```

You'll note the `dbErrorParse` function, which we defined earlier. We take the results from this and inject it into an `ErrorMessages` function that returns granular information about any given error and not database errors exclusively:

```
type ErrMsg struct {
    ErrCode int
    StatusCode int
    Msg string
}
func ErrorMessages(err int64) (ErrMsg) {
    var em ErrMsg{}
    errorMessage := ""
    statusCode := 200;
    errorCode := 0
    switch (err) {
      case 1062:
        errorMessage = "Duplicate entry"
        errorCode = 10
        statusCode = 409
    }

    em.ErrCode = errorCode
    em.StatusCode = statusCode
    em.Msg = errorMsg

    return em

  }
```

For now, this is pretty lean, dealing with a single type of error. We'll expand upon this as we go along and add more error handling mechanisms and messages (as well as taking a stab at translation tables).

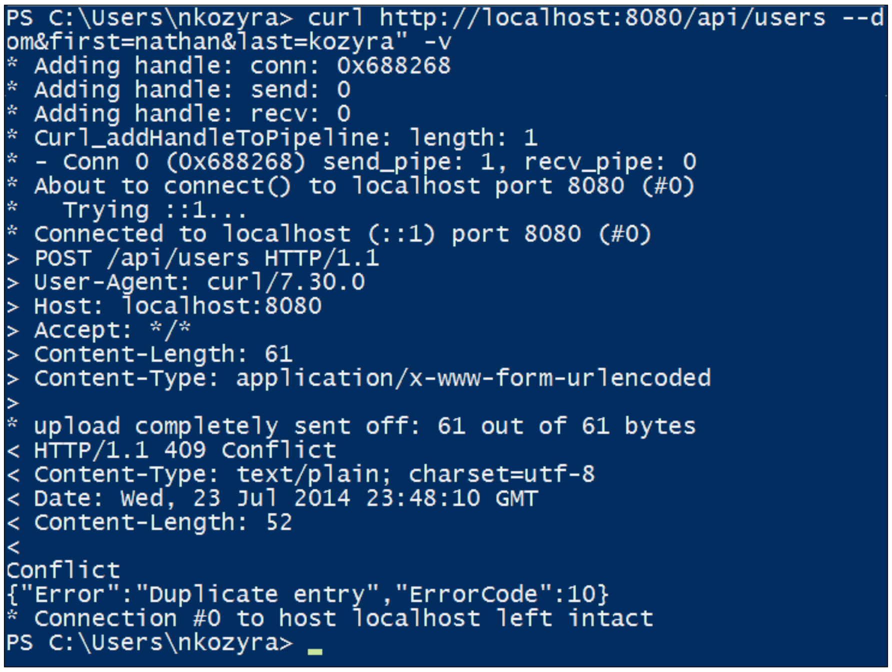
There's one last thing we need to do with regard to the HTTP status code. The easiest way to set the HTTP status code is through the `http.Error()` function:

```
http.Error(w, "Conflict", httpCode)
```

If we put this in our error conditional block, we'll return any status code we receive from the `ErrorMessages()` function:

```
if err != nil {
  errorMessage, errorCode := dbErrorParse( err.Error() )
  fmt.Println(errorMessage)
        error, httpCode, msg := ErrorMessages(errorCode)
  Response.Error = msg
  Response.ErrorCode = error
  http.Error(w, "Conflict", httpCode)
}
```

Running this again with cURL and the verbose flag (`-v`) will give us additional information about our errors, as shown in the following screenshot:



```
PS C:\Users\nkozyra> curl http://localhost:8080/api/users --d
om&first=nathan&last=kozyra" -v
* Adding handle: conn: 0x688268
* Adding handle: send: 0
* Adding handle: recv: 0
* Curl_addHandleToPipeline: length: 1
* - Conn 0 (0x688268) send_pipe: 1, recv_pipe: 0
* About to connect() to localhost port 8080 (#0)
*   Trying ::1...
* Connected to localhost (::1) port 8080 (#0)
> POST /api/users HTTP/1.1
> User-Agent: curl/7.30.0
> Host: localhost:8080
> Accept: */*
> Content-Length: 61
> Content-Type: application/x-www-form-urlencoded
>
* upload completely sent off: 61 out of 61 bytes
< HTTP/1.1 409 Conflict
< Content-Type: text/plain; charset=utf-8
< Date: Wed, 23 Jul 2014 23:48:10 GMT
< Content-Length: 52
<
Conflict
{"Error":"Duplicate entry","ErrorCode":10}
* Connection #0 to host localhost left intact
PS C:\Users\nkozyra> _
```

Handling binary data

First, we'll need to create a new field in MySQL to accommodate the image data. In the following case, we can go with `BLOB` data, which accepts large amounts of arbitrary binary data. For this purpose, we can assume (or enforce) that an image should not exceed 16 MB, so `MEDIUMBLOB` will handle all of the data that we throw at it:

```
ALTER TABLE `users`
  ADD COLUMN `user_image` MEDIUMBLOB NOT NULL AFTER `user_email`;
```

With our image column now in place, we can accept data. Add another field to our form for image data:

```
<div class="form-group">
<label for="createLast">Image</label>
<input type="file" class="form-control" name="image"
  id="createImage" placeholder="Image">
</div>
```

And in our server, we can make a few quick modifications to accept this. First, we should get the file data itself from the form, as follows:

```
    f, _, err := r.FormFile("image1")
    if err != nil {
      fmt.Println(err.Error())
    }
```

Next, we want to read this entire file and convert it to a string:

```
    fileData,_ := ioutil.ReadAll(f)
```

Then, we'll pack it into a `base64` encoded text representation of our image data:

```
    fileString := base64.StdEncoding.EncodeToString(fileData)
```

And then finally, we prepend our query with the inclusion of the new user image data:

```
sql := "INSERT INTO users set user_image='" + fileString + "',
    user_nickname='"
```

[We'll come back to a couple of these SQL statements that are assembled here in our last chapter on security.]

Summary

Three chapters in and we've got the skeleton of a simple social networking application that we can replicate in REST as well as JSON-RPC. We've also spent some time on properly relaying errors to the client in REST.

In our next chapter, *Designing APIs in Go*, we'll really begin to flesh out our social network as well as explore other Go packages that will be relevant to have a strong, robust API.

In addition, we'll bring in a few other libraries and external services to help give verbose responses to connections between our users and their relationships.

We'll also start to experiment with web sockets for a more interactive client experience on the Web. Finally, we'll handle binary data to allow our clients to upload images through our API.

4
Designing APIs in Go

We've now barreled through the basics of REST, handling URL routing, and multiplexing in Go, either directly or through a framework.

Hopefully, creating the skeleton of our API has been useful and informative, but we need to fill in some major blanks if we're going to design a functioning REST-compliant web service. Primarily, we need to handle versions, all endpoints, and the `OPTIONS` headers as well as multiple formats in an elegant, easy way that can be managed going forward.

We're going to flesh out the endpoints we want to lay out for an API-based application that allows clients to get all of the information they need about our application as well as create and update users, with valuable error information relating to both the endpoints.

By the end of this chapter, you should also be able to switch between REST and WebSocket applications as we'll build a very simple WebSocket example with a built-in client-side testing interface.

In this chapter, we'll cover the following topics:

- Outlining and designing our complete social network API
- Handling code organization and the basics of API versioning
- Allowing multiple formats (XML and JSON) for our API
- A closer look at WebSockets and implementing them in Go
- Creating more robust and descriptive error reporting
- Updating user records via the API

At the end of this chapter, you should be able to elegantly handle multiple formats and versions of your REST Web Services and have a better understanding of utilizing WebSockets within Go.

Designing our social network API

Now that we've gotten our feet wet a bit by making Go output data in our web service, one important step to take now is to fully flesh out what we want our major project's API to do.

Since our application is a social network, we need to focus not only on user information but also on connections and messaging. We'll need to make sure that new users can share information with certain groups, make and modify connections, and handle authentication.

With this in mind, let's scope out our following potential API endpoints, so that we can continue to build our application:

Endpoints	Method	Description
`/api/users`	`GET`	Return a list of users with optional parameters
`/api/users`	`POST`	Create a user
`/api/users/XXX`	`PUT`	Update a user's information
`/api/users/XXX`	`DELETE`	Delete a user
`/api/connections`	`GET`	Return a list of connections based on users
`/api/connections`	`POST`	Create a connection between users
`/api/connections/XXX`	`PUT`	Modify a connection
`/api/connections/XXX`	`DELETE`	Remove a connection between users
`/api/statuses`	`GET`	Get a list of statuses
`/api/statuses`	`POST`	Create a status
`/api/statuses/XXX`	`PUT`	Update a status
`/api/statuses/XXX`	`DELETE`	Delete a status
`/api/comments`	`GET`	Get list of comments
`/api/comments`	`POST`	Create a comment
`/api/comments/XXX`	`PUT`	Update a comment
`/api/comments/XXX`	`DELETE`	Delete a comment

In this case, any place where `XXX` exists is where we'll supply a unique identifier as part of the URL endpoint.

You'll notice that we've moved to all plural endpoints. This is largely a matter of preference and a lot of APIs use both (or only singular endpoints). The advantages of pluralized endpoints relate to consistency in the naming structure, which allows developers to have predictable calls. Using singular endpoints work as a shorthand way to express that the API call will only address a single record.

Each of these endpoints reflects a potential interaction with a data point. There is another set of endpoints that we'll include as well that don't reflect interaction with our data, but rather they allow our API clients to authenticate through OAuth:

Endpoint	**Method**	**Description**
`/api/oauth/authorize`	`GET`	Returns a list of users with optional parameters
`/api/oauth/token`	`POST`	Creates a user
`/api/oauth/revoke`	`PUT`	Updates a user's information

If you're unfamiliar with OAuth, don't worry about it for now as we'll dig in a bit deeper later on when we introduce authentication methods.

OAuth, short for **Open Authentication**, was born from a need to create a system for authenticating users with OpenID, which is a decentralized identity system.

By the time OAuth2 came about, the system had been largely retooled to be more secure as well as focus less on specific integrations. Many APIs today rely on and require OAuth to access and make changes on behalf of users via a third party.

The entire specification document (RFC6749) from the Internet Engineering Task Force can be found at `http://tools.ietf.org/html/rfc6749`.

The endpoints mentioned earlier represent everything that we'll need to build a minimalistic social network that operates entirely on a web service. We will be building a basic interface for this too, but primarily, we're focusing on building, testing, and tuning our application at the web service level.

One thing that we won't address here is the `PATCH` requests, which as we mentioned in the previous chapter, refer to partial updates of data.

In the next chapter, we will augment our web service to allow the `PATCH` updates, and we'll outline all our endpoints as part of our `OPTIONS` response.

Handling our API versions

If you spend any amount of time dealing with web services and APIs across the Internet, you'll discover a great amount of variation as to how various services handle their API versions.

Not all of these methods are particularly intuitive and often they break forward and backward compatibility. You should aim to avoid this in the simplest way possible.

Consider an API that, by default, uses versioning as part of the URI: `/api/v1.1/users`.

You will find this to be pretty common; for example, this is the way Twitter handles API requests.

There are some pluses and minuses to this approach, so you should consider the potential downsides for your URI methodology.

With API versions being explicitly defined, there is no default version, which means that users always have the version they've asked for. The good part of this is that you won't necessarily break anyone's API by upgrading. The bad part is that users may not know which version is the latest without checking explicitly or validating descriptive API messages.

As you may know, Go doesn't allow conditional imports. Although this is a design decision that enables tools such as `go fmt` and `go fix` to work quickly and elegantly, it can sometimes hamper application design.

For example, something like this is not directly possible in Go:

```
if version == 1 {
  import "v1"
} else if version == 2 {
  import "v2"
}
```

We can improvise around this a bit though. Let's assume that our application structure is as follows:

```
socialnetwork.go
/{GOPATH}/github.com/nkozyra/gowebservice/v1.go
/{GOPATH}/github.com/nkozyra/gowebservice/v2.go
```

We can then import each as follows:

```
import "github.com/nkozyra/gowebservice/v1"
import "github.com/nkozyra/gowebservice/v2"
```

This, of course, also means that we need to use these in our application, otherwise Go will trigger a compile error.

An example of maintaining multiple versions can be seen as follows:

```
package main

import
(
  "nathankozyra.com/api/v1"
  "nathankozyra.com/api/v2"
)

func main() {

  v := 1


  if v == 1 {
    v1.API()
    // do stuff with API v1
  } else {
    v2.API()
    // do stuff with API v2
  }

}
```

The unfortunate reality of this design decision is that your application will break one of programming cardinal rules: *don't duplicate code*.

This isn't a hard and fast rule of course, but duplicating code leads to functionality creep, fragmentation, and other headaches. As long as we make primary methods to do the same things across versions, we can mitigate these problems to some degree.

In this example, each of our API versions will import our standard API serving-and-routing file, as shown in the following code:

```
package v2

import
(
  "nathankozyra.com/api/api"
)

type API struct {
}

func main() {
  api.Version = 1
  api.StartServer()
}
```

And, of course, our v2 version will look nearly identical to a different version. Essentially, we use these as wrappers that bring in our important shared data such as database connections, data marshaling, and so on.

To demonstrate this, we could put a couple of our essential variables and functions into our `api.go` file:

```
package api

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

var Database *sql.DB

type Users struct {
  Users []User `json:"users"`
}

type User struct {
  ID int "json:id"
  Name  string "json:username"
  Email string "json:email"
  First string "json:first"
  Last  string "json:last"
}

func StartServer() {

  db, err := sql.Open("mysql", "root@/social_network")
  if err != nil {
  }
  Database = db
  routes := mux.NewRouter()

  http.Handle("/", routes)
  http.ListenAndServe(":8080", nil)
}
```

If this looks familiar, this is because it's the core of what we had in our first attempt at the API from the last chapter, with a few of the routes stripped for space here.

Now is also a good time to mention an intriguing third-party package for handling JSON-based REST APIs—**JSON API Server** (**JAS**). JAS sits on top of HTTP (like our API) but automates a lot of the routing by directing requests to resources automatically.

JSON API Server or JAS allows a simple set of JSON-specific API tools on top of the HTTP package to augment your web service with minimal impact.

You can read more about this at `https://github.com/coocood/jas`.

You can install it via Go by using this command: `go get github.com/coocood/jas`. Delivering our API in multiple formats

At this stage, it makes sense to formalize the way we approach multiple formats. In this case, we're dealing with JSON, RSS, and generic text.

We'll get to generic text in the next chapter when we talk about templates, but for now, we need to be able to separate our JSON and RSS responses.

The easiest way to do this is to treat any of our resources as an interface and then negotiate marshaling of the data based on a request parameter.

Some APIs define the format directly in the URI. We can also do this fairly easily (as shown in the following example) within our mux routing:

```
Routes.HandleFunc("/api.{format:json|xml|txt}/user",
  UsersRetrieve).Methods("GET")
```

The preceding code will allow us to extract the requested format directly from URL parameters. However, this is also a bit of a touchy point when it comes to REST and URIs. And though it's one with some debate on either side, for our purpose, we'll use the format simply as a query parameter.

In our `api.go` file, we'll need to create a global variable called `Format`:

```
var Format string
```

And a function that we can use to ascertain the format for each respective request:

```
func GetFormat(r *http.Request) {

  Format = r.URL.Query()["format"][0]

}
```

We'll call this with each request. Although the preceding option automatically restricts to JSON, XML, or text, we can build it into the application logic as well and include a fallback to `Format` if it doesn't match the acceptable options.

We can use a generic `SetFormat` function to marshal data based on the currently requested data format:

```
func SetFormat( data interface{} )  []byte {

  var apiOutput []byte
  if Format == "json" {
    output,_ := json.Marshal(data)
    apiOutput = output
  }else if Format == "xml" {
    output,_ := xml.Marshal(data)
    apiOutput = output
  }
  return apiOutput
}
```

Within any of our endpoint functions, we can return any data resource that is passed as an interface to `SetFormat()`:

```
func UsersRetrieve(w http.ResponseWriter, r *http.Request) {
  log.Println("Starting retrieval")
  GetFormat(r)
  start := 0
  limit := 10

  next := start + limit

  w.Header().Set("Pragma","no-cache")
```

```
    w.Header().Set("Link","<http://localhost:8080/api/users?
      start="+string(next)+"; rel=\"next\"")

    rows,_ := Database.Query("SELECT * FROM users LIMIT 10")
    Response:= Users{}

    for rows.Next() {

      user := User{}
      rows.Scan(&user.ID, &user.Name, &user.First, &user.Last,
        &user.Email )

      Response.Users = append(Response.Users, user)
    }
      output := SetFormat(Response)
    fmt.Fprintln(w,string(output))
  }
```

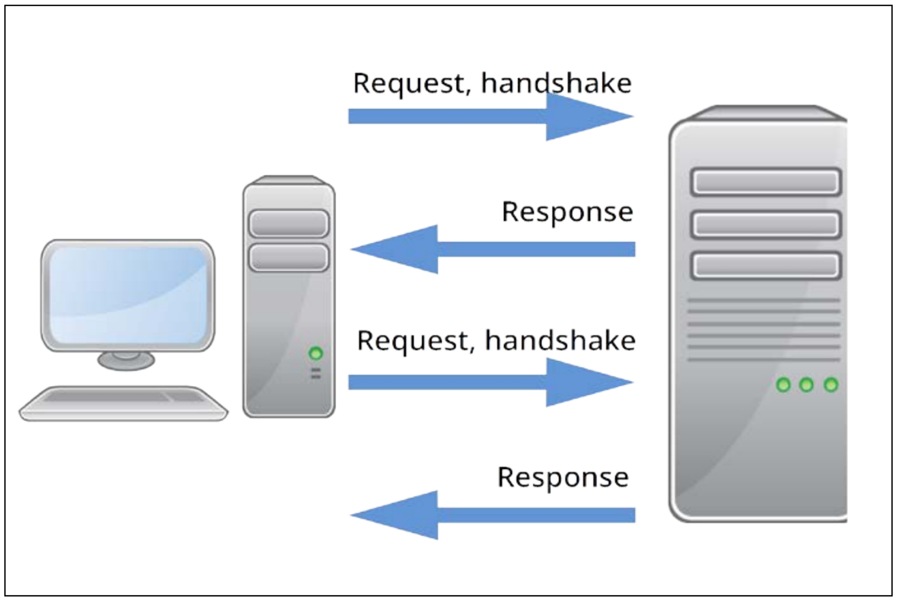
This allows us to remove the marshaling from the response function(s). Now that we have a pretty firm grasp of marshaling our data into XML and JSON, let's revisit another protocol for serving a web service.

Concurrent WebSockets

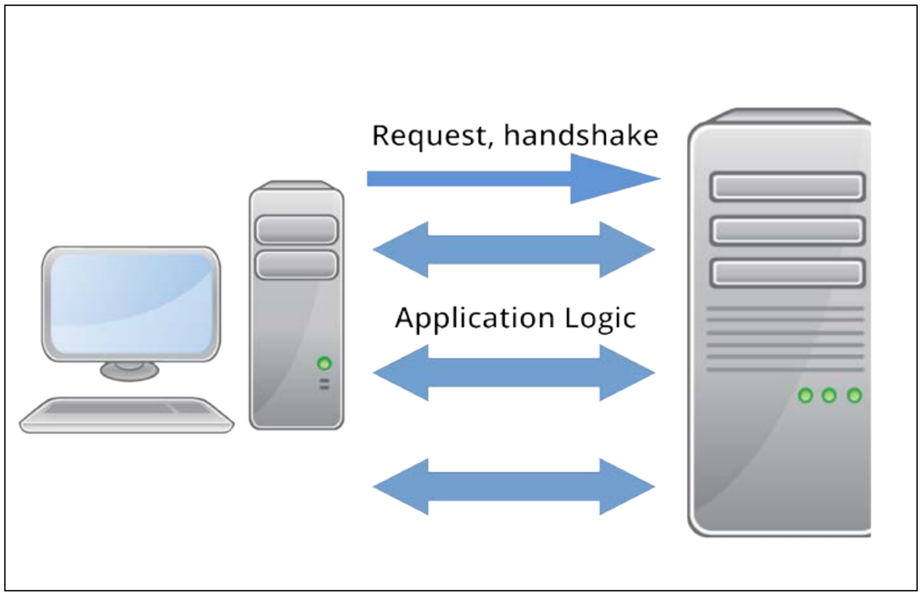
As mentioned in the previous chapter, a WebSocket is a method to keep an open connection between the client and the server, which is typically meant to replace multiple HTTP calls from a browser to a client, but also between two servers that may need to stay in a semi-reliable constant connection.

The advantages of using WebSockets for your API are reduced latency for the client and server and a generally less complex architecture for building a client-side solution for long-polling applications.

To outline the advantages, consider the following two representations; the first of the standard HTTP request:



Now compare this with the more streamlined WebSocket request over TCP, which eliminates the overhead of multiple handshakes and state control:



You can see how traditional HTTP presents levels of redundancy and latency that can hamper a long-lived application.

Granted, it's only HTTP 1 that has this problem in a strict sense. HTTP 1.1 introduced keep-alives or persistence in a connection. And while that worked on the protocol side, most nonconcurrent web servers would struggle with resource allocation. Apache, for example, by default left keep-alive timeouts very low because long-lived connections would tie up threads and prevent future requests from completing in a reasonable manner of time.

The present and future of HTTP offers some alternatives to WebSockets, namely some big options that have been brought to the table by the SPDY protocol, which was developed primarily by Google.

While HTTP 2.0 and SPDY offer concepts of multiplexing connections without closing them, particularly in the HTTP pipelining methodology, there is no wide-ranging client-side support for them yet. For the time being, if we approach an API from a web client, WebSockets provide far more client-side predictability.

It should be noted that SPDY support across web servers and load balancers is still largely experimental. Caveat emptor.

While REST remains our primary target for our API and demonstrations, you'll find a very simple WebSocket example in the following code that accepts a message and returns the length of that message along the wire:

```
package main

import (

    "fmt"
    "net/http"
    "code.google.com/p/go.net/websocket"
    "strconv"
)

var addr = ":12345"


func EchoLengthServer(ws *websocket.Conn) {

    var msg string

    for {
      websocket.Message.Receive(ws, &msg)
```

```
        fmt.Println("Got message",msg)
        length := len(msg)
        if err := websocket.Message.Send(ws,
          strconv.FormatInt(int64(length), 10) )  ; err != nil {
            fmt.Println("Can't send message length")
            break
          }
      }
```

Note the loop here; it's essential to keep this loop running within the `EchoLengthServer` function, otherwise your WebSocket connection will close immediately on the client side, preventing future messages.

```
}

func websocketListen() {

    http.Handle("/length", websocket.Handler(EchoLengthServer))
    err := http.ListenAndServe(addr, nil)
    if err != nil {
        panic("ListenAndServe: " + err.Error())
    }

}
```

This is our primary socket router. We're listening on port `12345` and evaluating the incoming message's length and then returning it. Note that we essentially *cast* the `http` handler to a `websocket` handler. This is shown here:

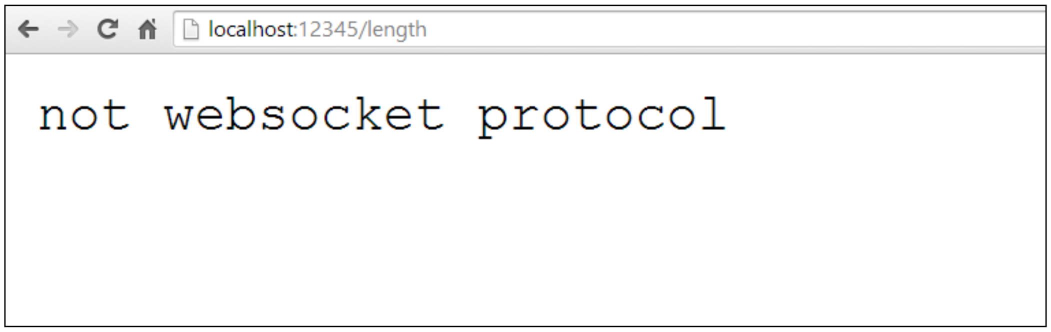
```
func main() {

    http.HandleFunc("/websocket", func(w http.ResponseWriter, r *http.
Request) {
        http.ServeFile(w, r, "websocket.html")
    })
    websocketListen()

}
```

This last piece, in addition to instantiating the WebSocket portion, also serves a flat file. Due to some cross-domain policy issues, it can be cumbersome to test client-side access and functionality of a WebSocket example unless the two are running on the same domain and port.

To manage cross-domain requests, a protocol handshake must be initiated. This is beyond the scope of the demonstration, but if you choose to pursue it, know that this particularly package does provide the functionality with a `serverHandshaker` interface that references the `ReadHandshake` and `AcceptHandshake` methods.

> The source for handshake mechanisms of `websocket.go` can be found at `https://code.google.com/p/go/source/browse/websocket/websocket.go?repo=net`.

Since this is a wholly WebSocket-based presentation at the `/length` endpoint, if you attempt to reach it via HTTP, you'll get a standard error, as shown in the following screenshot:



```
not websocket protocol
```

Hence, the flat file will be returned to the same domain and port. In the preceding code, we simply include jQuery and the built-in WebSocket support that exists in the following browsers:

- **Chrome**: Version 21 and higher versions
- **Safari**: Version 6 and higher versions
- **Firefox**: Version 21 and higher versions
- **IE**: Version 10 and higher versions
- **Opera**: Versions 22 and higher versions

Modern Android and iOS browsers also handle WebSockets now.

The code for connecting to the WebSocket-side of the server and testing some messages is as follows. Note that we don't test for WebSocket support here:

```
<html>
<head>
  <script src="http://ajax.googleapis.com/ajax/
    libs/jquery/1.11.1/jquery.min.js"></script>
```

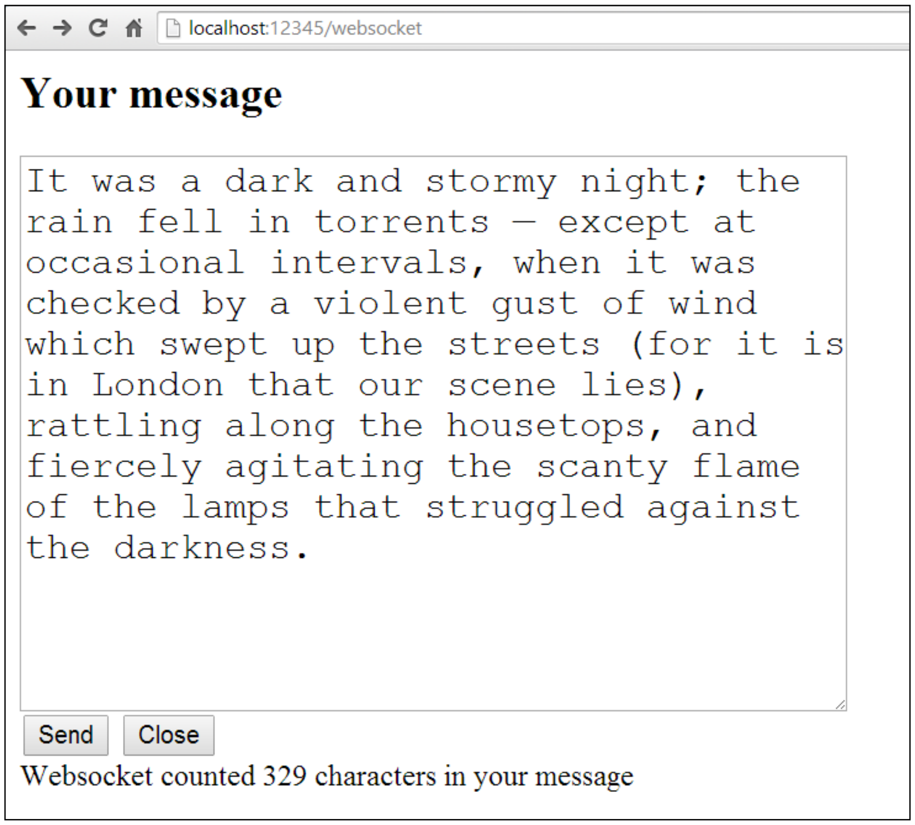
```
</head>

<body>

<script>
  var socket;

  function update(msg) {

    $('#messageArea').html(msg)

  }
```

This code returns the message that we get from the WebSocket server:

```
  function connectWS(){

    var host = "ws://localhost:12345/length";

    socket = new WebSocket(host);
    socket.onopen = function() {
      update("Websocket connected")
    }

    socket.onmessage = function(message){

      update('Websocket counted '+message.data+'
        characters in your message');
    }

    socket.onclose = function() {
      update('Websocket closed');
    }

  }

  function send() {

    socket.send($('#message').val());

  }

  function closeSocket() {

    socket.close();
  }

  connectWS();
```

```
</script>


<div>
  <h2>Your message</h2>
  <textarea style="width:50%;height:300px;font-size:20px;"
    id="message"></textarea>
  <div><input type="submit" value="Send" onclick="send()" />
    <input type="button" onclick="closeSocket();" value="Close"
      /></div>
</div>

<div id="messageArea"></div>
</body>
</html>
```

When we visit the `/websocket` URL in our browser, we'll get the text area that allows us to send messages from the client side to the WebSocket server, as shown in the following screenshot:



localhost:12345/websocket

Your message

```
It was a dark and stormy night; the
rain fell in torrents — except at
occasional intervals, when it was
checked by a violent gust of wind
which swept up the streets (for it is
in London that our scene lies),
rattling along the housetops, and
fiercely agitating the scanty flame
of the lamps that struggled against
the darkness.
```

Websocket counted 329 characters in your message

Separating our API logic

As we mentioned in the section on versioning earlier, the best way for us to achieve consistency across versions and formats is to keep our API logic separate from our overall version and delivery components.

We've seen a bit of this in our `GetFormat()` and `SetFormat()` functions, which span all the endpoints and versions.

Expanding our error messages

In the last chapter, we briefly touched on sending error messages via our HTTP status codes. In this case, we passed along a 409 status conflict when a client attempted to create a user with an e-mail address that already existed in the database.

The `http` package provides a noncomprehensive set of status codes that you can use for standard HTTP issues as well as REST-specific messages. The codes are noncomprehensive because there are some additional messages that go along with some of these codes, but the following list satisfies the RFC 2616 proposal:

Error	**Number**
`StatusContinue`	100
`StatusSwitchingProtocols`	101
`StatusOK`	200
`StatusCreated`	201
`StatusAccepted`	202
`StatusNonAuthoritativeInfo`	203
`StatusNoContent`	204
`StatusResetContent`	205
`StatusPartialContent`	206
`StatusMultipleChoices`	300
`StatusMovedPermanently`	301
`StatusFound`	302
`StatusSeeOther`	303
`StatusNotModified`	304
`StatusUseProxy`	305
`StatusTemporaryRedirect`	307
`StatusBadRequest`	400
`StatusUnauthorized`	401

Error	**Number**
`StatusPaymentRequired`	402
`StatusForbidden`	403
`StatusNotFound`	404
`StatusMethodNotAllowed`	405
`StatusNotAcceptable`	406
`StatusProxyAuthRequired`	407
`StatusRequestTimeout`	408
`StatusConflict`	409
`StatusGone`	410
`StatusLengthRequired`	411
`StatusPreconditionFailed`	412
`StatusRequestEntityTooLarge`	413
`StatusRequestURITooLong`	414
`StatusUnsupportedMediaType`	415
`StatusRequestedRangeNotSatisfiable`	416
`StatusExpectationFailed`	417
`StatusTeapot`	418
`StatusInternalServerError`	500
`StatusNotImplemented`	501
`StatusBadGateway`	502
`StatusServiceUnavailable`	503
`StatusGatewayTimeout`	504
`StatusHTTPVersionNotSupported`	505

You may recall that we hard coded this error message before; our error-handling should still be kept above the context of API versions. For example, in our `api.go` file, we had a switch control in our `ErrorMessage` function that explicitly defined our 409 HTTP status code error. We can augment this with constants and global variables that are defined in the `http` package itself:

```
func ErrorMessages(err int64) (int, int, string) {
  errorMessage := ""
  statusCode := 200;
  errorCode := 0
  switch (err) {
    case 1062:
      errorMessage = http.StatusText(409)
```

```
        errorCode = 10
        statusCode = http.StatusConflict
    }

    return errorCode, statusCode, errorMessage

  }
```

You may recall that this does some translation of errors in other components of the application; in this case 1062 was a MySQL error. We can also directly and automatically implement the HTTP status codes here as a default in the switch:

```
      default:
        errorMessage = http.StatusText(err)
        errorCode = 0
        statusCode = err
```

Updating our users via the web service

We have an ability here to present another point of potential error when we allow users to be updated via the web service.

To do this, we'll add an endpoint to the `/api/users/XXX` endpoint by adding a route:

```
    Routes.HandleFunc("/api/users/{id:[0-9]+}",
      UsersUpdate).Methods("PUT")
```

And in our `UsersUpdate` function, we'll first check to see if the said user ID exists. If it does not exist, we'll return a 404 error (a document not found error), which is the closest approximation of a resource record not found.

If the user does exist, we'll attempt to update their e-mail ID through a query; if that fails, we'll return the conflict message (or another error). If it does not fail, we'll return 200 and a success message in JSON. Here's the beginning of the `UserUpdates` function:

```
  func UsersUpdate(w http.ResponseWriter, r *http.Request) {
    Response := UpdateResponse{}
    params := mux.Vars(r)
    uid := params["id"]
    email := r.FormValue("email")

    var userCount int
```

```
     err := Database.QueryRow("SELECT COUNT(user_id) FROM users
       WHERE user_id=?", uid).Scan(&userCount)
     if userCount == 0 {

         error, httpCode, msg := ErrorMessages(404)
         log.Println(error)
         log.Println(w, msg, httpCode)
         Response.Error = msg
         Response.ErrorCode = httpCode
         http.Error(w, msg, httpCode)

     }else if err != nil {
       log.Println(error)
     } else {

       _,uperr := Database.Exec("UPDATE users SET user_email=?
          WHERE user_id=?",email,uid)
       if uperr != nil {
         _, errorCode := dbErrorParse( uperr.Error() )
         _, httpCode, msg := ErrorMessages(errorCode)

         Response.Error = msg
         Response.ErrorCode = httpCode
         http.Error(w, msg, httpCode)
       } else {
         Response.Error = "success"
         Response.ErrorCode = 0
         output := SetFormat(Response)
         fmt.Fprintln(w,string(output))
       }
     }
   }
```

We'll expand on this a bit, but for now, we can create a user, return a list of users, and update users' e-mail addresses.

While working with APIs, now is a good time to mention two browser-based tools: **Postman** and **Poster**, that let you work directly with REST endpoints from within a browser.

For more information on Postman in Chrome, go to `https://chrome.google.com/webstore/detail/postman-rest-client/fdmmgilgnpjigdojojpjoooidkmcomcm?hl=en`.

For more information on Poster in Firefox, go to `https://addons.mozilla.org/en-US/firefox/addon/poster/`.

Both tools do essentially the same thing; they allow you to interface with an API directly without having to develop a specific HTML or script-based tool or using cURL directly from the command line.

Summary

Through this chapter, we have the guts of our social networking web service scoped out and ready to fill in. We've shown you how to create and outlined how to update our users as well as return valuable error information when we cannot update our users.

This chapter has dedicated a lot of time to the infrastructure—the formats and endpoints—of such an application. On the former, we looked at XML and JSON primarily, but in the next chapter, we'll explore templates so that you can return data in any arbitrary format in which you deem necessary.

We'll also delve into authentication, either via OAuth or a simple HTTP basic authentication, which will allow our clients to connect securely to our web service and make requests that protect sensitive data. To do this, we'll also lock our application down to HTTPS for some of our requests.

In addition, we'll focus on a REST aspect that we've only touched on briefly—outlining our web service's behavior via the `OPTIONS HTTP` verb. Finally, we'll look more closely at the way headers can be used to approximate state on both the server and receiving end of a web service.

5
Templates and Options in Go

With the basics of our social networking web service fleshed out, it's time we take our project from a demo toy to something that can actually be used, and perhaps eventually in production as well.

To do this, we need to focus on a number of things, some of which we'll address in this chapter. In the last chapter, we looked at scoping out the primary functions of our social network application. Now, we need to make sure that each of those things is possible from a REST standpoint.

In order to accomplish that, in this chapter, we'll look at:

- Using `OPTIONS` to provide built-in documentation and a REST-friendly explanation of our resources' endpoints purposes
- Considering alternative output formats and an introduction on how to implement them
- Implementing and enforcing security for our API
- Allowing user registration to utilize secure passwords
- Allowing users to authenticate from a web-based interface
- Approximating an OAuth-like authentication system
- Allowing external applications to make requests on behalf of other users

After the implementation of these things, we will have the foundation of a service that will allow users to interface with it, either directly via an API or through a third-party service.

Sharing our OPTIONS

We've hinted a bit at the value and purpose of the `OPTIONS` HTTP verb as it relates to the HTTP specification and the best practices of REST.

As per RFC 2616, the HTTP/1.1 specification, responses to the `OPTIONS` requests should return information about what the client can do with the resource and/or requested endpoint.

> You can find the **HTTP/1.1 Request for Comments** (**RFC**) at `https://www.ietf.org/rfc/rfc2616.txt`.

In other words, in our early examples, calls to `/api/users` with `OPTIONS` should return an indication that `GET`, `POST`, `PUT`, and `DELETE` are presently available options at that REST resource request.

At present, there's no predefined format for what the body content should resemble or contain although the specification indicates that this may be outlined in a future release. This gives us some leeway in how we present available actions; in most such cases we will want to be as robust and informative as possible.

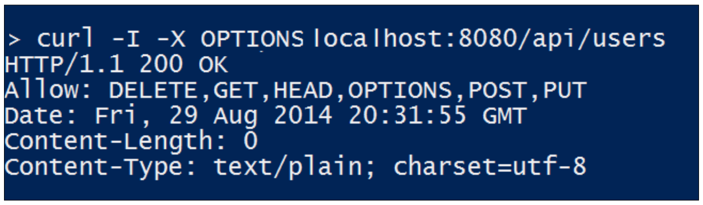
The following code is a simple modification of our present API that includes some basic information about the `OPTIONS` request that we outlined earlier. First, we'll add the method-specific handler for the request in our exported `Init()` function of the `api.go` file:

```
func Init() {
  Routes = mux.NewRouter()
  Routes.HandleFunc("/api/users", UserCreate).Methods("POST")
  Routes.HandleFunc("/api/users", UsersRetrieve).Methods("GET")
  Routes.HandleFunc("/api/users/{id:[0-9]+}",
    UsersUpdate).Methods("PUT")
  Routes.HandleFunc("/api/users", UsersInfo).Methods("OPTIONS")
}
```

And then, we'll add the handler:

```
func UsersInfo(w http.ResponseWriter, r *http.Request) {
  w.Header().Set("Allow","DELETE,GET,HEAD,OPTIONS,POST,PUT")
}
```

Calling this with cURL directly gives us what we're looking for. In the following screenshot, you'll notice the `Allow` header right at the top of the response:


```
> curl -I -X OPTIONS localhost:8080/api/users
HTTP/1.1 200 OK
Allow: DELETE,GET,HEAD,OPTIONS,POST,PUT
Date: Fri, 29 Aug 2014 20:31:55 GMT
Content-Length: 0
Content-Type: text/plain; charset=utf-8
```

This alone would satisfy most generally accepted requirements for the `OPTIONS` verb in the REST-based world, but remember that there is no format for the body and we want to be as expressive as we can.

One way in which we can do this is by providing a documentation-specific package; in this example, it is called specification. Keep in mind that this is wholly optional, but it is a nice treat for any developers who happen to stumble across it. Let's take a look at how we can set this up for self-documented APIs:

```
package specification
type MethodPOST struct {
  POST EndPoint
}
type MethodGET struct {
  GET EndPoint
}
type MethodPUT struct {
  PUT EndPoint
}
type MethodOPTIONS struct {
  OPTIONS EndPoint
}
type EndPoint struct {
  Description string `json:"description"`
  Parameters []Param `json:"parameters"`
}
type Param struct {
  Name string "json:name"
  ParameterDetails Detail `json:"details"`
}
type Detail struct {
  Type string "json:type"
  Description string `json:"description"`
  Required bool "json:required"
}
var UserOPTIONS = MethodOPTIONS{ OPTIONS: EndPoint{ Description:
  "This page" } }
var UserPostParameters = []Param{ {Name: "Email",
  ParameterDetails: Detail{Type:"string", Description: "A new
  user's email address", Required: false} } }
var UserPOST = MethodPOST{ POST: EndPoint{ Description:
  "Create a user", Parameters: UserPostParameters } }
var UserGET = MethodGET{ GET: EndPoint{ Description: "Access a
  user" }}
```

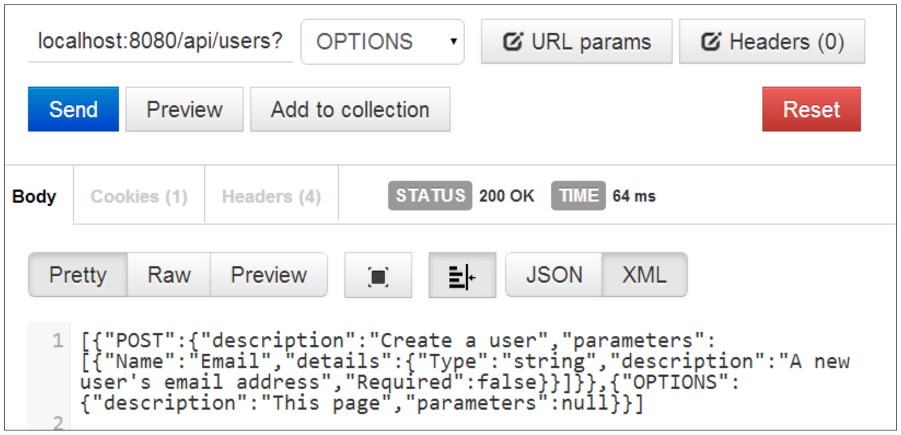
You can then reference this directly in our `api.go` file. First, we'll create a generic slice of interfaces that will encompass all the available methods:

```
type DocMethod interface {
}
```

Then, we can compile our various methods within our `UsersInfo` method:

```
func UsersInfo(w http.ResponseWriter, r *http.Request) {
  w.Header().Set("Allow","DELETE,GET,HEAD,OPTIONS,POST,PUT")
  UserDocumentation := []DocMethod{}
  UserDocumentation = append(UserDocumentation,
    Documentation.UserPOST)
  UserDocumentation = append(UserDocumentation,
    Documentation.UserOPTIONS)
  output := SetFormat(UserDocumentation)
  fmt.Fprintln(w,string(output))
}
```

Your screen should look similar to this:



```
[{"POST":{"description":"Create a user","parameters":
[{"Name":"Email","details":{"Type":"string","description":"A new
user's email address","Required":false}}]}},{"OPTIONS":
{"description":"This page","parameters":null}}]
```

Implementing alternative formats

When looking at the world of API formats, you know by now that there are two big players: **XML** and **JSON**. As human-readable formats, these two have owned the format world for more than a decade.

As is often the case, developers and technologists rarely settle happily for something for long. XML was number one for a very long time before the computational complexity of encoding and decoding as well as the verbosity of schema pushed many developers towards JSON.

JSON is not without its faults either. It's not all that readable by humans without some explicit spacing, which then increases the size of the document excessively. It doesn't handle commenting by default either.

There are a number of alternative formats that are sitting on the sideline. **YAML**, which stands for **YAML Ain't Markup Language**, is a whitespace-delimited format that uses indentation to make it extremely readable for humans. An example document would look something like this:

```
---
api:
  name: Social Network
  methods:
    - GET
    - POST
    - PUT
    - OPTIONS
    - DELETE
```

The indentation system as a method of simulating code blocks will look familiar to anyone with experience in Python.

> There are a number of YAML implementations for Go. The most noteworthy is `go-yaml` and this is available at `https://github.com/go-yaml/yaml`.

TOML, or **Tom's Obvious, Minimal Language**, takes an approach that will look very familiar to anyone who has worked with the `.ini` style config files.

Rolling our own data representation format

TOML is a good format to look at with regard to building our own data format, primarily because its simplicity lends itself to multiple ways of accomplishing the output within this format.

You may be immediately tempted to look at Go's text template format when devising something as simple as TOML because the control mechanisms to present it are largely there inherently. Take this structure and loop, for example:

```
type GenericData struct {
  Name string
  Options GenericDataBlock
}

type GenericDataBlock struct {
  Server string
  Address string
}

func main() {
  Data := GenericData{ Name: "Section", Options:
    GenericDataBlock{Server: "server01", Address: "127.0.0.1"}}

}
```

And, when the structure is parsed against the text template, it will generate precisely what we want as follows:`{{.Name}}`.

```
{{range $index, $value := Options}}
  $index = $value
{{end}}
```

One big problem with this method is that you have no inherent system for unmarshalling data. In other words, you can generate the data in this format, but you can't unravel it back into Go structures the other way.

Another issue is that as the format increases in complexity, it becomes less reasonable to use the limited control structures in the Go template library to fulfill all of the intricacies and quirks of such a format.

If you choose to roll your own format, you should avoid text templates and look at the encoding package that allows you to both produce and consume structured data formats.

We'll look at the encoding package closely in the following chapter.

Introducing security and authentication

A critical aspect of any web service or API is the ability to keep information secure and only allow access to specific users to do specific things.

Historically, there have been a number of ways to accomplish this and one of the earliest is HTTP digest authentication.

Another common one is inclusion of developer credentials, namely an API key. This isn't recommended much anymore, primarily because the security of the API relies exclusively on the security of these credentials. It is, however, largely a self-evident method for allowing authentication and as a service provider, it allows you to keep track of who is making specific requests and it also enables the throttling of requests.

The big player today is OAuth and we'll look at this shortly. However, first things first, we need to ensure that our API is accessible only via HTTPS.

Forcing HTTPS

At this point, our API is starting to enable clients and users to do some things, namely create users, update their data, and include image data for these users. We're beginning to dabble in things that we would not want to leave open in a real-world environment.

The first security step we can look at is forcing HTTPS instead of HTTP on our API. Go implements HTTPS via TLS rather than SSL since TLS is considered as a more secure protocol from the server side. One of the driving factors was vulnerabilities in SSL 3.0, particularly the Poodlebleed Bug that was exposed in 2014.

You can read more about Poodlebleed at `https://poodlebleed.com/`.

Let's look at how we can reroute any nonsecure request to its secure counterpoint in the following code:

```
package main

import
(
  "fmt"
  "net/http"
  "log"
  "sync"
)

const (
  serverName = "localhost"
  SSLport = ":443"
  HTTPport = ":8080"
  SSLprotocol = "https://"
  HTTPprotocol = "http://"
)

func secureRequest(w http.ResponseWriter, r *http.Request) {
  fmt.Fprintln(w,"You have arrived at port 443, but you are not
    yet secure.")
}
```

This is our (temporarily) correct endpoint. It's not yet TSL (or SSL), so we're not actually listening for HTTPS connections, hence the message.

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

This is our redirection handler. You'll probably take note with the `http.StatusOK` status code—obviously we'd want to send a 301 Moved Permanently error (or an `http.StatusMovedPermanently` constant). However, if you're testing this, there's a chance that your browser will cache the status and automatically attempt to redirect you.

```
func main() {
  wg := sync.WaitGroup{}
  log.Println("Starting redirection server, try to access @
    http:")

  wg.Add(1)
```

```
    go func() {
      http.ListenAndServe(HTTPport,
        http.HandlerFunc(redirectNonSecure))
      wg.Done()
    }()
    wg.Add(1)
    go func() {
      http.ListenAndServe(SSLport,http.
        HandlerFunc(secureRequest))
      wg.Done()
    }()
    wg.Wait()
  }
```

So, why have we wrapped these methods in anonymous goroutines? Well, take them out and you'll see that because the `ListenAndServe` function is blocking, we'll never run the two simultaneously by simply calling the following statements:

```
http.ListenAndServe(HTTPport,http.HandlerFunc(redirectNonSecure))
http.ListenAndServe(SSLport,http.HandlerFunc(secureRequest))
```

Of course, you have options in this regard. You could simply set the first as a goroutine and this would allow the program to move on to the second server. This method provides some more granular control for demonstration purposes.

Adding TLS support

In the preceding example, we were obviously not listening for HTTPS connections. Go makes this quite easy; however, like most SSL/TLS matters, the complication arises while handling your certificates.

For these examples, we'll be using self-signed certificates, and Go makes this easy as well. Within the `crypto/tls` package, there is a file called `generate_cert.go` that you can use to generate your certificate keys.

By navigating to your Go binary directory and then `src/pkg/crypto/tls`, you can generate a key pair that you can utilize for testing by running this:

```
go run generate_cert.go --host localhost --ca true
```

You can then take those files and move them wherever you want, ideally in the directory where our API is running.

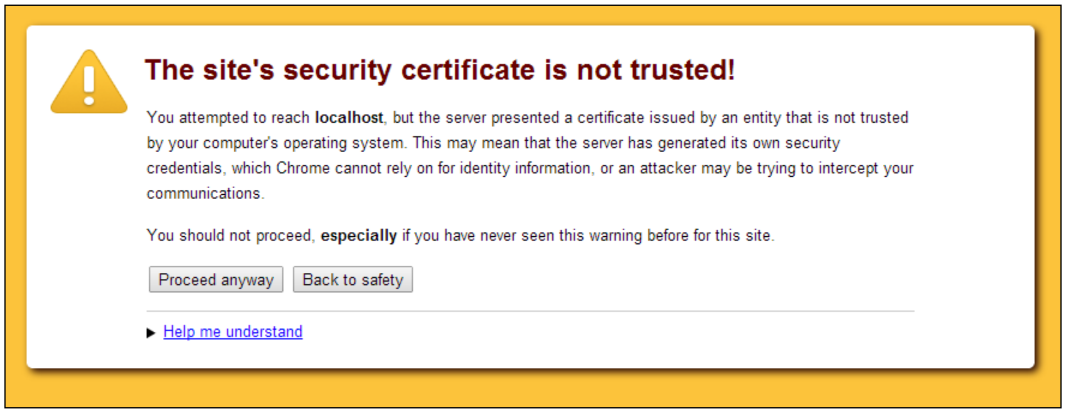
Next, let's remove our `http.ListenAndServe` function and change it to `http.ListenAndServeTLS`. This requires a couple of additional parameters that encompass the location of the keys:

```
http.ListenAndServeTLS(SSLport, "cert.pem", "key.pem",
  http.HandlerFunc(secureRequest))
```

For the sake of being more explicit, let's also modify our `secureRequest` handler slightly:

```
fmt.Fprintln(w,"You have arrived at port 443, and now you are
   marginally more secure.")
```

If we run this now and go to our browser, we'll hopefully see a warning, assuming that our browser would keep us safe:

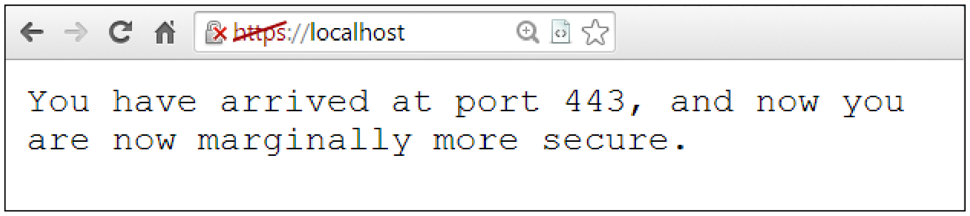


The site's security certificate is not trusted!

You attempted to reach **localhost**, but the server presented a certificate issued by an entity that is not trusted by your computer's operating system. This may mean that the server has generated its own security credentials, which Chrome cannot rely on for identity information, or an attacker may be trying to intercept your communications.

You should not proceed, **especially** if you have never seen this warning before for this site.

Assuming we trust ourselves, which is not always advisable, click through and we'll see our message from the secure handler:



```
You have arrived at port 443, and now you
are now marginally more secure.
```

And of course, if we again visit `http://localhost:8080`, we should now be automatically redirected with a 301 status code.

Creating self-signed certificates is otherwise fairly easy when you have access to an OS that supports OpenSSL.

You can get a signed (but not verified) certificate for free through a number of services for a one-year period if you'd like to experiment with real certificates and not self-signed ones. One of the more popular ones is StartSSL (`https://www.startssl.com/`), which makes getting free and paid certificates a painless process.

Letting users register and authenticate

You may recall that as part of our API application we have a self-contained interface that allows us to serve a HTML interface for the API itself. Any discussion of security goes out the door if we don't lock down our users.

Of course, the absolute simplest way of implementing user authentication security is through the storage and use of a password with a hashing mechanism. It's tragically common for servers to store passwords in clear text, so we won't do that; but, we want to implement at least one additional security parameter with our passwords.

We want to store not just the user's password, but at least a salt to go along with it. This is not a foolproof security measure, although it severely limits the threat of dictionary and rainbow attacks.

To do this, we'll create a new package called `password` as part of our suite, which allows us to generate random salts and then encrypt that value along with the password.

We can use `GenerateHash()` to both create and validate passwords.

A quick hit – generating a salt

Getting a password is simple, and creating a secure hash is also fairly easy. What we're missing to make our authentication process more secure is a salt. Let's look at how we can do this. First, let's add both a password and a salt field to our database:

```
ALTER TABLE `users`
  ADD COLUMN `user_password` VARCHAR(1024) NOT NULL AFTER
    `user_nickname`,
  ADD COLUMN `user_salt` VARCHAR(128) NOT NULL AFTER
    `user_password`,
  ADD INDEX `user_password_user_salt` (`user_password`,
    `user_salt`);
```

With this in place, let's take a look at our password package that will contain the salt and hash generation functions:

```
package password

import
(
  "encoding/base64"
  "math/rand"
  "crypto/sha256"
  "time"
)

const randomLength = 16

func GenerateSalt(length int) string {
  var salt []byte
  var asciiPad int64

  if length == 0 {
    length = randomLength
  }

  asciiPad = 32

  for i:= 0; i < length; i++ {
    salt = append(salt, byte(rand.Int63n(94) + asciiPad) )
  }

  return string(salt)
}
```

Our `GenerateSalt()` function produces a random string of characters within a certain set of characters. In this case, we want to start at 32 in the ASCII table and go up to 126.

```
func GenerateHash(salt string, password string) string {
  var hash string
  fullString := salt + password
  sha := sha256.New()
  sha.Write([]byte(fullString))
  hash = base64.URLEncoding.EncodeToString(sha.Sum(nil))

  return hash
}
```

Here, we generate a hash based on a password and a salt. This is useful not just for the creation of a password but also for validating it. The following `ReturnPassword()` function primarily operates as a wrapper for other functions, allowing you to create a password and return its hashed value:

```
func ReturnPassword(password string) (string, string) {
  rand.Seed(time.Now().UTC().UnixNano())

  salt := GenerateSalt(0)

  hash := GenerateHash(salt,password)

  return salt, hash
}
```

On our client side, you may recall that we sent all of our data via AJAX in jQuery. We had a single method on a single Bootstrap tab that allowed us to create users. First, let's remind ourselves of the tab setup.

And now, the `userCreate()` function, wherein we've added a few things. First, there's a password field that allows us to send that password along when we create a user. We may have been less comfortable about doing this before without a secure connection:

```
  function userCreate() {
    action = "https://localhost/api/users";
    postData = {};
    postData.email = $('#createEmail').val();
    postData.user = $('#createUsername').val();
    postData.first = $('#createFirst').val();
    postData.last= $('#createLast').val();
    postData.password = $('#createPassword').val();
```

Next, we can modify our `.ajax` response to react to different HTTP status codes. Remember that we are already setting up a conflict if a username or an e-mail ID already exists. So, let's handle this as well:

```
var formData = new FormData($('form')[0]);
$.ajax({

    url: action,  //Server script to process data
    dataType: 'json',
    type: 'POST',
    statusCode: {
      409: function() {
        $('#api-messages').html('Email address or nickname
          already exists!');
        $('#api-messages').removeClass
          ('alert-success').addClass('alert-warning');
        $('#api-messages').show();
        },
      200: function() {
        $('#api-messages').html('User created successfully!');
        $('#api-messages').removeClass
          ('alert-warning').addClass('alert-success');
        $('#api-messages').show();
        }
      },
```

Now, if we get a response of 200, we know our API-side has created the user. If we get 409, we report to the user that the e-mail address or username is taken in the alert area.

Examining OAuth in Go

As we briefly touched on in *Chapter 4*, *Designing APIs in Go*, OAuth is one of the more common ways of allowing an application to interact with a third-party app using another application's user authentication.

It's extraordinarily popular in social media services; Facebook, Twitter, and GitHub all use OAuth 2.0 to allow applications to interface with their APIs on behalf of users.

It's noteworthy here because while there are many API calls that we are comfortable leaving unrestricted, primarily the `GET` requests, there are others that are specific to users, and we need to make sure that our users authorize these requests.

Let's quickly review the methods that we can implement to enable something akin to OAuth with our server:

```
Endpoint
/api/oauth/authorize
/api/oauth/token
/api/oauth/revoke
```

Given that we have a small, largely demonstration-based service, our risk in keeping access tokens active for a long time is minimal. Long-lived access tokens obviously open up more opportunities for unwanted access by keeping the said access open to clients, who may not be observing the best security protocols.

In normal conditions, we'd want to set an expiry on a token, which we can do pretty simply by using a memcache system or a keystore with expiration times.
This allows values to die naturally, without having to explicitly destroy them.

The first thing we'll need to do is add a table for client credentials, namely `consumer_key` and `consumer_token`:

```
CREATE TABLE `api_credentials` (
  `user_id` INT(10) UNSIGNED NOT NULL,
  `consumer_key` VARCHAR(128) NOT NULL,
  `consumer_secret` VARCHAR(128) NOT NULL,
  `callback_url` VARCHAR(256) NOT NULL
  CONSTRAINT `FK__users` FOREIGN KEY (`user_id`) REFERENCES
    `users` (`user_id`) ON UPDATE NO ACTION ON DELETE NO ACTION
)
```

We'll check the details against our newly created database to verify credentials, and if they are correct, we'll return an access token.

An access token can be of any format; given our low security restrictions for a demonstration, we'll return an MD5 hash of a randomly generated string. In the real world, this probably wouldn't be sufficient even for a short-lived token, but it will serve its purpose here.

Remember, we implemented a random string generator as part of our `password` package. You can create a quick key and secret value in `api.go` by calling the following statements:

```
fmt.Println(Password.GenerateSalt(22))
fmt.Println(Password.GenerateSalt(41))
```

If you feed this key and secret value into the previously created table and associate it with an existing user, you'll have an active API client. Note that this may generate invalid URL characters, so we'll restrict our access to the `/oauth/token` endpoint to `POST`.

Our pseudo OAuth mechanism will go into its own package, and it will strictly generate tokens that we'll keep in a slice of tokens within our API package.

Within our core API package, we'll add two new functions to validate credentials and the `pseudoauth` package:

```
import(
Pseudoauth "github.com/nkozyra/gowebservice/pseudoauth"
)
```

The functions that we'll add are `CheckCredentials()` and `CheckToken()`. The first will accept a key, a nonce, a timestamp, and an encryption method, which we'll then hash along with the `consumer_secret` value to see that the signature matches. In essence, all of these request parameters are combined with the mutually known but unbroadcasted secret to create a signature that is hashed in a mutually known way. If those signatures correspond, the application can issue either a request token or an access token (the latter is often issued in exchange for a request token and we'll discuss more on this shortly).

In our case, we'll accept a `consumer_key` value, a nonce, a timestamp, and a signature and for the time being assume that HMAC-SHA1 is being used as the signature method. SHA1 is losing some favor do to the increased feasibility of collisions, but for the purpose of a development application, it will do and can be simply replaced later on. Go also provides SHA224, SHA256, SHA384, and SHA512 out of the box.

The purpose of the nonce and timestamp is exclusively added security. The nonce works almost assuredly as a unique identifying hash for the request, and the timestamp allows us to expire data periodically to preserve memory and/or storage. We're not going to do this here, although we will check to make sure that a nonce has not been used previously.

To begin authenticating the client, we look up the shared secret in our database:

```
func CheckCredentials(w http.ResponseWriter, r *http.Request)  {
  var Credentials string
  Response := CreateResponse{}
  consumerKey := r.FormValue("consumer_key")
  fmt.Println(consumerKey)
  timestamp := r.FormValue("timestamp")
  signature := r.FormValue("signature")
  nonce := r.FormValue("nonce")
  err := Database.QueryRow("SELECT consumer_secret from
    api_credentials where consumer_key=?",
      consumerKey).Scan(&Credentials)
    if err != nil {
    error, httpCode, msg := ErrorMessages(404)
    log.Println(error)
    log.Println(w, msg, httpCode)
    Response.Error = msg
    Response.ErrorCode = httpCode
    http.Error(w, msg, httpCode)
    return

  }
```

Here, we're taking the `consumer_key` value and looking up our shared `consumer_secret` token, which we'll pass along to our `ValidateSignature` function as follows:

```
  token,err := Pseudoauth.ValidateSignature
    (consumerKey,Credentials,timestamp,nonce,signature,0)
  if err != nil {
    error, httpCode, msg := ErrorMessages(401)
    log.Println(error)
    log.Println(w, msg, httpCode)
    Response.Error = msg
    Response.ErrorCode = httpCode
    http.Error(w, msg, httpCode)
    return
  }
```

If we find our request to be invalid (either due to incorrect credentials or an existing nonce), we'll return an unauthorized error and a 401 status code:

```
  AccessRequest := OauthAccessResponse{}
  AccessRequest.AccessToken = token.AccessToken
  output := SetFormat(AccessRequest)
  fmt.Fprintln(w,string(output))
}
```

Otherwise, we'll return the access code in a JSON body response. Here's the code for the `pseudoauth` package itself:

```
package pseudoauth
import
(
  "crypto/hmac"
  "crypto/sha1"
  "errors"
  "fmt"
  "math/rand"
  "strings"
  "time"
)
```

Nothing too surprising here! We'll need some crypto packages and `math/rand` to allow us to seed:

```
type Token struct {
  Valid bool
  Created int64
  Expires int64
  ForUser int
  AccessToken string
}
```

There's a bit more here than what we'll use at the moment, but you can see that we can create tokens with specific access rights:

```
var nonces map[string] Token
func init() {
  nonces = make(map[string] Token)
}


func ValidateSignature(consumer_key string,
    consumer_secret string, timestamp string,  nonce string,
    signature string, for_user int) (Token, error) {
  var hashKey []byte
  t := Token{}
  t.Created = time.Now().UTC().Unix()
  t.Expires = t.Created + 600
  t.ForUser = for_user

  qualifiedMessage := []string{consumer_key, consumer_secret,
    timestamp, nonce}
  fullyQualified := strings.Join(qualifiedMessage," ")

  fmt.Println(fullyQualified)
  mac := hmac.New(sha1.New, hashKey)
  mac.Write([]byte(fullyQualified))
  generatedSignature := mac.Sum(nil)

  //nonceExists := nonces[nonce]

  if hmac.Equal([]byte(signature),generatedSignature) == true {

    t.Valid = true
    t.AccessToken = GenerateToken()
    nonces[nonce] = t
    return t, nil
  } else {
    err := errors.New("Unauthorized")
    t.Valid = false
    t.AccessToken = ""
    nonces[nonce] = t
    return t, err
  }


}
```

This is a rough approximation of how services like OAuth attempt to validate signed requests; a nonce, a public key, a timestamp, and the shared private key are evaluated using the same encryption. If they match, the request is valid. If they don't match, an error should be returned.

We can use the timestamp later to give a short window for any given request so that in case of an accidental signature leak, the damage can be minimized:

```
  func GenerateToken() string {
    var token []byte
    rand.Seed(time.Now().UTC().UnixNano())
    for i:= 0; i < 32; i++ {
      token = append(token, byte(rand.Int63n(74) + 48) )
    }
    return string(token)
  }
```

Making requests on behalf of users

When it comes to making requests on behalf of users, there is a critical middle step that is involved in the OAuth2 process, and that's authentication on the part of the user. This cannot happen within a consumer application, obviously, because it would open a security risk wherein, maliciously or not, user credentials could be compromised.

Thus, this process requires a few redirects.

First, the initial request that will redirect users to a login location is required. If they're already logged in, they'll have the ability to grant access to the application. Next, our service will take a callback URL and send the user back along with their request token. This will enable a third-party application to make requests on behalf of the user, unless and until the user restricts access to the third-party application.

To store valid tokens, which are essentially permissive connections between a user and a third-party developer, we'll create a database for this:

```
CREATE TABLE `api_tokens` (
  `api_token_id` INT(10) UNSIGNED NOT NULL AUTO_INCREMENT,
  `application_user_id` INT(10) UNSIGNED NOT NULL,
  `user_id` INT(10) UNSIGNED NOT NULL,
  `api_token_key` VARCHAR(50) NOT NULL,
  PRIMARY KEY (`api_token_id`)
)
```

We'll need a few pieces to make this work, first, a login form for users who are not presently logged in, by relying on a `sessions` table. Let's create a very simple implementation in MySQL now:

```
CREATE TABLE `sessions` (
  `session_id` VARCHAR(128) NOT NULL,
  `user_id` INT(10) NOT NULL,
  UNIQUE INDEX `session_id` (`session_id`)
)
```

Next, we'll need an authorization form for users who are logged in that allows us to create a valid API access token for the user and service and redirects the user to the callback.

The template can be a very simple HTML template that can be placed at `/authorize`. So, we need to add that route to `api.go`:

```
Routes.HandleFunc("/authorize",
  ApplicationAuthorize).Methods("POST")
Routes.HandleFunc("/authorize",
  ApplicationAuthenticate).Methods("GET")
```

Requests to `POST` will check confirmation and if all is well, pass this:

```
<!DOCTYPE html>
<html>
  <head>
    <title>{{.Title}}</title>
  </head>
  <body>
  {{if .Authenticate}}
      <h1>{{.Title}}</h1>
      <form action="{{.Action}}" method="POST">
      <input type="hidden" name="consumer_key"
        value="{.ConsumerKey}" />
      Log in here
      <div><input name="username" type="text" /></div>
      <div><input name="password" type="password" /></div>
      Allow {{.Application}} to access your data?
      <div><input name="authorize" value="1" type="radio">
        Yes</div>
      <div><input name="authorize" value="0" type="radio">
        No</div>
      <input type="submit" value="Login" />
  {{end}}
  </form>
  </body>
</html>
```

Go's templating language is largely, but not completely, without logic. We can use an `if` control structure to keep both pages' HTML code in a single template.
For brevity, we'll also create a very simple `Page` struct that allows us to construct very basic response pages:

```
type Page struct {
  Title string
  Authorize bool
  Authenticate bool
  Application string
  Action string
  ConsumerKey string
}
```

We're not going to maintain login state for now, which means each user will need to log in anytime they wish to give a third party access to make API requests on their behalf. We'll fine-tune this as we go along, particularly in using secure session data and cookies that are available in the Gorilla toolkit.

So, the first request will include a login attempt with a `consumer_key` value to identify the application. You can also include the full credentials (nonce, and so on) here, but since this will only allow your application access to a single user, it's probably not necessary.

```
func ApplicationAuthenticate(w http.ResponseWriter, r
  *http.Request) {
  Authorize := Page{}
  Authorize.Authenticate = true
  Authorize.Title = "Login"
  Authorize.Application = ""
  Authorize.Action = "/authorize"

  tpl := template.Must(template.New("main")
    .ParseFiles("authorize.html"))
  tpl.ExecuteTemplate(w, "authorize.html", Authorize)
}
```

All requests will be posted to the same address, which will then allow us to validate the login credentials (remember `GenerateHash()` from our `password` package), and if they are valid, we will create the connection in `api_connections` and then return the user to the callback URL associated with the API credentials.

Here is the function that determines whether the login credentials are correct and if so, redirects to the callback URL with the `request_token` value that we created:

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

  username := r.FormValue("username")
  password := r.FormValue("password")
  allow := r.FormValue("authorize")

  var dbPassword string
  var dbSalt string
  var dbUID string

  uerr := Database.QueryRow("SELECT user_password, user_salt,
    user_id from users where user_nickname=?",
      username).Scan(&dbPassword, &dbSalt, &dbUID)
  if uerr != nil {

  }
```

With the `user_password` value, the `user_salt` value, and a submitted password value, we can verify the validity of the password by using our `GenerateHash()` function and doing a direct comparison, as they are Base64 encoded.

```
consumerKey := r.FormValue("consumer_key")
fmt.Println(consumerKey)

var CallbackURL string
var appUID string
err := Database.QueryRow("SELECT user_id,callback_url from
  api_credentials where consumer_key=?",
  consumerKey).Scan(&appUID, &CallbackURL)
if err != nil {

  fmt.Println(err.Error())
  return
}

expectedPassword := Password.GenerateHash(dbSalt, password)
if dbPassword == expectedPassword && allow == "1" {

  requestToken := Pseudoauth.GenerateToken()

  authorizeSQL := "INSERT INTO api_tokens set
    application_user_id=" + appUID + ", user_id=" + dbUID + ",
    api_token_key='" + requestToken + "' ON DUPLICATE KEY UPDATE
    user_id=user_id"

  q, connectErr := Database.Exec(authorizeSQL)
  if connectErr != nil {

  } else {
    fmt.Println(q)
  }
  redirectURL := CallbackURL + "?request_token=" + requestToken
  fmt.Println(redirectURL)
  http.Redirect(w, r, redirectURL, http.StatusAccepted)
```

After checking `expectedPassword` against the password in the database, we can tell whether the user authenticated correctly. If they did, we create the token and redirect the user back to the callback URL. It is then the responsibility of the other application to store the token for future use.

```
      } else {

        fmt.Println(dbPassword, expectedPassword)
        http.Redirect(w, r, "/authorize", http.StatusUnauthorized)
      }

    }
```

Now that we have the token on the third-party side, we can make API requests with that token and our `client_token` value to make requests on behalf of individual users, such as creating connections (friends and followers), sending automated messages, or setting status updates.

Summary

We began this chapter by looking at ways to bring in more REST-style options and features, better security, and template-based presentation. Towards this goal, we examined a basic abstraction of the OAuth security model that allows us to enable external clients to work within a user's domain.

With our application now accessible via OAuth-style authentication and secured by HTTPS, we can now expand the third-party integration of our social networking application, allowing other developers to utilize and augment our service.

In the next chapter, we'll look more at the client-side and consumer-side of our application, expanding our OAuth options and empowering more actions via the API that will include creating and deleting connections between users as well as creating status updates.

6
Accessing and Using Web Services in Go

In the previous chapter, we briefly touched on the OAuth 2.0 process and emulated this process within our own API.

We're going to explore this process a bit further in this chapter by connecting our users to a few existing ubiquitous services that offer OAuth 2.0 connectivity and allowing actions in our application to create actions in their applications.

An example of this is when you post something on one social network and are given the option to similarly post or cross-post it on another one. This is precisely the type of flow with which we'll be experimenting here.

In order to really wrap our heads around this, we'll connect existing users in our application to another one that utilizes OAuth 2.0 (such as Facebook, Google+, and LinkedIn) and then share resources between our system and the others.

While we can't make these systems return the favor, we'll continue down the road and simulate another application that is attempting to work within the infrastructure of our application.

In this chapter, we'll look at:

- Connecting to other services via OAuth 2.0 as a client
- Letting our users share information from our application to another web application
- Allowing our API consumers to make requests on behalf of our users
- How to ensure that we are making safe connections outside of OAuth requests

By the end of this chapter, as a client, you should be comfortable using OAuth to connect user accounts to other services. You should also be comfortable at making secure requests, creating ways to allow other services to connect to your services, and making third-party requests on behalf of your users.

Connecting our users to other services

To get a better understanding of how the OAuth 2.0 process works in practice, let's connect to a few popular social networks, specifically Facebook and Google+. This isn't merely a project for experimentation; it's how a great deal of modern social networks operate by allowing intercommunication and sharing among services.

Not only is this common, but it also tends to induce a higher degree of adoption when you allow seamless connections between dissonant applications. The ability to share from such sources on services such as Twitter and Facebook has helped to expedite their popularity.

As we explore the client side of things, we'll get a good grasp of how a web service like ours can allow third-party applications and vendors to work within our ecosystem and broaden the depth of our application.

To start this process, we're going to get an existing OAuth 2.0 client for Go. There are a few that are available, but to install Goauth2, run a `go get` command as follows:

```
go get code.google.com/p/goauth2/oauth
```

If we want to compartmentalize this access to OAuth 2.0 services, we can create a standalone file in our imports directory that lets us create a connection to our OAuth provider and get the relevant details from it.

In this brief example, we'll connect a Facebook service and request an authentication token from Facebook. After this, we'll return to our web service to grab and likely store the token:

```
package main

import (
  "code.google.com/p/goauth2/oauth"
  "fmt"
)
```

This is all we'll need to create a standalone package that we can call from elsewhere. In this case, we have just one service; so, we'll create the following variables as global variables:

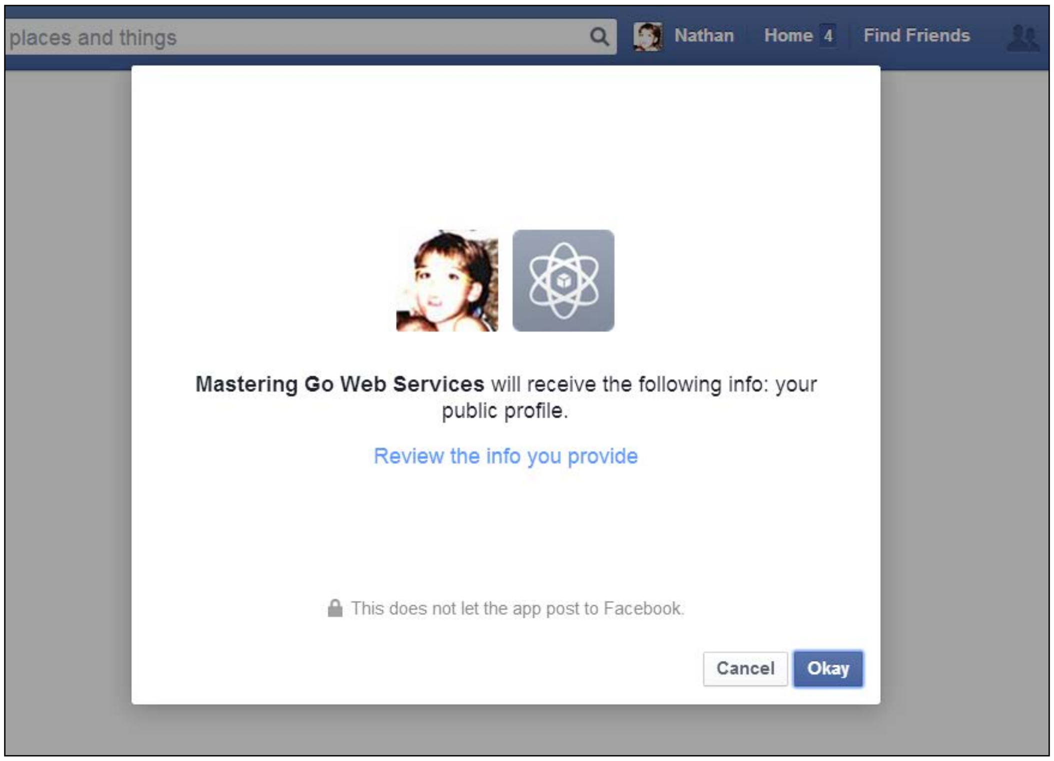
```
var (
  clientID     = "[Your client ID here]"
  clientSecret = "[Your client secret here]"
  scope        = ""
  redirectURL  = "http://www.mastergoco.com/codepass"
  authURL      = "https://www.facebook.com/dialog/oauth"
  tokenURL     = "https://graph.facebook.com/oauth/access_token"
  requestURL   = "https://graph.facebook.com/me"
  code         = ""
)
```

You will get these endpoints and variables from the provider, but they're obviously obscured here.

The `redirectURL` variable represents a place where you'll catch the sent token after a user logs in. We'll look closely at the general flow shortly. The `main` function is written as follows:

```
func main() {

  oauthConnection := &oauth.Config{
    ClientId:     clientID,
    ClientSecret: clientSecret,
    RedirectURL:  redirectURL,
    Scope:        scope,
    AuthURL:      authURL,
    TokenURL:     tokenURL,
  }


  url := oauthConnection.AuthCodeURL("")
  fmt.Println(url)

}
```

If we take the URL that's generated and visit it directly, it'll take us to the login page that is similar to the rough version that we built on the last page. Here's an authentication page that is presented by Facebook:



Mastering Go Web Services will receive the following info: your public profile.

Review the info you provide

This does not let the app post to Facebook.

If the user (in this case, me) accepts this authentication and clicks on **Okay**, the page will redirect back to our URL and pass an OAuth code along with it, which will be something like this:

```
http://www.mastergoco.com/codepass?code=h9U1_YNL1paTy-IsvQIor6u2jONwt
ipxqSbFMCo3wzYsSK7BxEVLsJ7ujtoDc
```

We can use this code as a semipermanent user acceptance code for future requests. This will not work if a user rescinds access to our application or if we choose to change the permissions that our application wishes to use in a third-party service.

You can start to see the possibilities of a very connected application and why third-party authentication systems that has the ability to sign up and sign in via Twitter, Facebook, Google+, and so on, have become viable and appealing prospects in recent years.

In order to do anything useful with this as a tie-on to our API (assuming that the terms of services of each social network allow it), we need to do three things:

First, we need to make this less restrictive than just one service. To do this, we'll create a map of the `OauthService` struct:

```
type OauthService struct {
  clientID string
  clientSecret string
  scope string
  redirectURL string
  authURL string
  tokenURL string
  requestURL string
  code string
}
```

We can then add this as per our need:

```
    OauthServices := map[string] OauthService{}

    OauthServices["facebook"] = OauthService {
      clientID:  "***",
      clientSecret: "***",
      scope: "",
      redirectURL: "http://www.mastergoco.com/connect/facebook",
      authURL: "https://www.facebook.com/dialog/oauth",
      tokenURL: "https://graph.facebook.com/oauth/access_token",
      requestURL: "https://graph.facebook.com/me",
      code: "",
    }
    OauthServices["google"] = OauthService {
      clientID:  "***.apps.googleusercontent.com",
      clientSecret: "***",
      scope: "https://www.googleapis.com/auth/plus.login",
      redirectURL: "http://www.mastergoco.com/connect/google",
      authURL: "https://accounts.google.com/o/oauth2/auth",
      tokenURL: "https://accounts.google.com/o/oauth2/token",
      requestURL: "https://graph.facebook.com/me",
      code: "",
    }
```

The next thing that we'll need to do is make this an actual redirect instead of something that spits the code into our console. With this in mind, it's time to integrate this code into the `api.go` file. This will allow our registered users to connect their user information on our social network to others, so that they can broadcast their activity on our app more globally. This brings us to our following last step, which is to accept the code that each respective web service returns:

```
func Init() {
  Routes = mux.NewRouter()
  Routes.HandleFunc("/interface", APIInterface).Methods("GET",
    "POST", "PUT", "UPDATE")
  Routes.HandleFunc("/api/users", UserCreate).Methods("POST")
  Routes.HandleFunc("/api/users", UsersRetrieve).Methods("GET")
  Routes.HandleFunc("/api/users/{id:[0-9]+}",
    UsersUpdate).Methods("PUT")
  Routes.HandleFunc("/api/users", UsersInfo).Methods("OPTIONS")
  Routes.HandleFunc("/authorize",
    ApplicationAuthorize).Methods("POST")
  Routes.HandleFunc("/authorize",
    ApplicationAuthenticate).Methods("GET")
  Routes.HandleFunc("/authorize/{service:[a-z]+}",
    ServiceAuthorize).Methods("GET")
  Routes.HandleFunc("/connect/{service:[a-z]+}",
    ServiceConnect).Methods("GET")
  Routes.HandleFunc("/oauth/token",
    CheckCredentials).Methods("POST")
}
```

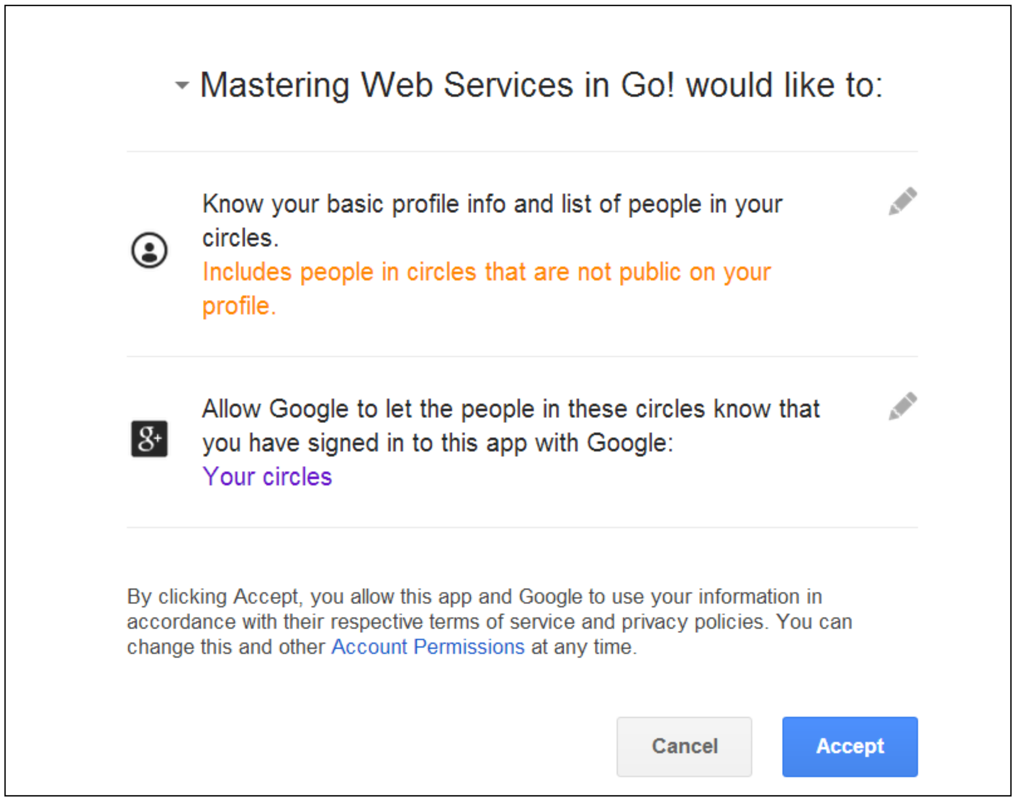
We'll add two endpoint routes to our `Init()` function; one allows a service to authorize (that is, send off to that site's OAuth authentication) and the other allows us to keep the resulting information as follows:

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

  params := mux.Vars(r)
  service := params["service"]
  redURL := OauthServices.GetAccessTokenURL(service, "")
  http.Redirect(w, r, redURL, http.StatusFound)

}
```

Here, we'll set up a Google+ authentication conduit. It goes without saying, but don't forget to replace your `clientID`, `clientSecret`, and `redirectURL` variables with your values:

```
OauthServices["google"] = OauthService {
  clientID:  "***.apps.googleusercontent.com",
  clientSecret: "***",
  scope: "https://www.googleapis.com/auth/plus.login",
  redirectURL: "http://www.mastergoco.com/connect/google",
  authURL: "https://accounts.google.com/o/oauth2/auth",
  tokenURL: "https://accounts.google.com/o/oauth2/token",
  requestURL: "https://accounts.google.com",
  code: "",
}
```

By visiting `http://localhost/authorize/google`, we'll get kicked to the interstitial authentication page of Google+. Here's an example that is fundamentally similar to the Facebook authentication that we saw earlier:



Mastering Web Services in Go! would like to:

Know your basic profile info and list of people in your circles.
Includes people in circles that are not public on your profile.

Allow Google to let the people in these circles know that you have signed in to this app with Google:
Your circles

By clicking Accept, you allow this app and Google to use your information in accordance with their respective terms of service and privacy policies. You can change this and other Account Permissions at any time.

When a user clicks on **Accept**, we'll be returned to our redirect URL with the code that we're looking for.

For most OAuth providers, a client ID and a client secret will be provided from a dashboard.

However, on Google+, you'll retrieve your client ID from their Developers console, which allows you to sign up new apps and request access to different services. They don't openly present a client secret though, so you'll need to download a JSON file that contains not only the secret, but also other relevant data that you might need to access the service in a format similar to this:

{"web":{"auth_uri":"https://accounts.google.com/o/oauth2/auth","client_secret":"***","token_uri":"https://accounts.google.com/o/oauth2/token","client_email":"***@developer.gserviceaccount.com","client_x509_cert_url":"https://www.googleapis.com/robot/v1/metadata/x509/***@developer.gserviceaccount.com","client_id":"***.apps.googleusercontent.com","auth_provider_x509_cert_url":"https://www.googleapis.com/oauth2/v1/certs"}}

You can grab the pertinent details directly from this file.

Of course, to ensure that we know who made the request and how to store it, we'll need some sense of state.

Saving the state with a web service

There are quite a few ways to save state within a single web request. However, things tend to get more complicated in a situation like this wherein our client makes one request, he or she is then redirected to another URL, and then comes back to our.

We can pass some information about the user in our redirect URL, for example, `http://mastergoco.com/connect/google?uid=1`; but this is somewhat inelegant and opens a small security loophole wherein a man-in-the-middle attacker could find out about a user and an external OAuth code.

The risk here is small but real enough; therefore, we should look elsewhere. Luckily, Gorilla also provides a nice library for secure sessions. We can use these whenever we've verified the identity of a user or client and store the information in a cookie store.

To get started, let's create a `sessions` table:

```
CREATE TABLE IF NOT EXISTS `sessions` (
  `session_id` varchar(128) NOT NULL,
  `user_id` int(10) NOT NULL,
  `session_start_time` int(11) NOT NULL,
  `session_update_time` int(11) NOT NULL,
  UNIQUE KEY `session_id` (`session_id`)
)
```

Next, include the `sessions` package:

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

Then, move it into the `import` section of our `api.go` file:

```
import (
  ...
  "github.com/gorilla/mux"
  "github.com/gorilla/sessions"
```

Right now we're not authenticating the service, so we'll enforce that on our `ApplicationAuthorize` (`GET`) handler:

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

  params := mux.Vars(r)
  service := params["service"]

  loggedIn := CheckLogin()
  if loggedIn == false {
    redirect = url.QueryEscape("/authorize/" + service)
    http.Redirect(w, r, "/authorize?redirect="+redirect,
      http.StatusUnauthorized)
    return
  }

  redURL := OauthServices.GetAccessTokenURL(service, "")
  http.Redirect(w, r, redURL, http.StatusFound)

}
```

Now, if a user attempts to connect to a service, we'll check for an existing login and if it does not exist, redirect the user to our login page. Here's the test code to check this:

```
func CheckLogin(w http.ResponseWriter, r *http.Request) bool {
  cookieSession, err := r.Cookie("sessionid")
  if err != nil {
    fmt.Println("no such cookie")
    Session.Create()
    fmt.Println(Session.ID)
    currTime := time.Now()
    Session.Expire = currTime.Local()
    Session.Expire.Add(time.Hour)

    return false
  } else {
    fmt.Println("found cookki")
    tmpSession := UserSession{UID: 0}
    loggedIn := Database.QueryRow("select user_id from sessions
      where session_id=?", cookieSession).Scan(&tmpSession.UID)
    if loggedIn != nil {
      return false
    } else {
      if tmpSession.UID == 0 {
        return false
      } else {

        return true
      }
    }
  }
}
```

This is a pretty standard test that looks for a cookie. If it doesn't exist, create a `Session` struct and save a cookie, and return false. Otherwise, return true if the cookie has been saved in the database already after a successful login.

This also relies on a new global variable, `Session`, which is of the new struct type `UserSession`:

```
var Database *sql.DB
var Routes *mux.Router
var Format string
type UserSession struct {
  ID              string
  GorillaSesssion *sessions.Session
  UID             int
  Expire          time.Time
}

var Session UserSession

func (us *UserSession) Create() {
  us.ID = Password.GenerateSessionID(32)
}
```

At the moment, there is an issue with our login page and this exists only to allow a third-party application to allow our users to authorize its use. We can fix this by simply changing our authentication page to set an `auth_type` variable based on whether we see `consumer_key` or `redirect_url` in the URL. In our `authorize.html` file, make the following change:

```
<input type="hidden" name="auth_type" value="{{.PageType}}" />
```

And in our `ApplicationAuthenticate()` handler, make the following change:

```
  if len(r.URL.Query()["consumer_key"]) > 0 {
    Authorize.ConsumerKey = r.URL.Query()["consumer_key"][0]
  } else {
    Authorize.ConsumerKey = ""
  }
  if len(r.URL.Query()["redirect"]) > 0 {
    Authorize.Redirect = r.URL.Query()["redirect"][0]
  } else {
    Authorize.Redirect = ""
  }

if Authorize.ConsumerKey == "" && Authorize.Redirect != "" {
  Authorize.PageType = "user"
} else {
  Authorize.PageType = "consumer"
}
```

This also requires a modification of our `Page{}` struct:

```
type Page struct {
  Title        string
  Authorize    bool
  Authenticate bool
```

```
    Application  string
    Action       string
    ConsumerKey  string
    Redirect     string
    PageType     string
  }
```

If we receive an authorization request from a `Page` type of user, we'll know that this is just a login attempt. If, instead, it comes from a client, we'll know it's another application attempting to make a request for our user.

In the former scenario, we'll utilize a redirect URL to pass the user back around after a successful authentication, assuming that the login is successful.

Gorilla offers a flash message; this is essentially a single-use session variable that will be removed as soon as it's read. You can probably see how this is valuable here. We'll set the flash message before redirecting it to our connecting service and then read that value on return, at which point it will be disposed of. Within our `ApplicationAuthorize()` handler function, we delineate between client and user logins. If the user logs in, we'll set a flash variable that can be retrieved.

```
if dbPassword == expectedPassword && allow == "1" &&
  authType == "client" {

  requestToken := Pseudoauth.GenerateToken()

  authorizeSQL := "INSERT INTO api_tokens set
    application_user_id=" + appUID + ", user_id=" + dbUID + ",
    api_token_key='" + requestToken + "' ON DUPLICATE KEY UPDATE
    user_id=user_id"

  q, connectErr := Database.Exec(authorizeSQL)
  if connectErr != nil {

      } else {
    fmt.Println(q)
  }
  redirectURL := CallbackURL + "?request_token=" + requestToken
  fmt.Println(redirectURL)
  http.Redirect(w, r, redirectURL, http.StatusAccepted)

}else if dbPassword == expectedPassword && authType == "user" {
  UserSession, _ = store.Get(r, "service-session")
      UserSession.AddFlash(dbUID)
  http.Redirect(w, r, redirect, http.StatusAccepted)
}
```

But this alone will not keep a persistent session, so we'll integrate this now. When a successful login happens in the `ApplicationAuthorize()` method, we'll save the session in our database and allow some persistent connection for our users.

Using data from other OAuth services

Having successfully connected to another service (or services, depending on which OAuth providers you've brought in), we can now cross-pollinate multiple services against ours.

For example, posting a status update within our social network may also warrant posting a status update on, say, Facebook.

To do this, let's first set up a table for statuses:

```
CREATE TABLE `users_status` (
  `users_status_id` INT NOT NULL AUTO_INCREMENT,
  `user_id` INT(10) UNSIGNED NOT NULL,
  `user_status_timestamp` INT(11) NOT NULL,
  `user_status_text` TEXT NOT NULL,
  PRIMARY KEY (`users_status_id`),
  CONSTRAINT `status_users` FOREIGN KEY (`user_id`) REFERENCES
  `users` (`user_id`) ON UPDATE NO ACTION ON DELETE NO ACTION
)
```

Our statuses will consist of the user's information, a timestamp, and the text of the status message. Nothing too fancy for now!

Next, we'll need to add API endpoints for creating, reading, updating, and deleting the statuses. So, in our `api.go` file, let's add these:

```
func Init() {
  Routes = mux.NewRouter()
  Routes.HandleFunc("/interface", APIInterface).Methods("GET",
    "POST", "PUT", "UPDATE")
  Routes.HandleFunc("/api/users", UserCreate).Methods("POST")
  Routes.HandleFunc("/api/users", UsersRetrieve).Methods("GET")
  Routes.HandleFunc("/api/users/{id:[0-9]+}",
    UsersUpdate).Methods("PUT")
  Routes.HandleFunc("/api/users", UsersInfo).Methods("OPTIONS")
  Routes.HandleFunc("/api/statuses",StatusCreate).Methods("POST")
  Routes.HandleFunc("/api/statuses",StatusRetrieve).Methods("GET")
  Routes.HandleFunc("/api/statuses/{id:[0-
    9]+}",StatusUpdate).Methods("PUT")
  Routes.HandleFunc("/api/statuses/{id:[0-
    9]+}",StatusDelete).Methods("DELETE")
  Routes.HandleFunc("/authorize",
    ApplicationAuthorize).Methods("POST")
  Routes.HandleFunc("/authorize",
    ApplicationAuthenticate).Methods("GET")
  Routes.HandleFunc("/authorize/{service:[a-z]+}",
    ServiceAuthorize).Methods("GET")
  Routes.HandleFunc("/connect/{service:[a-z]+}",
    ServiceConnect).Methods("GET")
  Routes.HandleFunc("/oauth/token",
    CheckCredentials).Methods("POST")
}
```

For now, we'll create some dummy handlers for the `PUT`/`Update` and `DELETE` methods:

```
func StatusDelete(w http.ResponseWriter, r *http.Request) {
  fmt.Fprintln(w, "Nothing to see here")
}

func StatusUpdate(w http.ResponseWriter, r *http.Request) {
  fmt.Fprintln(w, "Coming soon to an API near you!")
}
```

Remember, without these we'll be unable to test without receiving compiler errors in the meantime. In the following code, you'll find the `StatusCreate` method that allows us to make requests for users who have granted us a token. Since we already have one of the users, let's create a status:

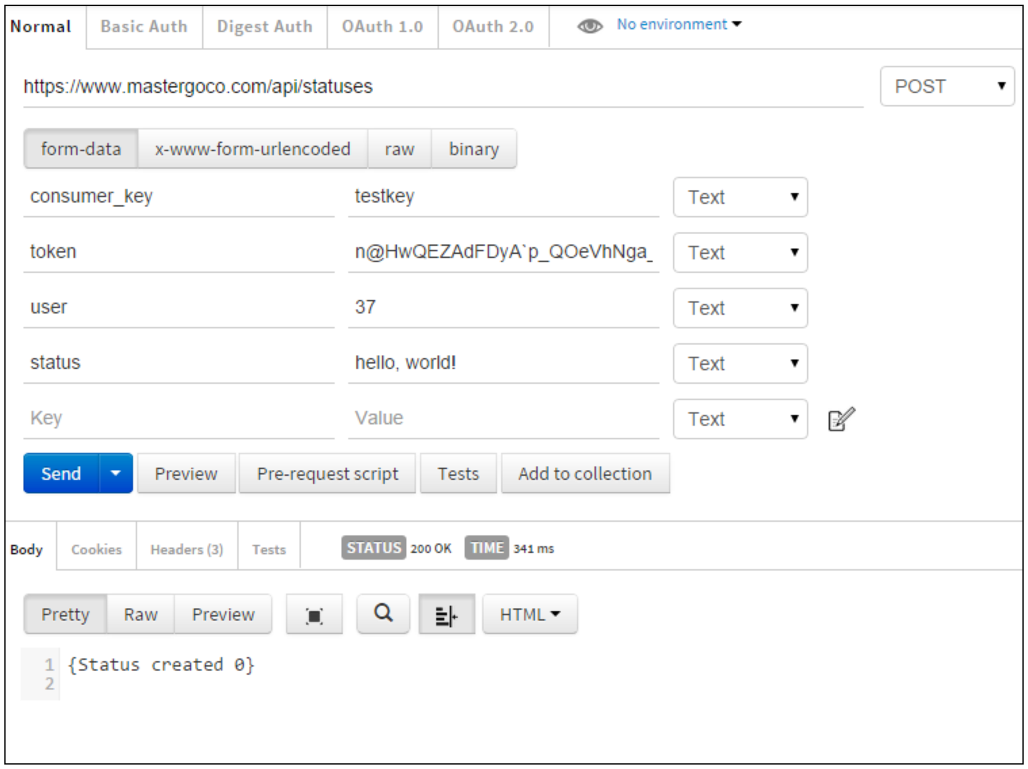
```
func StatusCreate(w http.ResponseWriter, r *http.Request) {

  Response := CreateResponse{}
  UserID := r.FormValue("user")
  Status := r.FormValue("status")
  Token := r.FormValue("token")
  ConsumerKey := r.FormValue("consumer_key")

  vUID := ValidateUserRequest(ConsumerKey,Token)
```

We'll use a test of the key and the token to get a valid user who is allowed to make these types of requests:

```
  if vUID != UserID {
    Response.Error = "Invalid user"
    http.Error(w, Response.Error, 401)
```

```
      } else  {
        _,inErr := Database.Exec("INSERT INTO users_status set
          user_status_text=?, user_id=?", Status, UserID)
        if inErr != nil {
          fmt.Println(inErr.Error())
          Response.Error = "Error creating status"
          http.Error(w, Response.Error, 500)
          fmt.Fprintln(w, Response)
        } else {
          Response.Error = "Status created"
          fmt.Fprintln(w, Response)
        }
      }


    }
```

If a user is confirmed as valid through the key and token, the status will be created.



https://www.mastergoco.com/api/statuses POST

consumer_key	testkey	Text
token	n@HwQEZAdFDyA`p_QOeVhNga_	Text
user	37	Text
status	hello, world!	Text
Key	Value	Text

STATUS 200 OK TIME 341 ms

```
{Status created 0}
```

With a knowledge of how OAuth works in general and by having an approximate, lower-barrier version baked into our API presently, we can start allowing external services to request access to our users' accounts to execute within our services on behalf of individual users.

We touched on this briefly in the last chapter, but let's do something usable with it.

We're going to allow another application from another domain make a request to our API that will create a status update for our user. If you use a separate HTML interface, either like the one that we used in earlier chapters or something else, you can avoid the cross-domain policy issues that you'll encounter when you return a cross-origin resource sharing header.

To do this, we can return the `Access-Control-Allow-Origin` header with the domains that we wish to allow to access to our API. If, for example, we want to allow `http://www.example.com` to access our API directly through the client side, we can create a slice of allowed domains at the top of our `api.go` file:

```
var PermittedDomains []string
```

Then, we can add these on the `Init()` function of our `api.go` file:

```
func Init(allowedDomains []string) {
  for _, domain := range allowedDomains {
    PermittedDomains = append(PermittedDomains,domain)
  }

Routes = mux.NewRouter()
Routes.HandleFunc("/interface", APIInterface).Methods("GET",
  "POST", "PUT", "UPDATE")
```

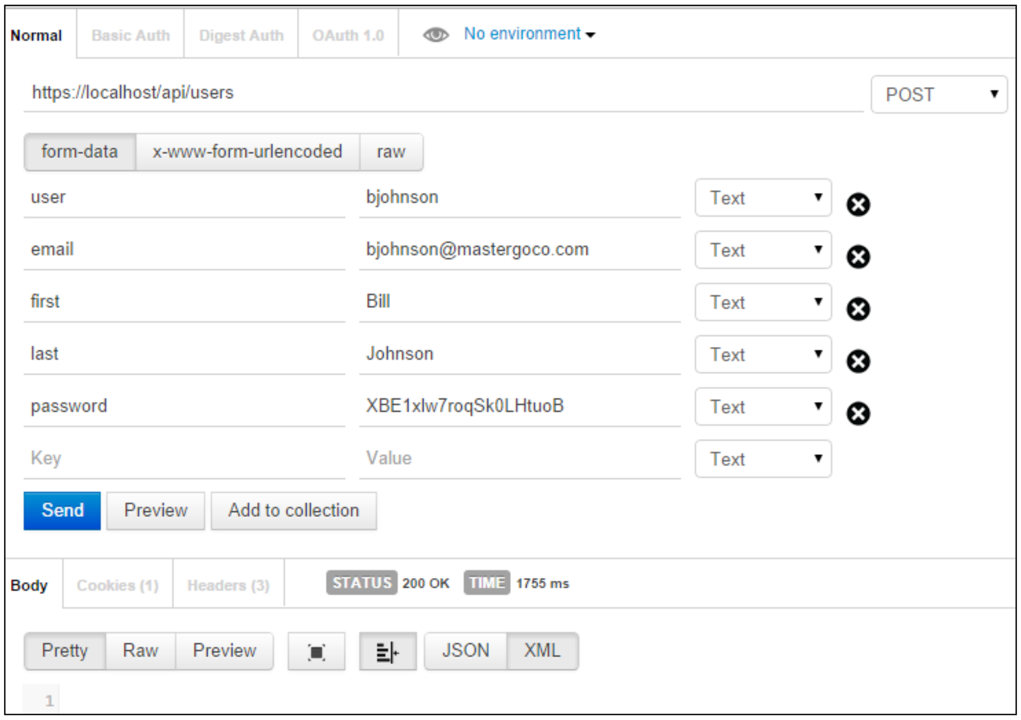
And then, we can call them from our present version of the API, currently at `v1`. So, in `v1.go`, we need to invoke the list of domains when calling `api.Init()`:

```
func API() {
  api.Init([]string{"http://www.example.com"})
```

And finally, within any handler where you wish to observe these domain rules, add a loop through those domains with the pertinent header set:

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

...
  for _,domain := range PermittedDomains {
    fmt.Println ("allowing",domain)
    w.Header().Set("Access-Control-Allow-Origin", domain)
  }
```

To start with, let's create a new user, Bill Johnson, through either of the aforementioned methods. In this case, we'll go back to Postman and just do a direct request to the API:



https://localhost/api/users POST

user	bjohnson	Text
email	bjohnson@mastergoco.com	Text
first	Bill	Text
last	Johnson	Text
password	XBE1xlw7roqSk0LHtuoB	Text
Key	Value	Text

STATUS 200 OK TIME 1755 ms

After the creation of the new user, we can follow our pseudo-OAuth process to allow Bill Johnson to give our application access and generate a status.

First, we pass the user to `/authorize` with our `consumer_key` value. On successful login and after agreeing to allow the application to access the user's data, we'll create a `token_key` value and pass it to the redirect URL.

With this key, we can make a status request programmatically as before by posting to the `/api/statuses` endpoint with our key, the user, and the status.

Connecting securely as a client in Go

You may encounter situations when instead of using an OAuth client; you're forced to make requests securely on your own. Normally, the `http` package in Go will ensure that the certificates included are valid and it will prevent you from testing.

```
package main

import
(
  "net/http"
  "fmt"
)

const (
  URL = "https://localhost/api/users"
)

func main() {

  _, err := http.Get(URL)
  if err != nil {

    fmt.Println(err.Error())
  }

}

type Client struct {
        // Transport specifies the mechanism by which individual
        // HTTP requests are made.
        // If nil, DefaultTransport is used.
        Transport RoundTripper
```

This allows us to inject a custom `Transport` client and thus override error handling; in the interactions with our (or any) API via the browser, this is not suggested beyond testing and it can introduce security issues with untrusted sources.

```
package main

import
(
  "crypto/tls"
  "net/http"
  "fmt"
)

const (
  URL = "https://localhost/api/users"
)

func main() {

  customTransport := &http.Transport{ TLSClientConfig:
    &tls.Config{InsecureSkipVerify: true} }
  customClient := &http.Client{ Transport: customTransport }
  response, err := customClient.Get(URL)
  if err != nil {
    fmt.Println(err.Error())
  } else {
    fmt.Println(response)
  }

}
```

We then get a valid response (with header, in struct):

```
&{200 OK 200 HTTP/1.1 1 1
  map[Link:[<http://localhost:8080/api/users?start= ; rel="next"]
  Pragma:[no
-cache] Date:[Tue, 16 Sep 2014 01:51:50 GMT] Content-Length:[256]
  Content-Type:[text/plain; charset=
utf-8] Cache-Control:[no-cache]] 0xc084006800 256 [] false map[]
  0xc084021dd0}
```

This is something that is best employed solely in testing, as the security of the connection can clearly be a dubious matter when a certificate is ignored.

Summary

We took our initial steps for third-party integration of our application in the last chapter. In this chapter, we looked a bit at the client side to see how we can incorporate a clean and simple flow.

We authenticated our users with other OAuth 2.0 services, which allowed us to share information from our social network with others. This is the basis of what makes social networks so developer friendly. Permitting other services to play with the data of our users and other users also creates a more immersive experience for users in general.

In the next chapter, we'll look at integrating Go with web servers and caching systems to construct a platform for a performant and scalable architecture.

We'll also push the functionality of our API in the process, which will allow more connections and functionality.

7 Working with Other Web Technologies

In our last chapter, we looked at how our web service can play nicely and integrate with other web services through APIs or OAuth integrations.

Continuing that train of thought, we'll take a pit stop as we develop the technology around our social network service to see how we can also integrate other technologies with it, independent of other services.

Very few applications run on a stack that's limited to just one language, one server type, or even one set of code. Often, there are multiple languages, operating systems, and designated purposes for multiple processes. You may have web servers running with Go on Ubuntu, which is a database server that runs PostgreSQL.

In this chapter, we'll look at the following topics:

- Serving our web traffic through a reverse proxy to leverage the more advanced features provided by mature HTTP products
- Connecting to NoSQL or key/value datastores, which we can utilize as our core data provider or with which we can do ancillary work such as caching
- Enabling sessions for our API and allowing clients and users to make requests without specifying credentials again
- Allowing users to connect with each other by way of friending or adding other users to their network

When we've finished all of this, you should have an idea about how to connect your web service with NoSQL and database solutions that are different to MySQL. We will utilize datastores later on to give us a performance boost in *Chapter 10, Maximizing Performance*.

You will hopefully also be familiar enough with some out-of-the-box solutions for handling APIs, be able to bring middleware into your web service, and be able to utilize message passing to communicate between dissonant or segregated systems.

Let's get started by looking at ways in which we can connect with other web servers to impose some additional functionality and failure mitigation into our own service that is presently served solely by Go's `net/http` package.

Serving Go through a reverse proxy

One of the most prominent features of Go's internal HTTP server might have also triggered an immediate, skeptical response: if it's so easy to start serving applications with Go, then is it fully featured as it relates to web serving?

This is an understandable question, particularly given Go's similarity to interpreted scripting languages. After all, Ruby on Rails, Python, NodeJS, and even PHP all come with out-of-the-box simple web servers. Rarely are these simple servers suggested as production-grade servers due to their limitations in feature set, security updates, and so on.

That being said, Go's `http` package is robust enough for many production-level projects; however, you may find not only some missing features but also some reliability by integrating Go with a reverse proxy that has a more mature web server.

A "reverse proxy" is a misnomer or at least a clunky way to illustrate an internal, incoming proxy that routes client requests opaquely through one system to another server, either within the same machine or network. In fact, it's often referred to simply as a gateway for this reason.

The potential advantages are myriad. These include being able to employ a well-known, well-supported, fully featured web server (versus only having the building blocks to build your own in Go), having a large community for support, and having a lot of pre-built, available plugins and tools.

Whether it's necessary or advantageous or has a good return on investment is a matter of preference and the situation you're in, but it can often help in logging and debugging web apps.

Using Go with Apache

Apache's web server is the elder statesman in web serving. First released in 1996, it quickly became a stalwart and as of 2009, it has served more than 100 million websites. It has remained in the most popular web server in the world since shortly after its inception, although some estimates have placed Nginx as the new number 1 (we will talk a little more about this in some time).

Putting Go behind Apache is super easy but there is one caveat; Apache, as it comes installed, is a blocking, nonconcurrent web server. This is different to Go, which delineates requests as goroutines or NodeJS or even Nginx. Some of these are bound to threads and some aren't. Go is obviously not bound, and this ultimately impacts how performant the servers can be.

To start, let's create a simple `hello world` web application in Go, which we'll call `proxy-me.go`:

```
package main

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

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

        fmt.Fprintln(w, "hello world")
}

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

There is nothing too complicated here. We listen on port 8080 and we have one very simple route, `/hello`, which just says `hello world`. To get Apache to serve this as a reverse proxy in pass-through, we edit our default server configuration as follows:

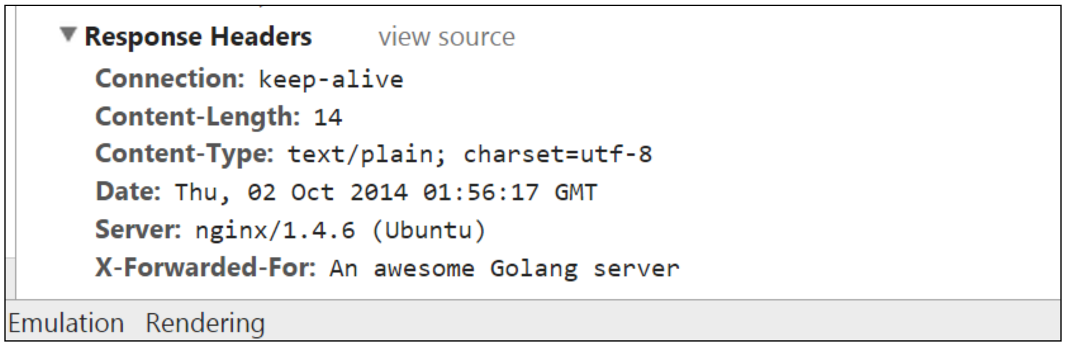
```
ProxyRequests Off
ProxyPreserveHost On

<VirtualHost *:80>

        ServerAdmin webmaster@localhost
```

```
            DocumentRoot /var/www/html

            ProxyPass /  http://localhost:8080/
            ProxyPassReverse /  http://localhost:8080/

            ErrorLog ${APACHE_LOG_DIR}/error.log
            CustomLog ${APACHE_LOG_DIR}/access.log combined


    </VirtualHost>
```

The default server configuration is generally stored at `/etc/apache2/sites-enabled/` for Linux and `[Drive]:/[apache install directory]/conf/` in Windows.

We can verify that we're seeing the page served by Apache rather than directly through Go by viewing the headers on a request to the `/hello` route.

When we do this, we'll see not only the Server as **Apache/2.4.7**, but also our custom header that was passed along. Typically, we'd use the **X-Forwarded-For** header for another purpose, but it's analogous enough to use as a demonstration, as shown in the following screenshot:



```
Connection: keep-alive
Content-Length: 14
Content-Type: text/plain; charset=utf-8
Date: Thu, 02 Oct 2014 01:56:17 GMT
Server: nginx/1.4.6 (Ubuntu)
X-Forwarded-For: An awesome Golang server
```

Go and NGINX as reverse proxies

While Apache is the old king of web serving, in recent years, it has been surpassed in popularity by Nginx at least by some measurements.

Nginx was initially written as an approach to the C10K problem—serving 10,000 concurrent connections. It's not an impossible task, but one that previously required expensive solutions to address it.

Since Apache, by default, spawns new threads and/or processes to handle new requests, it often struggles under heavy load.

On the other hand, Nginx was designed with an event model that is asynchronous and does not spawn new processes for each request. In many ways this makes it complementary to the way Go works with concurrency in the HTTP package.

Like Apache, the benefits of putting Nginx instead of Go are as follows:

- It has access and error logs. This is something that you'll need to build using the log package in Go. While it's easy enough to do, it's one fewer hassle.
- It has extraordinarily fast static file serving. In fact, Apache users often use Nginx exclusively to serve static files.
- It has SPDY support. SPDY is a new and somewhat experimental protocol that manipulates the HTTP protocol to introduce some speed and security features. There are some attempts to implement Go's HTTP and TLS at package libraries for SPDY, but nothing has been built natively into the net/HTTP package.
- It has built-in caching options and hooks for popular caching engines.
- It has the flexibility to delegate some requests to other processes.

We will discuss the usage of SPDY directly in both Nginx and within Go in *Chapter 10, Maximizing Performance*.

It's worth noting that asynchronous, nonblocking, and concurrent HTTP serving will almost always be bound to the constraints of technical externalities such as network latency, file and database blocking, and so on.

With that in mind, let's take a look at the setup for quickly putting Nginx instead of Go as a reverse proxy.

Nginx allows a pass through very simply by modifying the default configuration file. Nginx has no native support for Windows yet; so, in most *nix solutions, this file can be found by navigating to `/etc/nginx/sites-enabled`.

> Alternately, you can do a proxy globally by making the change within the `.conf` file available at `/etc/nginx/nginx.conf`.

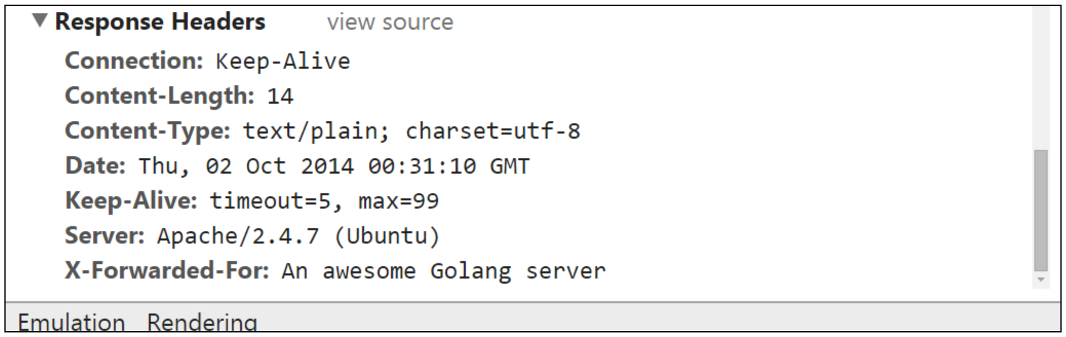
Let's look at a sample Nginx configuration operation that will let us proxy our server.

```
server {
        listen 80 default_server;
        listen [::]:80 default_server ipv6only=on;
```

```
            root /usr/share/nginx/html;
            index index.html index.htm;

            # Make site accessible from http://localhost/
            server_name localhost;

            location / {
                    proxy_set_header X-Real-IP $remote_addr;
                    proxy_set_header X-Forwarded-For $remote_addr;
                    proxy_set_header Host $host;
                    proxy_pass http://127.0.0.1:8080;
                    #       try_files $uri $uri/ =404;

            }
```

With this modification in place, you can start Nginx by running `/etc/init.d/nginx`, and then start the Go server with `go run proxy-me.go`.

If we hit our localhost implementation, we'll see something that looks a lot like our last request's headers but with Nginx instead of Apache as our proxy server:



```
Connection: Keep-Alive
Content-Length: 14
Content-Type: text/plain; charset=utf-8
Date: Thu, 02 Oct 2014 00:31:10 GMT
Keep-Alive: timeout=5, max=99
Server: Apache/2.4.7 (Ubuntu)
X-Forwarded-For: An awesome Golang server
```

Enabling sessions for the API

Mostly, we expose APIs for machines to use. In other words, we expect that some applications will be directly interfacing with our web service rather than the users.

However, this is not always the case. Sometimes, users interact with APIs using the browser, either directly or through a conduit like JavaScript with JSONP and/or AJAX requests.

In fact, the fundamentals of the aesthetics of Web 2.0 were rooted in providing users a seamless, desktop-like experience. This has come to fruition today and includes a lot of JavaScript MVC frameworks that handle presentation layers. We'll tackle this in our next chapter.

The term Web 2.0 has largely been supplanted and it is now usually referred to as a **Single Page App** or **SPA**. What was once a mixture of server-generated (or served) HTML pages with some pieces built or updated through XML and JavaScript has ceded to JavaScript frameworks that build entire client-side applications.

Almost all of these rely on an underlying API, which is generally accessible through stateless requests over HTTP/HTTPS, although some newer models use web sockets to enable real-time communication between the server and the presentation model. This is something that we'll look at in the next chapter as well.

Irrespective of the model, you cannot simply expose this API to the world without some authentication. If, for example, an API is accessible from a `/admin` request without authentication, it's probably also accessible from outside. You cannot rely on a user's information such as an HTTP referer.

> Grammarians may note the misspelling of referrer in the previous sentence. However, it's not a typo. In the initial HTTP request for comments proposal, the term was included without the double *r* in the spelling and it has largely stuck ever since.

However, relying on every OAuth request is overkill when it's a user who is making many requests per page. You could cache tokens in local storage or cookies, but browser support for the former is still limited and the latter limits the revocability of a token.

A traditional and simple solution for this is to allow sessions for authentication that are based on cookies. You may still want to leave an API open for access from outside a main application so that it can be authenticated via an API key or OAuth, but it should also enable users to interface with it directly from client-side tools to provide a clean SPA experience.

Sessions in a RESTful design

It's worth noting that because sessions typically enforce some sense of state, they are not inherently considered as a part of a RESTful design. However, it can also be argued that sessions can be used solely for authentication and not state. In other words, an authentication and a session cookie can be used elusively as a method for verifying identity.

Of course, you can also do this by passing a username and password along with every secure request. This is not an unsafe practice on its own, but it means that users will need to supply this information with every request, or the information will need to be stored locally. This is the problem that sessions that are stored in cookies attempt to solve.

As mentioned earlier, this will never apply to third-party applications, which for the most part need some sort of easily revokable key to work and rarely have a username and a password (although ours are tied to users, so they technically do).

The easiest way to do this is to allow a username and a password to go directly into the URL request, and you may see this sometimes. The risk here is that if a user shares the URL in full accidentally, the data will be compromised. In fact, this happens often with newer GitHub users, as it's possible to automatically push config files that contain GitHub passwords.

To reduce this risk, we should mandate that a username and a password be passed via a header field, although it should still be in cleartext. Assuming that a solid TSL (or SSL) option is in place, cleartext in the header of the request is not inherently a problem, but could be one if an application can at any point switch to (or be accessed by) unsecure protocols. This is a problem that time-restricted token systems attempt to address.

We can store session data anywhere. Our application presently uses MySQL, but session data will be read frequently. So, it's not ideal to encumber our database with information that has very little in terms of relational information.

Remember, we'll be storing an active user, their session's start time, the last update time (changed with every request), and perhaps where they are within the application. This last piece of information can be used in our application to tell users what their friends are currently doing within our social network.

With these conditions in mind, relying on our primary datastore is not an ideal solution. What we want is something more ephemeral, faster, and more concurrent that enables many successive requests without impacting our datastore.

One of the most popular solutions today for handling sessions in this regard is to yield relational databases to NoSQL solutions that include document and column stores or key-value datastores.

Using NoSQL in Go

Long ago, the world of data storage and retrieval was relegated almost exclusively to the realm of relational databases. In our application, we are using MySQL, largely because it's been a lingua franca for quick applications and SQL translates fairly easily across similar databases (Microsoft's SQL Server, PostgreSQL, Oracle, and so on).

In recent years, however, a big push has been made toward NoSQL. More accurately, the push has been towards data storage solutions that rely less on typical relational database structures and schemas and more on highly performant, key-value stores.

A key-value store is exactly what anyone who works with associative arrays, hashes, and maps (in Go) would expect, that is, some arbitrary data associated with a key. Many of these solutions are very fast because of the lack of indexed relationships, mitigation of locking, and a de-emphasis of consistency. In fact, many solutions guarantee no ACIDity out of the box (but some offer methods for employing it optionally).

ACID refers to the properties that developers expect in a database application. Some or all of these may be missing or may be optional parameters in any given NoSQL or key-value datastore solution. The term **ACID** can be elaborated as follows:

- **Atomicity**: This indicates that all parts of a transaction must succeed for any part to succeed
- **Consistency**: This refers to the database's state at the start of a transaction does not change before the completion of a transaction
- **Isolation**: This refers to the table or row locking mechanism that prevents access to data that is presently in the state of transaction
- **Durability**: This ensures that a successful transaction can and will survive a system or application failure

NoSQL solutions can be used for a lot of different things. They can be outright replacements for SQL servers. They can supplement data with some data that requires less consistency. They can work as quickly accessible, automatically expiring cache structures. We'll look at this in a moment.

If you choose to introduce a NoSQL solution into your application, be thoughtful about the potential impact this could bring to your application. For example, you can consider whether the potential tradeoff for ACID properties will be outweighed by performance boosts and horizontal scalability that a new solution provides.

While almost any SQL or traditional relational database solution out there has some integration with Go's `database/sql` package, this is not often the case with key-value stores that need some sort of package wrapper around them.

Now, we'll briefly look at a few of the most popular solutions for key-value stores and when we talk about caching in the next section, we'll come back and use NoSQL as a basic caching solution.

[NoSQL is, despite the recent resurgence, not a new concept. By definition, anything that eschews SQL or relational database concepts qualifies as NoSQL, and there have been dozens of such solutions since the 1960s. It probably bears to be mentioned that we're not spending any time on these solutions—like Ken Thompson's DBM or BerkeleyDB—but instead the more modern stories.]

Before we start exploring the various NoSQL solutions that we can use to handle sessions, let's enable them in our application by providing an alternative username/password authentication.

You may recall that back when we enabled third-party authentication proxies, we enabled sessions and stored them in our MySQL database in the `CheckLogin()` function. This function was only called in response to a `POST` request to the `ApplicationAuthorize` function. We'll open this up to more methods. First, let's create a new function called `CheckSession()`, if it doesn't exist, which will validate the cookie's session ID, and then validate against our session store if it does:

```
func CheckSession(w http.ResponseWriter, r *http.Request) bool {

}
```

You may recall that we also had a basic session struct and a method within `api.go`. We'll move these to sessions as well:

```
var Session UserSession
```

This command becomes the following:

```
var Session Sessions.UserSession
```

To create our session store, we'll make a new package called `sessions.go` within our API's subdirectory/sessions. This is the skeleton without any NoSQL specific methods:

```
package SessionManager

import
(
  "log"
  "time"
  "github.com/gorilla/sessions"
  Password "github.com/nkozyra/api/password"
)

var Session UserSession

type UserSession struct {
  ID              string
  GorillaSesssion *sessions.Session
  UID             int
  Expire          time.Time
}

func (us *UserSession) Create() {
  us.ID = Password.GenerateSessionID(32)
}

type SessionManager struct {

}

func GetSession() {

  log.Println("Getting session")
}

func SetSession() {

  log.Println("Setting session")
}
```

Let's look at a few simple NoSQL models that have strong third-party integrations with Go to examine how we can keep these sessions segregated and enable client-side access to our APIs in a way that they remain secure.

Memcached

We'll start with Memcached, specifically because it's not really a datastore like our other options. While it is still a key-value store in a sense, it's a general purpose caching system that maintains data exclusively in memory.

Developed by Brad Fitzpatrick for the once massively popular LiveJournal site, it was designed and intended to reduce the amount of direct access to the database, which is one of the most common bottlenecks in web development.

Memcached was originally written in Perl but has since been rewritten in C and it has reached a point of large-scale usage.

The pros and cons of this are already apparent—you get the speed of memory without the drag of disk access. This is obviously huge, but it precludes using data that should be consistent and fault tolerant without some redundancy process.

For this reason, it's ideal for caching pieces of the presentation layer and sessions. Sessions are already ephemeral in nature, and Memcached's built-in expiration feature allows you to set a maximum age for any single piece of data.

Perhaps Memcached's biggest advantage is its distributed nature. This allows multiple servers to share data in-memory values across a network.

It's worth noting here that Memcached operates as a first-in, first out system. Expiration is only necessary for programmatic purposes. In other words, there's no need to force a maximum age unless you need something to expire at a certain time.

In the `api.go` file, we'll check a cookie against our Memcached session proxy, or we'll create a session:

```
func CheckSession(w http.ResponseWriter, r *http.Request) bool {
  cookieSession, err := r.Cookie("sessionid")
  if err != nil {
    fmt.Println("Creating Cookie in Memcache")
    Session.Create()
    Session.Expire = time.Now().Local()
    Session.Expire.Add(time.Hour)
    Session.SetSession()
  } else {
    fmt.Println("Found cookie, checking against Memcache")
    ValidSession,err := Session.GetSession(cookieSession.Value)
    fmt.Println(ValidSession)
    if err != nil {
      return false
    } else {
      return true
    }

  }
  return true
}
```

And then, here is our `sessions.go` file:

```
package SessionManager

import
(
  "encoding/json"
  "errors"
  "time"
  "github.com/bradfitz/gomemcache/memcache"
  "github.com/gorilla/sessions"
  Password "github.com/nkozyra/api/password"

)

var Session UserSession

type UserSession struct {
  ID              string `json:"id"`
  GorillaSesssion *sessions.Session `json:"session"`
  SessionStore  *memcache.Client `json:"store"`
  UID             int `json:"uid"`
  Expire          time.Time `json:"expire"`
}

func (us *UserSession) Create() {
  us.SessionStore = memcache.New("127.0.0.1:11211")
  us.ID = Password.GenerateSessionID(32)
}

func (us *UserSession) GetSession(key string) (UserSession, error) {
  session,err := us.SessionStore.Get(us.ID)
  if err != nil {
    return UserSession{},errors.New("No such session")
  } else {
    var tempSession = UserSession{}
    err := json.Unmarshal(session.Value,tempSession)
    if err != nil {

    }
    return tempSession,nil
  }
}
```

`GetSession()` attempts to grab a session by key. If it exists in memory, it will pass its value to the referenced `UserSession` directly. Note that we make one minor change when we verify a session in the following code. We increase the cookie's expiry time by one hour. This is optional, but it allows a session to remain active if a user leaves one hour after their last action (and not their first one):

```
func (us *UserSession) SetSession() bool {
  jsonValue,_ := json.Marshal(us)
  us.SessionStore.Set(&memcache.Item{Key: us.ID, Value:
    []byte(jsonValue)})
  _,err := us.SessionStore.Get(us.ID)
  if err != nil {
      return false
  }
    Session.Expire = time.Now().Local()
    Session.Expire.Add(time.Hour)
    return true
}
```

Brad Fitzpatrick has joined the Go team at Google, so it should come as no surprise that he has written a Memcached implementation in Go. It should also come as no surprise that this is the implementation that we'll use for this example.

You can read more about this at `https://github.com/bradfitz/gomemcache` and install it using the `go get github.com/bradfitz/gomemcache/memcache` command.

MongoDB

MongoDB is one of the earlier big names in the latter day NoSQL solutions; it is a document store that relies on JSON-esque documents with open-ended schemas. Mongo's format is called BSON, for Binary JSON. So, as you can imagine, this opens up some different data types, namely BSON object and BSON array, which are both stored as binary data rather than string data.

You can read more about the Binary JSON format at `http://bsonspec.org/`.

As a superset, BSON wouldn't provide much in the way of a learning curve, and we won't be using binary data for session storage anyway, but there are places where storing data can be useful and thrifty. For example, BLOB data in SQL databases.

MongoDB has earned some detractors in recent years as newer, more feature-rich NoSQL solutions have come to the forefront, but you can still appreciate and utilize the simplicity it provides.

There are a couple of decent packages for MongoDB and Go out there, but the most mature is mgo.

- More information and download links for MongoDB are available at `http://www.mongodb.org/`
- mgo can be found at `https://labix.org/mgo` and it can installed using the `go get gopkg.in/mgo.v2` command

Mongo does not come with a built-in GUI, but there are a number of third-party interfaces and quite a few of them are HTTP-based. Here, I'll recommend Genghis (`http://genghisapp.com/`) that uses just a single file for either PHP or Ruby.

Let's look at how we can jump from authentication into session storage and retrieval using Mongo.

We'll supplant our previous example with another. Create a second file and another package subdirectory called `sessions2.go`.

In our `api.go` file, change the import call from `Sessions "github.com/nkozyra/api/sessions"` to `Sessions "github.com/nkozyra/api/sessionsmongo"`.

We'll also need to replace the `"github.com/bradfitz/gomemcache/memcache"` import with the mgo version, but since we're just modifying the storage platform, much of the rest remains the same:

```
package SessionManager

import
(
  "encoding/json"
  "errors"

  "log"
  "time"
  mgo "gopkg.in/mgo.v2"
  _ "gopkg.in/mgo.v2/bson"
  "github.com/gorilla/sessions"
```

```
    Password "github.com/nkozyra/api/password"

)

var Session UserSession

type UserSession struct {
  ID              string `bson:"_id"`
  GorillaSesssion *sessions.Session `bson:"session"`
  SessionStore  *mgo.Collection `bson:"store"`
  UID             int `bson:"uid"`
  Value          []byte `bson:"Valid"`
  Expire          time.Time `bson:"expire"`
}
```

The big change to our struct in this case is that we're setting our data to BSON instead of JSON in the string literal attribute. This is not actually critical and it will still work with the `json` attribute type.

```
func (us *UserSession) Create() {
  s, err := mgo.Dial("127.0.0.1:27017/sessions")
  defer s.Close()
  if err != nil {
    log.Println("Can't connect to MongoDB")
  } else {
    us.SessionStore = s.DB("sessions").C("sessions")
  }
  us.ID = Password.GenerateSessionID(32)
}
```

Our method of connection obviously changes, but we also need to work within a collection (that is analogous to a table in database nomenclature), so we connect to our database and then the collection that are both named `session`:

```
func (us *UserSession) GetSession(key string) (UserSession, error) {
  var session UserSession
  err := us.SessionStore.Find(us.ID).One(session)
  if err != nil {
    return UserSession{},errors.New("No such session")
  }
    var tempSession = UserSession{}
    err := json.Unmarshal(session.Value,tempSession)
    if err != nil {

    }
```

```
        return tempSession,nil


    }
```

`GetSession()` works in almost exactly the same way, aside from the datastore method being switched to `Find()`. The `mgo.One()` function assigns the value of a single document (row) to an interface.

```
    func (us *UserSession) SetSession() bool {
      jsonValue,_ := json.Marshal(us)
      err := us.SessionStore.Insert(UserSession{ID: us.ID, Value:
        []byte(jsonValue)})
      if err != nil {
          return false
      } else {
        return true
      }
    }
```

Enabling connections using a username and password

To permit users to enter a username and password for their own connections instead of relying on a token or leaving the API endpoint open, we can create a piece of middleware that can be called directly into any specific function.

In this case, we'll do several authentication passes. Here's an example in the `/api/users` GET function, which was previously open:

```
      authenticated := CheckToken(r.FormValue("access_token"))


      loggedIn := CheckLogin(w,r)
      if loggedIn == false {
        authenticated = false
        authenticatedByPassword := MiddlewareAuth(w,r)
        if authenticatedByPassword == true {
            authenticated = true
        }
      } else {
```

```
    authenticated = true
  }

  if authenticated == false {
    Response := CreateResponse{}
    _, httpCode, msg := ErrorMessages(401)
    Response.Error = msg
    Response.ErrorCode = httpCode
    http.Error(w, msg, httpCode)
   return
  }
```

You can see the passes that we make here. First, we check for a token and then we check for an existing session. If this doesn't exist, we check for a login `username` and `password` and validate them.

If all these three fail, then we return an unauthorized error.

Now, we already have the `MiddlewareAuth()` function in another part of the code in `ApplicationAuthorize()`, so let's move it:

```
func MiddlewareAuth(w http.ResponseWriter, r *http.Request) (bool,
  int) {

  username := r.FormValue("username")
  password := r.FormValue("password")

  var dbPassword string
  var dbSalt string
  var dbUID string

  uerr := Database.QueryRow("SELECT user_password, user_salt,
    user_id from users where user_nickname=?",
      username).Scan(&dbPassword, &dbSalt, &dbUID)
  if uerr != nil {

  }


  expectedPassword := Password.GenerateHash(dbSalt, password)

  if (dbPassword == expectedPassword) {
    return true, dbUID
  } else {
    return false, 0
  }
}
```

If users access the `/api/users` endpoint via a `GET` method, they will now need a `username` and `password` combination, an `access_token`, or a valid session in cookie data.

We also return the expected `user_id` on a valid authentication, which will otherwise return a value of 0.

Allowing our users to connect to each other

Let's take a step back into our application and add some functionality that's endemic to social networks—the ability to create connections such as friending. In most social networks, this grants read access to the data among those connected as friends.

Since we already have a valid view to see users, we can create some new routes to allow users to initiate connections.

First, let's add a few endpoints to our `Init()` function in the `api.go` file:

```
for _, domain := range allowedDomains {
  PermittedDomains = append(PermittedDomains, domain)
}
Routes = mux.NewRouter()
Routes.HandleFunc("/interface", APIInterface).Methods("GET",
  "POST", "PUT", "UPDATE")
Routes.HandleFunc("/api/users", UserCreate).Methods("POST")
Routes.HandleFunc("/api/users", UsersRetrieve).Methods("GET")
Routes.HandleFunc("/api/users/{id:[0-9]+}",
  UsersUpdate).Methods("PUT")
Routes.HandleFunc("/api/users", UsersInfo).Methods("OPTIONS")
Routes.HandleFunc("/api/statuses", StatusCreate).Methods("POST")
Routes.HandleFunc("/api/statuses", StatusRetrieve).Methods("GET")
Routes.HandleFunc("/api/statuses/{id:[0-9]+}",
  StatusUpdate).Methods("PUT")
Routes.HandleFunc("/api/statuses/{id:[0-9]+}",
  StatusDelete).Methods("DELETE")
Routes.HandleFunc("/api/connections",
  ConnectionsCreate).Methods("POST")
Routes.HandleFunc("/api/connections",
  ConnectionsDelete).Methods("DELETE")
Routes.HandleFunc("/api/connections",
  ConnectionsRetrieve).Methods("GET")
```

Note that we don't have a PUT request method here. Since our connections are friendships and binary, they won't be changed but they will be either created or deleted. For example, if we add a mechanism for blocking a user, we can create that as a separate connection type and allow changes to be made to it.

Let's set up a database table to handle these:

```
CREATE TABLE IF NOT EXISTS `users_relationships` (
  `users_relationship_id` int(13) NOT NULL,
  `from_user_id` int(10) NOT NULL,
  `to_user_id` int(10) NOT NULL,
  `users_relationship_type` varchar(10) NOT NULL,
  `users_relationship_timestamp` timestamp NOT NULL DEFAULT
    CURRENT_TIMESTAMP,
  `users_relationship_accepted` tinyint(1) NOT NULL DEFAULT '0',
  PRIMARY KEY (`users_relationship_id`),
  KEY `from_user_id` (`from_user_id`),
  KEY `to_user_id` (`to_user_id`),
  KEY `from_user_id_to_user_id` (`from_user_id`,`to_user_id`),
  KEY `from_user_id_to_user_id_users_relationship_type`
    (`from_user_id`,`to_user_id`,`users_relationship_type`)
)
```

With this in place, we can now duplicate the code that we used to ensure that the users are authenticated for our /api/connections POST method and allow them to initiate friend requests.

Let's look at the ConnectionsCreate() method:

```
func ConnectionsCreate(w http.ResponseWriter, r *http.Request) {
  log.Println("Starting retrieval")
  var uid int
  Response := CreateResponse{}
  authenticated := false
  accessToken := r.FormValue("access_token")
  if accessToken == "" || CheckToken(accessToken) == false {
    authenticated = false
  } else {
    authenticated = true
  }

  loggedIn := CheckLogin(w,r)
  if loggedIn == false {
    authenticated = false
    authenticatedByPassword,uid := MiddlewareAuth(w,r)
    if authenticatedByPassword == true {
        fmt.Println(uid)
        authenticated = true
    }
  } else {
    uid = Session.UID
    authenticated = true
  }

  if authenticated == false {

    _, httpCode, msg := ErrorMessages(401)
    Response.Error = msg
    Response.ErrorCode = httpCode
    http.Error(w, msg, httpCode)
    return
  }
```

This is the same code as our `/api/users` `GET` function. We'll come back to this after we look at the full example.

```
  toUID := r.FormValue("recipient")
  var count int
  Database.QueryRow("select count(*) as ucount from users where
    user_id=?",toUID).Scan(&count)

  if count < 1 {
    fmt.Println("No such user exists")
    _, httpCode, msg := ErrorMessages(410)
    Response.Error = msg
    Response.ErrorCode = httpCode
    http.Error(w, msg, httpCode)
    return
```

Here, we check for an existing user. If we are trying to connect to a user that doesn't exist, we return a 410: Gone HTTP error.

```
  } else {
    var connectionCount int
    Database.QueryRow("select count(*) as ccount from
      users_relationships where from_user_id=? and
      to_user_id=?",uid, toUID).Scan(&connectionCount)
    if connectionCount > 0 {
      fmt.Println("Relationship already exists")
      _, httpCode, msg := ErrorMessages(410)
```

```
                Response.Error = msg
          Response.ErrorCode = httpCode
          http.Error(w, msg, httpCode)
          return
```

Here, we check whether such a request has been initiated. If it has, then we also pass a Gone reference error. If neither of these error conditions is met, then we can create a relationship:

```
        } else {

          fmt.Println("Creating relationship")
          rightNow := time.Now().Unix()
          Response.Error = "success"
          Response.ErrorCode = 0
          _,err := Database.Exec("insert into users_relationships set
            from_user_id=?, to_user_id=?, users_relationship_type=?,
            users_relationship_timestamp=?",uid, toUID, "friend",
            rightNow)
          if err != nil {
            fmt.Println(err.Error())
          } else {
            output := SetFormat(Response)
            fmt.Fprintln(w, string(output))
          }
        }
      }
    }
```

With a successful call, we create a pending user relationship between the authenticated user and the intended one.

You may have noted the duplication of code in this function. This is something that's typically settled with middleware and Go has some options that are available to inject in the process. In the next chapter, we'll look at some frameworks and packages that can assist in this as well to build our own middleware.

Summary

We now have a featured social network that is available through web services with forced TLS, authentication from users, and it has the ability to interact with other users.

In this chapter, we also looked at offloading our session management to NoSQL databases and putting other web servers instead of Go to provide additional features and failover protections.

In the next chapter, we'll flesh out our social network even more as we try to interact with our API from the client side. With the foundation in place that allows this, we can then let users directly authenticate and interact with the API through a client-side interface without needing API tokens, while simultaneously retaining the ability to use third-party tokens.

We'll also peek at using Go with complementary frontend frameworks like Go and Meteor to provide a more responsive, app-like web interface.

8
Responsive Go for the Web

If you spend any time developing applications on the Web (or off it, for that matter) it won't be long before you find yourself facing the prospect of interacting with an API from within a website itself.

In this chapter, we'll bridge the gap between the client and the server by allowing the browser to work as a conduit for our web service directly via a few technologies that includes Google's own AngularJS.

Earlier in this book, we created a stopgap client-side interface for our API. This existed almost exclusively for the purpose of viewing the details and output of our web service through a simple interface.

However, it's important to keep in mind that it's not only machines that are processing APIs, but also client-side interfaces that are initiated directly by the users. For this reason, we're going to look at applying our own API in this format. We will keep it locked down by domain and enable RESTful and non-RESTful attributes that will allow a website to be responsive (not necessarily in the mobile sense) and operate exclusively via an API using HTML5 features.

In this chapter, we'll look at:

- Using client-side frameworks like jQuery and AngularJS to dovetail with our server-side endpoints
- Using server-side frameworks to create web interfaces
- Allowing our users to log in, view other users, create connections, and post messages via a web interface to our API
- Extending the functionality of our web service, and expanding it to allow direct access via an interface that we'll build in Go
- Employing HTML5 and several JavaScript frameworks to complement our server-side frameworks for Go

Creating a frontend interface

Before we get started, we'll need to address a couple of issues with the way browsers restrict information flow from the client to the server.

We'll also need to create an example site that works with our API. This should ideally be done on localhost on a different port or another machine because you will run into additional problems simply by using the `file://` access.

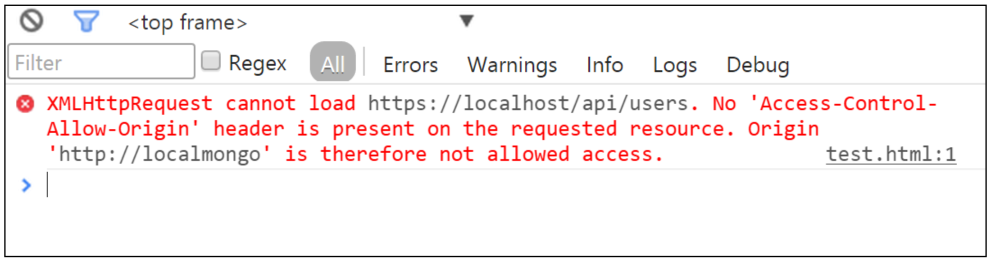
For the sake of building an API, it's entirely unnecessary to bundle an interface with the API, as we did for a simple demonstration earlier.

In fact, this may introduce cruft and confusion as a web service grows. In this example, we'll build our interface application separately and run it on port 444. You can choose any available port that you like, assuming that it doesn't interfere with our web service (443). Note that on many systems access to ports 1024 and below require `root/sudo`.

As is, if we attempt to run the interface on a different port than our secure web service, we'll run into cross-origin resource sharing issues. Make sure that any endpoint method that we expose for client-side and/or JavaScript consumption includes a header for `Access-Control-Allow-Origin`.

You can read more about the nature and mechanism of **Access-Control-Allow-Origin** at `https://developer.mozilla.org/en-US/docs/Web/HTTP/Access_control_CORS`.

You may be tempted to just use the `*` wildcard for this, but this will cause a lot of browser issues, particularly with the frontend frameworks that we'll be looking at. As an example, let's see what happens if we attempt to access the `/api/users` endpoint via `GET`:



```
XMLHttpRequest cannot load https://localhost/api/users. No 'Access-Control-
Allow-Origin' header is present on the requested resource. Origin
'http://localmongo' is therefore not allowed access.              test.html:1
```

The results can be unreliable and some frameworks reject the wildcard entirely. Using a wildcard also disables some key features that you may be interested in such as cookies.

You can see the following code that we used to attempt to access the web service to induce this error. The code is built in Angular, which we'll look at in more detail shortly:

```
<html>
<head>
  <title>CORS Test</title>
  <script src="//ajax.googleapis.com/ajax/libs
    /angularjs/1.2.26/angular.js"></script>
  <script src="//ajax.googleapis.com/ajax/libs
    /angularjs/1.2.26/angular-route.min.js"></script>
  <script>
    var app = angular.module
      ('testCORS', ['ngRoute']);
    app.controller('testWildcard', ['$scope', '$http',
      '$location', '$routeParams',
        function($scope,$http,$location,$routeParams) {
      $scope.messageFromAPI = '';
      $scope.users = [];
      $scope.requestAPI = function() {
        $http.get("https://localhost/api/users")
          .success(function(data,status,headers,config) {

          angular.forEach(data.users, function(val,key) {

          $scope.users.push({name: val.Name});

    })

  });
```

Here, we're making a `GET` request to our API endpoint. If this succeeds, we'll add users to our `$scope.users` array that is iterated though an AngularJS loop, which is shown in the following code. Without a domain origin allowance for our client, this will fail due to cross-origin policies in the browser:

```
      };

      $scope.requestAPI();

    }]);
  </script>
</head>
```

```
<body ng-app="testCORS">

  <div ng-controller="testWildcard">
    <h1 ng-model="messageFromAPI">Users</h1>
    <div ng-repeat="user in users">
      {{user.name}}
    </div>
```

This is the way AngularJS deals with loops by allowing you to specify a JavaScript array that is associated directly with a DOM-specific variable or a loop.

```
  </div>
</body>
</html>
```

In this example, we will get zero users due to the permissions' issue.

Luckily, we have previously addressed this issue in our application by introducing a very high-level configuration setting inside our `v1.go` file:

```
  api.Init([]string{"http://www.example.com","http:
    //www.mastergoco.com","http://localhost"})
```

You may recall that the `Init()` function accepts an array of allowed domains to which we can then set the `Access-Control-Allow-Origin` header:

```
func Init(allowedDomains []string) {
  for _, domain := range PermittedDomains {
    fmt.Println("allowing", domain)
    w.Header().Set("Access-Control-Allow-Origin", domain)
  }
```

As mentioned earlier, if we set a `*` wildcard domain, some browsers and libraries will disagree and the wildcard origin precludes the ability to neither set cookies nor honor SSL credentials. We can instead specify the domains more explicitly:

```
requestDomain := r.Header.Get("Origin")
if requestDomain != "" {
  w.Header.Set("Access-Control-Allow-Origin", requestDomain)
}
```

This permits you to retain the settings of the cookie and SSL certificate that are honoring the aspects of a non-wildcard access control header. It does open up some security issues that are related to cookies, so you must use this with caution.

If this loop is called within any function that can be accessible via a web interface, it will prevent the cross-origin issue.

Logging in

As before, we'll use Twitter's Bootstrap as a basic CSS framework, which allows us to quickly replicate a site structure that we might see anywhere online.

Remember that our earlier examples opened a login interface that simply passed a token to a third party for short-term use to allow the said application to perform actions on behalf of our users.

Since we're now attempting to allow our users to interface directly with our API (through a browser conduit), we can change the way that operates and allow sessions to serve as the authentication method.

Previously, we were posting login requests directly via JavaScript to the API itself, but since we're now using a full web interface, there's no reason to do that; we can post directly to the web interface itself. This primarily means eschewing the `onsubmit="return false"` or `onsubmit="userCreate();"` methods and just sending the form data to `/interface/login` instead:

```
func Init(allowedDomains []string) {
  for _, domain := range allowedDomains {
   PermittedDomains = append(PermittedDomains, domain)
  }
  Routes = mux.NewRouter()
  Routes.HandleFunc("/interface", APIInterface).Methods("GET", "POST",
"PUT", "UPDATE")
  Routes.HandleFunc("/interface/login",
    APIInterfaceLogin).Methods("GET")
  Routes.HandleFunc("/interface/login",
    APIInterfaceLoginProcess).Methods("POST")
  Routes.HandleFunc("/interface/register",
    APIInterfaceRegister).Methods("GET")
  Routes.HandleFunc("/interface/register",
    APIInterfaceRegisterProcess).Methods("POST")
```

This gives us enough to allow a web interface to create and login to our accounts utilizing existing code and still through the API.

Using client-side frameworks with Go

While we've spent the bulk of this book building a backend API, we've also been building a somewhat extensible, basic framework for the server-side.

When we need to access an API from the client side, we're bound by the limitations of HTML, CSS, and JavaScript. Alternatively, we can render pages on the server side as a consumer and we'll show that in this chapter as well.

However, most modern web applications operate on the client-side, frequently in the **single-page application** or **SPA**. This attempts to reduce the number of "hard" page requests that a user has to make, which makes a site appear less like an application and more like a collection of documents.

The primary way this is done is through asynchronous JavaScript data requests, which allow an SPA to *redraw* a page in response to user actions.

At first, there were two big drawbacks to this approach:

- First, the application state was not preserved, so if a user took an action and attempted to reload the page, the application would reset.
- Secondly, JavaScript-based applications fared very poorly in search engine optimization because a traditional web scraper would not render the JavaScript applications. It will only render the raw HTML applications.

But recently, some standardization and hacks have helped to mitigate these issues.

On state, SPAs have started utilizing a new feature in HTML5 that enables them to modify the address bar and/or history in browsers without requiring reloads, often by utilizing inline anchors. You can see this in an URL in Gmail or Twitter, which may look something like `https://mail.google.com/mail/u/0/#inbox/1494392317a0def6`.

This enables the user to share or bookmark a URL that is built through a JavaScript controller.

On SEO, this largely relegated SPAs to admin-type interfaces or areas where search engine accessibility was not a key factor. However, as search engines have begun parsing JavaScript, the window is open for widespread usage without negatively impacting the effects on SEO.

jQuery

If you do any frontend work or have viewed the source of any of the most popular websites on the planet, then you've encountered jQuery.

According to SimilarTech, jQuery is used by just about 67 million websites.

jQuery evolved as a method of standardizing an API among browsers where consistency was once an almost impossible task. Between the brazen self-determination of Microsoft's Internet Explorer and browsers that stuck to standards at variable levels, writing cross-browser code was once a very complicated matter. In fact, it was not uncommon to see this website best viewed with tags because there was no guarantee of functionality even with the latest versions of any given browser.

When jQuery took hold (following other similar frameworks such as Prototype, Moo Tools, and Dojo), the world of web development finally found a way to cover most of the available, modern web browsers with a single interface.

Consuming APIs with jQuery

Working with our API using jQuery couldn't be much simpler. When jQuery first started to come to fruition, the notion of AJAX was really taking hold. **AJAX** or **Asynchronous JavaScript** and **XML** were the first iteration towards a web technology that utilized the `XMLHttpRequest` object to get remote data and inject it into the DOM.

It's with some degree of irony that Microsoft, which is often considered as the greatest offender of web standards, laid the groundwork for `XMLHttpRequest` in the Microsoft Exchange Server that lead to AJAX.

Today, of course, XML is rarely a part of the puzzle, as most of what is consumed in these types of libraries is JSON. You can still use XML as source data, but it's likely that your responses will be more verbose than necessary.

Doing a simple `GET` request couldn't be easier as jQuery provides a simple shorthand function called `getJSON`, which you can use to get data from our API.

We'll now iterate through our users and create some HTML data to inject into an existing DOM element:

```
<script>

  $(document).ready(function() {
    $.getJSON('/api/users',function() {
        html = '';
      $(data.users).each(function() {
        html += '<div class="row">';
        html += '<div class="col-lg-3">'+ image + '</div>';
        html += '<div class="col-lg-9">
          <a href="/connect/'+this.ID+'/" >'
            + this.first + ' ' + this.last + '</a></div>';
        html += '</div>';
      });
    });
  });
</script>
```

The GET requests will only "get" us so far though. To be fully compliant with a RESTful web service, we need to be able to do the GET, POST, PUT, DELETE, and OPTIONS header requests. In fact, the last method will be important to allow requests across disparate domains.

As we mentioned earlier, getJSON is a shorthand function for the built-in ajax() method, which allows more specificity in your requests. For example, $.getJSON('/api/users') translates into the following code:

```
$.ajax({
  url: '/api/users',
  cache: false,
  type: 'GET', // or POST, PUT, DELETE
});
```

This means that we can technically handle all endpoints and methods in our API by setting the HTTP method directly.

While XMLHttpRequest accepts all of these headers, HTML forms (at least through HTML 4) only accept the GET and POST requests. Despite this, it's always a good idea to do some cross-browser testing if you're going to be using PUT, DELETE, OPTIONS, or TRACE requests in client-side JavaScript.

You can download and read the very comprehensive documentation that jQuery provides at http://jquery.com/. There are a few common CDNs that allow you to include the library directly and the most noteworthy is Google Hosted Libraries, which is as follows:

```
<script src="//ajax.googleapis.com/ajax/libs/jquery/2.1.1/jquery.min.js"></script>
```

The latest version of the library is available at https://developers.google.com/speed/libraries/devguide#jquery.

AngularJS

If we go beyond the basic toolset that jQuery provides, we'll start delving into legitimate, fully formed frameworks. In the last five years these have popped up like weeds. Many of these are traditional **Model-View-Controller** (**MVC**) systems, some are pure templating systems, and some frameworks work on both the client- and server-side, providing a unique push-style interface through websockets.

Like Go, Angular (or AngularJS) is a project maintained by Google and it aims to provide full-featured MVC on the client side. Note that over time, Angular has moved somewhat away from MVC as a design pattern and it has moved more towards MVVM or Model View ViewModel, which is a related pattern.

Angular goes far beyond the basic functionality that jQuery provides. In addition to general DOM manipulation, Angular provides true controllers as part of a larger app/application as well as for robust unit testing.

Among other things, Angular makes interfacing with APIs from the client side quick, easy, and pleasant. The framework provides a lot more MVC functionality that includes the ability to bring in separate templates from `.html/template` files.

> Actual push notifications are expected by many to become a standard feature in HTML5 as the specifications mature.
>
> The W3C had a working draft for the Push API at the time of writing this book. You can read more about it at `http://www.w3.org/TR/2014/WD-push-api-20141007/`.
>
> For now, workarounds include libraries such as Meteor (which will be discussed later) and others that utilize WebSockets in HTML5 to emulate real-time communication without being able to work within the confines of other browser-related restraints such as dormant processes in inactive tabs, and so on.

Consuming APIs with Angular

Enabling an Angular application to work with a REST API is, as with jQuery, built directly into the bones of the framework.

Compare this call to the `/api/users` endpoint that we just looked at:

```
$http.$get('/api/users'.
  success(function(data, status, headers, config) {
    html += '<div class="row">';
```

```
        html += '<div class="col-lg-3">'+ image + '</div>';
        html += '<div class="col-lg-9"><a href="/connect/'
          +this.ID+'/" >'+ this.first + ' ' + this.last +
            '</a></div>';
        html += '</div>';
      }).
      error(function(data, status, headers, config) {
        alert('error getting API!')
      });
```

Except syntax, Angular isn't all that different from jQuery; it also has a method that accepts a callback function or a promise as a second parameter. However, instead of setting the property for the method similar to jQuery, Angular provides short methods for most of the HTTP verbs.

This means that we can do our `PUT` or `DELETE` requests directly:

```
$http.$delete("/api/statuses/2").success(function(data,headers,config)
{
  console.log('Date of response:', headers('Date'))
  console.log(data.message)
}).error(function(data,headers,config) {
  console.log('Something went wrong!');
  console.log('Got this error:', headers('Status'));
});
```

Note that in the preceding example, we're reading header values. To make this work across domains, you need to also set a header that enables these headers to be shared for other domains:

```
Access-Control-Expose-Headers: [custom values]
```

Since domains are explicitly whitelisted with the `Access-Control-Allow-Origin` header, this controls the specific header keys that will be available to clients and not domains. In our case, we will set something for the `Last-Modified` and `Date` values.

You can read more about Angular and download it from `https://angularjs.org/`. You can also include the library directly from Google Hosted Libraries CDN, which is as follows:

```
<script src="//ajax.googleapis.com/
  ajax/libs/angularjs/1.2.26/angular.min.js"></script>
```

You can find the most recent version of the library at `https://developers.google.com/speed/libraries/devguide#angularjs`.

Setting up an API-consuming frontend

For the purpose of consuming an API, a frontend will be almost entirely free of internal logic. After all, the entirety of the application is called via HTML into a SPA, so we don't need much beyond a template or two.

Here is our `header.html` file, which contains the basic HTML code:

```
<html>
  <head>Social Network</title>

    <link href="//maxcdn.bootstrapcdn.com/bootstrap
      /3.3.0/css/bootstrap.min.css" rel="stylesheet">
    <script src="//ajax.googleapis.com/ajax/
      libs/jquery/2.1.1/jquery.min.js"></script>
    <script src="//maxcdn.bootstrapcdn.com/
      bootstrap/3.3.0/js/bootstrap.min.js"></script>
    <script src="//ajax.googleapis.com/
      ajax/libs/angularjs/1.2.26/angular.min.js"></script>
    <script src="//cdnjs.cloudflare.com/ajax/
      libs/react/0.12.0/react.min.js"></script>
    <script src="/js/application.js"></script>
  </head>

  <body ng-app="SocialNetwork">

    <div ng-view></div>
  </body>
```

The line with `application.js` is noteworthy because that's where all the logic will exist and utilize one of the frontend frameworks below.

The `ng-view` directive is no more than a placeholder that will be replaced with the values within a controller's routing. We'll look at that soon.

Note that we're calling AngularJS, jQuery, and React all in this header. These are options and you shouldn't necessarily import all of them. In all likelihood, this will cause conflicts. Instead, we'll explore how to handle our API with each of them.

As you might expect, our footer will be primarily closing tags:

```
</body>
</html>
```

We'll utilize Go's `http` template system to generate our basic template. The example here shows this:

```
<div ng-controller="webServiceInterface">
  <h1>{{Page.Title}}</h1>
  <div ng-model="webServiceError" style="display:none;"></div>
  <div id="webServiceBody" ng-model="body">
    <!-- nothing here, yet -->

  </div>
</div>
```

The heart of this template will not be hardcoded, but instead, it will be built by the JavaScript framework of choice.

Creating a client-side Angular application for a web service

As mentioned earlier, the `ng-view` directive within an `ng-app` element refers to dynamic content that is brought in according to the router that pairs URLs with controllers.

More accurately, it joins the pseudo-URL fragments (which we mentioned earlier) that are built on top of the `#` anchor tag. Let's first set up the application itself by using the following code snippet.

```
var SocialNetworkApp = angular.module('SocialNetwork',
  ['ngSanitize','ngRoute']);
SocialNetworkApp.config(function($routeProvider) {
  $routeProvider
  .when('/login',
    {
      controller: 'Authentication',
      templateUrl: '/views/auth.html'
    }
  ).when('/users',
    {
      controller: 'Users',
      templateUrl: '/views/users.html'
    }
  ).when('/statuses',
    {
      controller: 'Statuses',
      templateUrl: '/views/statuses.html'
    }
  );
});
```

Each one of these URLs, when they are accessed, tells Angular to pair a controller with a template and put them together within the `ng-view` element. This is what allows users to navigate across a site without doing hard page loads.

Here is `auth.html`, which is held in our `/views/` directory and allows us to log in and perform a user registration:

```
<div class="container">
  <div class="row">
    <div class="col-lg-5">
      <h2>Login</h2>
      <form>
        <input type="email" name="" class="form-control"
          placeholder="Email" ng-model="loginEmail" />
        <input type="password" name="" class="form-control"
          placeholder="Password" ng-model="loginPassword" />
        <input type="submit" value="Login" class="btn"
          ng-click="login()" />
      </form>
    </div>

    <div class="col-lg-2">
      <h3>- or -</h3>
    </div>

    <div class="col-lg-5">
      <h2>Register</h2>
      <form>
        <input type="email" name="" class="form-control"
          ng-model="registerEmail" placeholder="Email"
          ng-keyup="checkRegisteredEmail();" />
        <input type="text" name="" class="form-control"
          ng-model="registerFirst" placeholder="First Name" />
        <input type="text" name="" class="form-control"
          ng-model="registerLast" placeholder="Last Name" />
        <input type="password" name="" class="form-control"
          ng-model="registerPassword" placeholder="Password"
            ng-keyup="checkPassword();" />
        <input type="submit" value="Register" class="btn"
          ng-click="register()" />
      </form>
    </div>
  </div>
</div>
```

The JavaScript used to control this, as mentioned earlier, is merely a thin wrapper around our API. Here's the `Login()` process:

```
$scope.login = function() {
  postData = { email: $scope.loginEmail, password: $scope.
loginPassword };
  $http.$post('https://localhost/api/users', postData).
success(function(data) {

    $location.path('/users');

  }).error(function(data,headers,config) {
    alert ("Error: " + headers('Status'));
  });
};
```

And, here is the `Register()` process:

```
$scope.register = function() {
  postData = { user: $scope.registerUser, email:
    $scope.registerEmail, first: $scope.registerFirst, last:
    $scope.registerLast, password: $scope.registerPassword };
  $http.$post('https://localhost/api/users',
    postData).success(function(data) {

    $location.path('/users');

  }).error(function(data,headers,config) {
    alert ("Error: " + headers('Status'));
  });
};
  Routes.HandleFunc("/api/user",UserLogin).Methods("POST","GET")

  Routes.HandleFunc("/api/user",APIDescribe).Methods("OPTIONS")
```

We will like to make a note about the `OPTIONS` header here. This is an important part of how the CORS standard operates; essentially, requests are buffered with a preflight call using the `OPTIONS` verb that returns information on allowed domains, resources, and so on. In this case, we include a catchall called `APIDescribe` within `api.go`:

```
func APIDescribe(w http.ResponseWriter, r *http.Request) {
  w.Header().Set("Access-Control-Allow-Headers", "Origin,
    X-Requested-With, Content-Type, Accept")
  w.Header().Set("Access-Control-Allow-Origin", "*")
}
```

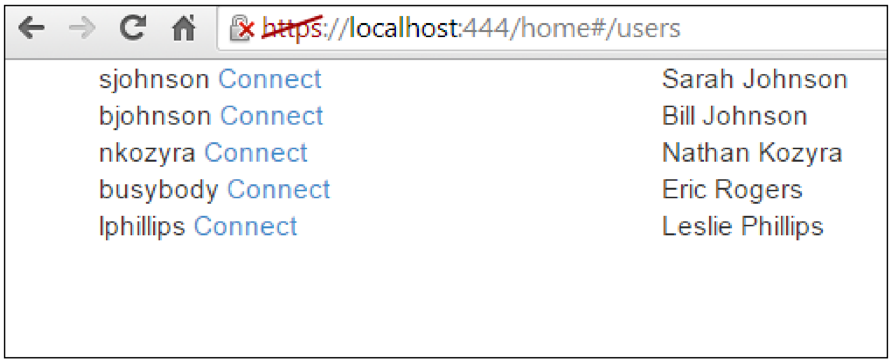
Viewing other users

Once we are logged in, we should be able to surface other users to an authenticated user to allow them to initiate a connection.

Here's how we can quickly view other users within our `users.html` Angular template:

```
<div class="container">
  <div class="row">
    <div ng-repeat="user in users">
      <div class="col-lg-3">{{user.Name}}
      <a ng-click="createConnection({{user.ID}});">
        Connect</a></div>
      <div class="col-lg-8">{{user.First}} {{user.Last}}</div>
    </div>

  </div>
</div>
```

We make a call to our `/api/users` endpoint, which returns a list of users who are logged in. You may recall that we put this behind the authentication wall in the last chapter.



sjohnson Connect	Sarah Johnson
bjohnson Connect	Bill Johnson
nkozyra Connect	Nathan Kozyra
busybody Connect	Eric Rogers
lphillips Connect	Leslie Phillips

There's not a lot of flair with this view. This is just a way to see people who you may be interested in connecting with or friending in our social application.

Rendering frameworks on the server side in Go

For the purposes of building pages, rendering frameworks is largely academic and it is similar to having prerendered pages from JavaScript and returning them.

For this reason, our total code for an API consumer is extraordinarily simple:

```
package main

import
(
  "github.com/gorilla/mux"
  "fmt"
  "net/http"
  "html/template"
)
var templates = template.Must(template.ParseGlob("templates/*"))
```

Here, we designate a directory to use for template access, which is the idiomatic template in this case. We don't use `views` because we'll use that for our Angular templates, and those chunks of HTML are called by `templateUrl`. Let's first define our SSL port and add a handler.

```
const SSLport = ":444"

func SocialNetwork(w http.ResponseWriter, r *http.Request) {
  fmt.Println("got a request")
  templates.ExecuteTemplate(w, "socialnetwork.html", nil)
}
```

That's it for our endpoint. Now, we're simply showing the HTML page. This can be done simply in any language and still interface with our web service easily:

```
func main() {

  Router := mux.NewRouter()
  Router.HandleFunc("/home", SocialNetwork).Methods("GET")
  Router.PathPrefix("/js/").Handler(http.StripPrefix("/js/",
    http.FileServer(http.Dir("js/"))))
  Router.PathPrefix("/views/").Handler(http.StripPrefix("/views/",
    http.FileServer(http.Dir("views/"))))
```

These last two lines allow serving files from a directory. Without these, we'll get error 404 when we attempt to call JavaScript or HTML include files. Let's add our SSLPort and certificates next.

```
  http.ListenAndServeTLS(SSLport, "cert.pem", "key.pem", Router)
  }
```

As mentioned earlier, the choice of the port and even HTTP or HTTPS is wholly optional, given that you allow the resulting domain to be in your list of permitted domains within `v1.go`.

Creating a status update

Our last example allows a user to view their latest status updates and create another one. It's slightly different because it calls upon two different API endpoints in a single view—the loop for the latest statuses and the ability to post, that is, to create a new one.

The `statuses.html` file looks a little like this:

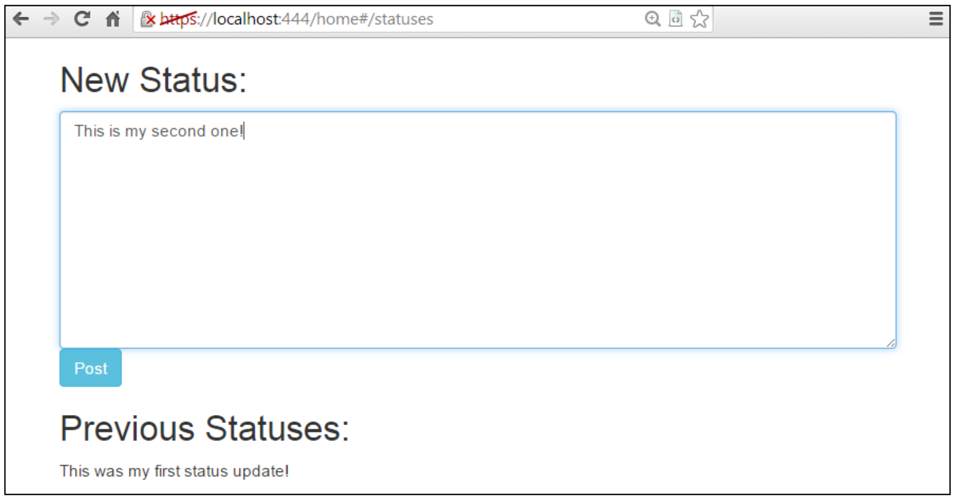
```
<div class="container">
  <div class="row">
    <div class="col-lg-12">
       <h2>New Status:</h2>
       <textarea class="form-control" rows="10"
         ng-mode="newStatus"></textarea>

       <a class="btn btn-info" ng-click="createStatus()">Post</a>
```

Here, we call on a `createStatus()` function within the controller to post to the `/api/statuses` endpoint. The rest of the code shown here shows a list of previous statuses through the ng-repeat directive:

```
    </div>
  </div>
  <div class="row">
    <div class="col-lg-12">
      <h2>Previous Statuses:</h2>
      <div ng-repeat="status in statuses">
        <div>{{status.text}}></div>
      </div>
  </div>
</div>
```

The preceding code simply displays the text as it is returned.

```
SocialNetworkApp.controller('Statuses',['$scope', '$http',
  '$location', '$routeParams', function
    ($scope,$http,$location,$routeParams) {

  $scope.statuses = [];
  $scope.newStatus;

  $scope.getStatuses = function() {
    $http.get('https://www.mastergoco.com
      /api/statuses').success(function(data) {

    });
  };

  $scope.createStatus = function() {
    $http({
      url: 'https://www.mastergoco.com/api/statuses',
      method: 'POST',
      data: JSON.stringify({ status: $scope.newStatus }),
            headers: {'Content-Type': 'application/json'}

  }).success(function(data) {
      $scope.statuses = [];
      $scope.getStatuses();
    });
  }

  $scope.getStatuses();

}]);
```



New Status:

This is my second one!

Previous Statuses:

This was my first status update!

Here, we can see a simple demonstration where previous status messages are displayed below a form for adding new status messages.

Summary

We've touched on the very basics of developing a simple web service interface in Go. Admittedly, this particular version is extremely limited and vulnerable to attack, but it shows the basic mechanisms that we can employ to produce usable, formalized output that can be ingested by other services.

Having superficially examined some of the big framework players for the Web as well as general purpose libraries such as jQuery, you have more than enough options to test your API against a web interface and create a single-page application.

At this point, you should have the basic tools at your disposal that are necessary to start refining this process and our application as a whole. We'll move forward and apply a fuller design to our API as we push forward, as two randomly chosen API endpoints will obviously not do much for us.

In the next chapter we'll dive in deeper with API planning and design, the nitty-gritty of RESTful services, and look at how we can separate our logic from our output. We'll briefly touch on some logic/view separation concepts and move toward more robust endpoints and methods in *Chapter 3, Routing and Bootstrapping*.

9
Deployment

When all is said and done, and you're ready to launch your web service or API, there are always considerations that need to be taken into account with regards to launching, from code repository, to staging, to live environments, to stop, start, and update policies.

Deploying compiled applications always carries a little more complexity than doing so with interpreted applications. Luckily, Go is designed to be a very modern, compiled language. By this, we mean that a great deal of thought has been devoted to the kinds of problems that traditionally plagued servers and services built in C or C++.

With this in mind, in this chapter, we're going to look at some tools and strategies that are available to us for painlessly deploying and updating our application with minimal downtime.

We're also going to examine some things that we can do to reduce the internal load of our web service, such as offloading image storage and messaging as part of our deployment strategy.

By the end of this chapter, you should have some Go-specific and general tips that will minimize some of the heartache that is endemic to deploying APIs and web services, particularly those that are frequently updated and require the least amount of downtime.

In this chapter, we'll look at:

- Application design and structure
- Deployment options and strategies for the cloud
- Utilization of messaging systems
- Decoupling image hosting from our API server and connecting it with a cloud-based CDN

Project structures

Though the design and infrastructure of your application is a matter of institutional and personal preference, the way you plan its architecture can have a very real impact on the approach that you use to deploy your application to the cloud or anywhere in production.

Let's quickly review the structure that we have for our application, keeping in mind that we won't need package objects unless we intend to produce our application for mass cross-platform usage:

```
bin/
  api # Our API binary

pkg/

src/
  github.com/
    nkozyra/
    api/
      /api/api.go
        /interface/interface.go
        /password/password.go
        /pseudoauth/pseudoauth.go
        /services/services.go
        /specification/specification.go
        /v1/v1.go
        /v2/v2.go
```

The structure of our application may be noteworthy depending on how we deploy it to the cloud.

If there's a conduit process before deployment that handles the build, dependency management, and push to the live servers, then this structure is irrelevant as the source and Go package dependencies can be eschewed in lieu of the binary.

However, in scenarios where the entire project is pushed to each application server or servers or NFS/file servers, the structure remains essential. In addition, as noted earlier, any place where cross-platform distribution is a consideration, the entire structure of the Go project should be preserved.

Even when this is not critical, if the build machine (or machines) are not exactly like the target machines, this impacts your process for building the binary, although it does not preclude solely dealing with that binary.

In an example GitHub repository, it might also require to obfuscate the nonbinary code if there is any open directory access, similar to our `interface.go` application.

Using process control to keep your API running

The methods for handling version control and development processes are beyond the scope of this book, but a fairly common issue with building and deploying compiled code for the Web is the process of installing and restarting the said processes.

Managing the way updates happen while minimizing or removing downtime is critical for live applications.

For scripting languages and languages that rely on an external web server to expose the application via the Web, this process is easy. The scripts either listen for changes and restart their internal web serving or they are interpreted when they are uncached and the changes work immediately.

This process becomes more complicated with long-running binaries, not only for updating and deploying our application but also for ensuring that our application is alive and does not require manual intervention if the service stops.

Luckily, there are a couple of easy ways to handle this. The first is just strict process management for automatic maintenance. The second is a Go-specific tool. Let's look at process managers first and how they work with a Go web service.

Using supervisor

There are a few big solutions here for *nix servers, from the absurdly simple to the more complex and granular. There's not a lot of difference in the way they operate, so we'll just briefly examine how we can manage our web service with one: Supervisor. Supervisor is readily available on most Linux distributions as well as on OS X, so it is a good example for testing locally.

Some other process managers of note are as follows:

- Upstart: `http://upstart.ubuntu.com/`
- Monit: `http://mmonit.com/monit/`
- Runit: `http://smarden.org/runit/`

The basic premise of these direct supervision init daemon monitoring process managers is to listen for running applications if there are no live attempts to restart them based on a set of configured rules.

It's worth pointing out here that these systems have no real distributed methods that allow you to manage multiple servers' processes in aggregate, so you'll generally have to yield to a load balancer and network monitoring for that type of feedback.

In the case of Supervisor, after installing it, all we need is a simple configuration file that can be typically located by navigating to `/etc/supervisor/conf.d/` on *nix distros. Here's an example of such a file for our application:

```
[program:socialnetwork]
command=/var/app/api
autostart=true
autorestart=true
stderr_logfile=/var/log/api.log
stdout_logfile=/var/log/api.log
```

While you can get more complex—for example, grouping multiple applications together to allow synchronous restarts that are useful for upgrades—that's all you should need to keep our long-running API going.

When it's time for updates, say from GIT to staging to live, a process that restarts the service can be triggered either manually or programmatically through a command such as the following one:

```
supervisorctl restart program:socialnetwork
```

This allows you to not only keep your application running, but it also imposes a full update process that pushes your code live and triggers a restart of the process. This ensures the least possible amount of downtime.

Using Manners for more graceful servers

While init replacement process managers work very well on their own, they do lack some control from within the application. For example, simply killing or restarting the web server would almost surely drop any active requests.

On its own, Manners lacks some of the listening control of a process such as **goagain**, which is a library that corrals your TCP listeners in goroutines and allows outside control for restarts via SIGUSR1/SIGUSR2 interprocess custom signals.

However, you can use the two together to create such a process. Alternatively, you can write the internal listener directly, as goagain may end up being a slight overkill for the aim of gracefully restarting a web server.

An example of using Manners as a drop-in replacement/wrapper around `net/http` will look something like this:

```
package main

import
(
  "github.com/braintree/manners"
  "net/http"
  "os"
  "os/signal"
)

var Server *GracefulServer

func SignalListener() {
  sC := make(chan os.signal, 1)
  signal.Notify(sC, syscall.SIGUSR1, syscall.SIGUSR2)
  s := <- sC
  Server.Shutdown <- true
}
```

After running within a goroutine and blocking with the channel that is listening for SIGUSR1 or SIGUSR2, we will pass our Boolean along the `Server.Shutdown` channel when such a signal is received.

```
func Init(allowedDomains []string) {
  for _, domain := range allowedDomains {
    PermittedDomains = append(PermittedDomains, domain)
  }
  Routes = mux.NewRouter()
  Routes.HandleFunc("/interface", APIInterface).Methods("GET",
    "POST", "PUT", "UPDATE")
  Routes.HandleFunc("/api/user",UserLogin).Methods("POST","GET")
  ...
}
```

This is just a rehash of our `Init()` function within `api.go`. This registers the Gorilla router that we'll need for our Manners wrapper.

```
func main() {

  go func() {
    SignalListener()
  }()
  Server = manners.NewServer()
  Server.ListenAndServe(HTTPport, Routes)
}
```

In the `main()` function, instead of just starting our `http.ListenAndServe()` function, we use the Manners server.

This will prevent open connections from breaking when we send a shutdown signal.

- You can install Manners with `go get github.com/braintree/manners`.
- You can read more about Manners at `https://github.com/braintree/manners`.
- You can install goagain with `go get github.com/rcrowley/goagain`.
- You can read more about goagain at `https://github.com/rcrowley/goagain`.

Deploying with Docker

In the last few years, there have been very few server-side products that have made as big a wave as Docker in the tech world.

Docker creates something akin to easily deployable, preconfigured virtual machines that have a much lower impact on the host machine than traditional VM software such as VirtualBox, VMWare, and the like.

It is able to do this with much less overall weight than VMs by utilizing Linux Containers, which allows the user space to be contained while retaining access to a lot of the operating system itself. This prevents each VM from needing to be a full image of the OS and the application for all practical purposes.

In order to be used in Go, this is generally a good fit, particularly if we create builds for multiple target processors and wish to easily deploy Docker containers for any or all of them. It is even better that the setup aspect is largely handled out of the box now, as Docker has created language stacks and included Go within them.

While at its core Docker is essentially just an abstraction of a typical Linux distribution image, using it can make upgrading and quickly provisioning a breeze, and it may even provide additional security benefits. The last point depends a bit on your application and its dependencies.

Docker operates with the use of very simple configuration files, and using a language stack, you can easily create a container that can be launched and has everything we need for our API.

Take a look at this Docker file example to see how we'd get all the necessary packages for our social networking web service:

```
FROM golang:1.3.1-onbuild

RUN go install github.com/go-sql-driver/mysql
RUN go install github.com/gorilla/mux
RUN go install github.com/gorilla/sessions
RUN go install github.com/nkozyra/api/password
RUN go install github.com/nkozyra/api/pseudoauth
RUN go install github.com/nkozyra/api/services
RUN go install github.com/nkozyra/api/specification
RUN go install github.com/nkozyra/api/api

EXPOSE 80 443
```

The file can then be built and run using simple commands:

```
docker build -t api .
docker run --name api-running api -it --rm
```

You can see how, at bare minimum, this would greatly speed up the Go update procedure across multiple instances (or containers in this case).

Complete Docker the base images are also available for the Google Cloud Platform. These are useful for quickly deploying the most recent version of Go if you use or would like to test Google Cloud.

Deploying in cloud environments

For those who remember the days of rooms full of physical single-purpose servers, devastating hardware faults, and insanely slow rebuild and backup times, the emergence of cloud hosting has in all likelihood been a godsend.

Nowadays, a full architecture can often be built from templates in short order, and autoscaling and monitoring are easier than ever. Now, there are a lot of players in the market too, from Google, Microsoft, and Amazon to smaller companies such as Linode and Digital Ocean that focus on simplicity, thrift, and ease of usage.

Each web service comes with its own feature set as well as disadvantages, but most share a very common workflow. For the sake of exploring additional functionality that may be available via APIs within Golang itself, we'll look at Amazon Web Services.

> Note that similar tools exist for other cloud platforms in Go. Even Microsoft's platform, Azure, has a client library that is written for Go.

Amazon Web Services

As with many of the aforementioned cloud services, deploying to Amazon Web Service or AWS is by and large no different than deploying it to any standard physical server's infrastructure.

There are a few differences with AWS though. The first is the breadth of services provided by it. Amazon does not strictly deal with only static virtual servers. It also deals with an array of supportive services such as DNS, e-mail, and SMS services (via their SNS service), long-term storage, and so on.

Despite all that has been said so far, note that many of the alternate cloud services provide similar functionality that may prove analogous to that provided with the following examples.

Using Go to interface directly with AWS

While some cloud services do offer some form of an API with their service, none are as robust as Amazon Web Services.

The AWS API provides direct access to every possible action in its environment, from adding instances, to provisioning IP addresses, to adding DNS entries and much more.

As you might expect, interfacing directly with this API can open up a lot of possibilities since it relates to automating the health of your application as well as managing updates and bug fixes.

To interface with AWS directly, we'll initiate our application with the `goamz` package:

```
package main
import (
    "launchpad.net/goamz/aws"
    "launchpad.net/goamz/ec2"
)
```

To grab the two dependencies to run this example, run the `go get launchpad.net/goamz/aws` command and the `go get launchpad.net/goamz/ec2` command.

You can find additional documentation about this at `http://godoc.org/launchpad.net/goamz`. The `goamz` package also includes a package for the Amazon S3 storage service and some additional experimental packages for Amazon's SNS service and Simple Database Service.

Starting a new instance based on an image is simple. Perhaps it is too simple if you're used to deploying it manually or through a controlled, automated, or autoscaled process.

```
AWSAuth, err := aws.EnvAuth()
if err != nil {
    fmt.Println(err.Error())
}
instance := ec2.New(AWSAuth, aws.USEast)
instanceOptions := ec2.RunInstances({
    ImageId:      "ami-9eaa1cf6",
    InstanceType: "t2.micro",
})
```

In this instance `ami-9eaa1cf6` refers to Ubuntu Server 14.04.

Having an interface to Amazon's API will be important in our next section where we'll take our image data and move it out of our relational database and into a CDN.

Handling binary data and CDNs

You may recall that way back in *Chapter 3, Routing and Bootstrapping*, we looked at how to store binary data, specifically image data, for our application in a database in the BLOB format.

At that time, we handled this in a very introductory way to simply get binary image data into some sort of a storage system.

Amazon S3 is part of the content distribution/delivery network aspect of AWS, and it operates on the notion of buckets as collections of data, with each bucket having its own set of access control rights. It should be noted that AWS also presents a true CDN called Cloudfront, but S3 can be used for this purpose as a storage service.

Let's first look at using the `goamz` package to list up to 100 items in a given bucket:

Replace ----------- in the code with your credentials.

```
package main

import
(
  "fmt"
    "launchpad.net/goamz/aws"
    "launchpad.net/goamz/s3"
)

func main() {
  Auth := aws.Auth { AccessKey: `-----------`, SecretKey: `-------
    ----`, }
  AWSConnection := s3.New(Auth, aws.USEast)

  Bucket := AWSConnection.Bucket("social-images")

    bucketList, err := Bucket.List("", "", "", 100)
    fmt.Println(AWSConnection,Bucket,bucketList,err)
    if err != nil {
        fmt.Println(err.Error())
    }
    for _, item := range bucketList.Contents {
        fmt.Println(item.Key)
    }
}
```

In our social network example, we're handling this as part of the `/api/user/:id:` endpoint.

```
 func UsersUpdate(w http.ResponseWriter, r *http.Request) {
  Response := UpdateResponse{}
  params := mux.Vars(r)
  uid := params["id"]
  email := r.FormValue("email")
  img, _, err := r.FormFile("user_image")
  if err != nil {
    fmt.Println("Image error:")
    fmt.Println(err.Error())
```

Return uploaded, instead we either check for the error and continue attempting to process the image or we move on. We'll show how to handle an empty value here in a bit:

```
}
imageData, ierr := ioutil.ReadAll(img)
if err != nil {
  fmt.Println("Error reading image:")
  fmt.Println(err.Error())
```

At this point we've attempted to read the image and extract the data—if we cannot, we print the response through `fmt.Println` or `log.Println` and skip the remaining steps, but do not panic as we can continue editing in other ways.

```
} else {
  mimeType, _, mimerr := mime.ParseMediaType(string(imageData))
  if mimerr != nil {
    fmt.Println("Error detecting mime:")
    fmt.Println(mimerr.Error())
  } else {
    Auth := aws.Auth { AccessKey: `-----------`, SecretKey: `---
      --------`, }
    AWSConnection := s3.New(Auth, aws.USEast)
    Bucket := AWSConnection.Bucket("social-images")
    berr := Bucket.Put("FILENAME-HERE", imageData, "", "READ")
    if berr != nil {
      fmt.Println("Error saving to bucket:")
      fmt.Println(berr.Error())
    }
  }
}
```

In *Chapter 3, Routing and Bootstrapping*, we took the data as it was uploaded in our form, converted it into a Base64-encoded string, and saved it in our database.

Since we're now going to save the image data directly, we can skip this final step. We can instead read anything from the `FormFile` function in our request and take the entire data and send it to our S3 bucket, as follows:

```
f, _, err := r.FormFile("image1")
if err != nil {
  fmt.Println(err.Error())
}
fileData,_ := ioutil.ReadAll(f)
```

It would make sense for us to ensure that we have a unique identifier for this image—one that avoids race conditions.

Checking for the existence of a file upload

The `FormFile()`function actually calls `ParseMultipartForm()` under the hood and returns default values for the file, the file header, and a standard error if nothing exists.

Sending e-mails with net/smtp

Decoupling our API and social network from ancillary tools is a good idea to create a sense of specificity in our system, reduce conflicts between these systems, and provide more appropriate system and maintenance rules for each.

It would be simple enough to equip our e-mail system with a socket client that allows the system to listen directly for messages from our API. In fact, this could be accomplished with just a few lines of code:

```
package main

import
(
  "encoding/json"
  "fmt"
  "net"
)

const
(
  port = ":9000"
)

type Message struct {
  Title string `json:"title"`
  Body string `json:"body"`
  To string `json:"recipient"`
  From string `json:"sender"`
}

func (m Message) Send() {

}
```

```
func main() {

  emailQueue,_ := net.Listen("tcp",port)
  for {
    conn, err := emailQueue.Accept()
    if err != nil {

    }
    var message []byte
    var NewEmail Message
    fmt.Fscan(conn,message)
    json.Unmarshal(message,NewEmail)
    NewEmail.Send()
  }

}
```

Let's look at the actual send function that will deliver our message from the registration process in our API to the e-mail server:

```
func (m Message) Send() {
  mailServer := "mail.example.com"
  mailServerQualified := mailServer + ":25"
  mailAuth := smtp.PlainAuth(
        "",
        "[email]",
        "[password]",
        mailServer,
      )
  recip := mail.Address("Nathan Kozyra","nkozyra@gmail.com")
  body := m.Body

  mailHeaders := make(map[string] string)
  mailHeaders["From"] = m.From
  mailHeaders["To"] = recip.toString()
  mailHeaders["Subject"] = m.Title
  mailHeaders["Content-Type"] = "text/plain; charset=\"utf-8\""
  mailHeaders["Content-Transfer-Encoding"] = "base64"
  fullEmailHeader := ""
  for k, v := range mailHeaders {
    fullEmailHeader += base64.
      StdEncoding.EncodeToString([]byte(body))
  }

  err := smtp.SendMail( mailServerQualified, mailAuth, m.From,
    m.To, []byte(fullEmailHeader))
  if err != nil {
    fmt.Println("could not send email")
    fmt.Println(err.Error())
  }
}
```

While this system will work well, as we can listen on TCP and receive messages that tell us what to send and to what address, it's not particularly fault tolerant on its own.

We can address this problem easily by employing a message queue system, which we'll look at next with RabbitMQ.

RabbitMQ with Go

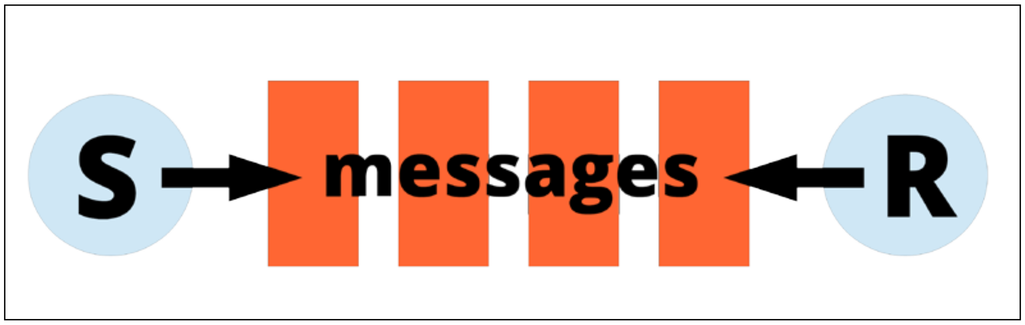
An aspect of web design that's specially relevant to APIs, but is a part of almost any web stack, is the idea of a message passing between servers and other systems.

It is commonly referred to as **Advanced Message Queuing Protocol** or **AMQP**. It can be an essential piece to an API/web service since it allows services that are otherwise separated to communicate with each other without utilizing yet another API.

By message passing, we're talking here about generic things that can or should be shared between dissonant systems getting moved to the relevant recipient whenever something important happens.

To draw another analogy, it's like a push notification on your phone. When a background application has something to announce to you, it generates the alert and passes it through a message passing system.

The following diagram is a basic representation of this system. The sender (S), in our case the API, will add messages to the stack that will then be retrieved by the receiver (R) or the e-mail sending process:



We believe that these processes are especially important to APIs because often, there's a institutional desire to segregate an API from the rest of the infrastructure. Although this is done to keep an API resource from impacting a live site or to allow two different applications to operate on the same data safely, it can also be used to allow one service to accept many requests while permitting a second service or system to process them as resources permit.

This also provides a very basic data glue for applications written in different programming languages.

In our web service, we can use an AMQP solution to tell our e-mail system to generate a welcome e-mail upon successful registration. This frees our core API from having to worry about doing that and it can instead focus on the core of our system.

There are a number of ways in which we can formalize the requests between system A and system B, but the easiest way to demonstrate a simple e-mail message is by setting a standard message and title and passing it in JSON:

```
type EmailMessage struct {
  Recipient string `json:"to"`
  Sender string `json:"from"`
  Title string `json:"title"`
  Body string `json:"body"`
  SendTime time.Time `json:"sendtime"`
  ContentType string `json:"content-type"`
}
```

Receiving e-mails in this way instead of via an open TCP connection enables us to protect the integrity of the messages. In our previous example, any message that would be lost due to failure, crash, or shutdown would be lost forever.

Message queues, on the other hand, operate like mailboxes with levels of configurable durability that allow us to dictate how messages should be saved, when they expire, and what processes or users should have access to them.

In this case, we use a literal message that is delivered as part of a package that will be ingested by our mail service through the queue. In the case of a catastrophic failure, the message will still be there for our SMTP server to process.

Another important feature is its ability to send a "receipt" to the message initiator. In this case, an e-mail system would tell the API or web service that the e-mail message was successfully taken from the queue by the e-mail process.

This is something that is not inconsequential to replicate within our simple TCP process. The number of fail-safes and contingencies that we'd have to build in would make it a very heavy, standalone product.

Luckily, integrating a message queue is pretty simple within Go:

```
func Listen() {

  qConn, err := amqp.Dial("amqp://user:pass@domain:port/")
  if err != nil {
    log.Fatal(err)
  }
```

This is just our connection to the RabbitMQ server. If any error with the connection is detected, we will stop the process.

```
  qC,err := qConn.Channel()
  if err != nil {
    log.Fatal(err)
  }

  queue, err := qC.QueueDeclare("messages", false, false, false,
    false, nil)
  if err != nil {
    log.Fatal(err)
  }
```

The name of the queue here is somewhat arbitrary like a memcache key or a database name. The key is to make sure that both the sending and receiving mechanisms search for the same queue name:

```
  messages, err := qC.Consume( queue.Name, "", true, false, false,
    false, nil)
  waitChan := make(chan int)
```

```
    go func() {
      for m := range messages {
        var tmpM Message
        json.Unmarshal(d.Body,tmpM)
        log.Println(tmpM.Title,"message received")
        tmpM.Send()
      }
```

In our loop here, we listen for messages and invoke the `Send()` method when we receive one. In this case, we're passing JSON that is then unmarshalled into a `Message` struct, but this format is entirely up to you:

```
    }()

    <- waitChan

  }
```

And, in our `main()` function, we need to make sure that we replace our infinite TCP listener with the `Listen()` function that calls the AMQP listener:

```
  func main() {

    Listen()
```

Now, we have the ability to take messages (in the e-mail sense) from the queue of messages (in the message queue sense), which means that we'd simply need to include this functionality in our web service as well.

In the example usage that we discussed, a newly registered user would receive an e-mail that prompts for the activation of the account. This is generally done to prevent sign ups with fake e-mail addresses. This is not an airtight security mechanism by any means, but it ensures that our application can communicate with a person who ostensibly has access to a real e-mail address.

Sending to the queue is also easy.

Given that we're sharing credentials across two separate applications, it makes sense to formalize this into a separate package:

```
  package emailQueue

  import
  (
    "fmt"
    "log"
    "github.com/streadway/amqp"
  )

  const
  (
    QueueCredentials = "amqp://user:pass@host:port/"
    QueueName = "email"
  )

  func Listen() {

  }

  func Send(Recipient string, EmailSubject string, EmailBody string)
    {

  }
```

In this way, both our API and our listener can import our `emailQueue` package and share these credentials. In our `api.go` file, add the following code:

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

    ...

    q, err := Database.Exec("INSERT INTO users set user_nickname=?,
      user_first=?, user_last=?, user_email=?, user_password=?,
        user_salt=?",NewUser.Name,NewUser.First,NewUser.Last,
          NewUser.Email,hash,salt)
    if err != nil {
      errorMessage, errorCode := dbErrorParse(err.Error())
      fmt.Println(errorMessage)
      error, httpCode, msg := ErrorMessages(errorCode)
      Response.Error = msg
          Response.ErrorCode = error
      http.Error(w, "Conflict", httpCode)
    } else {

      emailQueue.Send(NewUser.Email,"Welcome to the Social
        Network","Thanks for joining the Social Network!
          Your personal data will help us become billionaires!")

    }
```

And in our `e-mail.go` process:

```
emailQueue.Listen()
```

AMQP is a more generalized message passing interface with RabbitMQ extensions. You can read more about it at `https://github.com/streadway/amqp`.

More information on Grab Rabbit Hole is available at `https://github.com/michaelklishin/rabbit-hole` or can be downloaded using the `go get github.com/michaelklishin/rabbit-hole` command.

Summary

By separating the logic of our API from our hosted environment and ancillary, supportive services, we can reduce the opportunity for feature creep and crashes due to non-essential features.

In this chapter, we moved image hosting out of our database and into the cloud and stored raw image data and the resulting references to S3, a service that is often used as a CDN. We then used RabbitMQ to demonstrate how message passing can be utilized in deployment.

At this point, you should have a grasp of offloading these services as well as a better understanding of the available strategies for deployment, updates, and graceful restarts.

In our next chapter, we'll begin to round out the final, necessary requirements of our social network and in doing so, explore some ways to increase the speed, reliability, and overall performance of our web service.

We'll also introduce a secondary service that allows us to chat within our social network from the SPA interface as well as expand our image-to-CDN workflow to allow users to create galleries. We'll look at ways in which we can maximize image presentation and acquisition through both the interface and the API directly.

10
Maximizing Performance

With concepts relating to deploying and launching our application behind us, we'll lock in high-performance tactics within Go and related third-party packages in this chapter.

As your web service or API grows, performance issues may come to the fore. One sign of a successful web service is a need for more and more horsepower behind your stack; however, reducing this need through programmatic best practices is an even better approach than simply providing more processing power to your application.

In this chapter, we'll look at:

- Introducing middleware to reduce redundancy in our code and pave the way for some performance features
- Designing caching strategies to keep content fresh and provide it as quickly as possible
- Working with disk-based caching
- Working with memory caching
- Rate-limiting our API through middleware
- Google's SPDY protocol initiative

By the end of this chapter, you should know how to build your own middleware into your social network (or any other web service) to bring in additional features that introduce performance speedups.

Using middleware to reduce cruft

When working with the Web in Go, the built-in approaches to routing and using handlers don't always lend themselves to very clean methods for middleware out of the box.

For example, although we have a very simple `UsersRetrieve()` method, if we want to prevent consumers from getting to that point or run something before it, we will need to include these calls or parameters multiple times in our code:

```
func UsersRetrieve(w http.ResponseWriter, r *http.Request) {
  CheckRateLimit()
```

And an other call is:

```
func UsersUpdate( w http.ResponseWriter, r *http.Request) {
  CheckRateLimit()
  CheckAuthentication()
}
```

Middleware allows us to more cleanly direct the internal patterns of our application, as we can apply checks against rate limits and authentication as given in the preceding code. We can also bypass calls if we have some external signal that tells us that the application should be temporarily offline without stopping the application completely.

Considering the possibilities, let's think about useful ways in which we can utilize middleware in our application.

The best way to approach this is to find places where we've inserted a lot of needless code through duplication. An easy place to start is our authentication steps that exist as a potential block in a lot of sections of code in our `api.go` file. Refer to the following:

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

  CheckLogin(w,r)
```

We call the `CheckLogin()` function multiple times throughout the application, so we can offload this to middleware to reduce the cruft and duplicate code throughout.

Another method is the access control header setting that allows or denies requests based on the permitted domains. We use this for a few things, particularly for our server-side requests that are bound to CORS rules:

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

  w.Header().Set("Access-Control-Allow-Origin", "*")
```

```
    for _, domain := range PermittedDomains {
      fmt.Println("allowing", domain)
      w.Header().Set("Access-Control-Allow-Origin", domain)
    }
```

This too can be handled by middleware as it doesn't require any customization that is based on request type. On any request in which we wish to set the permitted domains, we can move this code into middleware.

Overall, this represents good code design, but it can sometimes be tricky without custom middleware handlers.

One popular approach to middleware is chaining, which works something like this:

```
firstFunction().then(nextFunction()).then(thirdFunction())
```

This is extremely common within the world of Node.js, where the `next()`, `then()`, and `use()` functions pepper the code liberally. And it's possible to do this within Go as well.

There are two primary approaches to this. The first is by wrapping handlers within handlers. This is generally considered to be ugly and is not preferred.

Dealing with wrapped handler functions that return to their parent can be a nightmare to parse.

So, let's instead look at the second approach: chaining. There are a number of frameworks that include middleware chaining, but introducing a heavy framework simply for the purpose of middleware chaining is unnecessary. Let's look at how we can do this directly within a Go server:

```
package main

import
(
  "fmt"
  "net/http"
)

func PrimaryHandler(w http.ResponseWriter, r *http.Request) {
  fmt.Fprintln(w, "I am the final response")
}

func MiddlewareHandler(h http.HandlerFunc) http.HandlerFunc {
  fmt.Println("I am middleware")
  return func(w http.ResponseWriter, r *http.Request) {
    h.ServeHTTP(w, r)
  }
}

func middleware(ph http.HandlerFunc, middleHandlers
  ..func(http.HandlerFunc) (http.HandlerFunc) ) http.HandlerFunc {
  var next http.HandlerFunc = ph
  for _, mw := range middleHandlers {
    next = mw(ph)
  }
  return next
}

func main() {
  http.HandleFunc("/middleware", middleware
    (PrimaryHandler,MiddlewareHandler))
  http.ListenAndServe(":9000",nil)
}
```

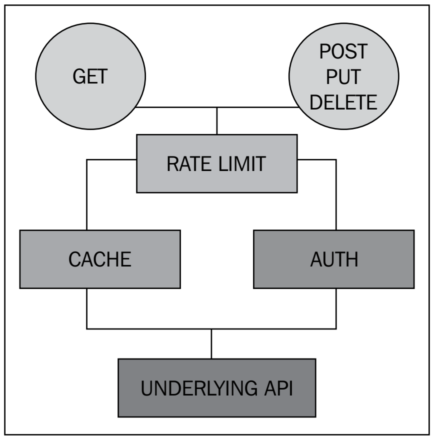
As mentioned earlier, there are a couple of places in our code and most server-based applications where middleware would be very helpful. Later in this chapter, we'll look at moving our authentication model(s) into middleware to reduce the amount of repetitious calls that we make within our handlers.

However, for performance's sake, another function for a middleware of this kind can be used as a blocking mechanism for cache lookups. If we want to bypass potential bottlenecks in our `GET` requests, we can put a caching layer between the request and the response.

We're using a relational database, which is one of the most common sources of web-based bottlenecks; so, in situations where stale or infrequently changing content is acceptable, placing the resulting queries behind such a barrier can drastically improve our API's overall performance.

Given that we have two primary types of requests that can benefit from middleware in different ways, we should spec how we'll approach the middleware strategy for various requests.

The following diagram is a model of how we can architect middleware. It can serve as a basic guide for where to implement specific middleware handlers for certain types of API calls:



All requests should be subject to some degree of rate-limiting, even if certain requests have much higher limits than others. So, the `GET`, `PUT`, `POST`, and `DELETE` requests will run through at least one piece of middleware on every request.

Any requests with other verbs (for example, `OPTIONS`) should bypass this.

The `GET` requests should be subject to caching, which we also described as making the data they return amenable to some degree of staleness.

On the other hand, PUT, POST, and DELETE requests obviously cannot be cached, as this will either force our responses to be inaccurate or it will lead to duplicate attempts to create or remove data.

Let's start with the `GET` requests and look at two related ways in which we can bypass a bottleneck when it is possible to deliver server-cached results instead of hitting our relational database.

Caching requests

There are, of course, more than one or two methods for inducing caching across the lifetime of any given request. We'll explore a few of them in this section to introduce the highest level of nonredundant caching.

There is client-side caching at a script or a browser level that is ostensibly bound to the rules that are sent to it from the server side. By this, we mean yielding to HTTP response headers such as `Cache-Control`, `Expires`, `If-None-Match`, `If-Modified-Since`, and so on.

These are the simplest forms of cache control that you can enforce, and they are also pretty important as part of a RESTful design. However, they're also a bit brittle as they do not allow any enforcement of those directives and clients that can readily dismiss them.

Next, there is proxy-based caching—typically third-party applications that either serve a cached version of any given request or pass-through to the originating server application. We looked at a precursor to this when we talked about using Apache or Nginx in front of our API.

Finally, there is server-level caching at the application level. This is typically done in lieu of proxy caching because the two tend to operate on the same rule sets. In most cases, appealing to a standalone proxy cache is the wisest option, but there are times when those solutions are unable to accommodate specific edge cases.

There's also some merit in designing these from scratch to better understand caching strategies for your proxy cache. Let's briefly look at building server-side application caching for our social network in both disk-based and memory-based ways, and see how we can utilize this experience to better define caching rules at the proxy level.

Simple disk-based caching

Not all that long ago, the way most developers handled caching requests was typically through disk-based caching at the application level.

In this approach, some parameters were set around the caching mechanisms and qualifiers of any given request. Then, the results of the request were saved to a string and then to a lock file. Finally, the lock file was renamed. The process was pretty steady although it was archaic and worked well enough to be reliable.

There were a number of downsides that were somewhat insurmountable at the time in the early days of the Web.

Note that disks, particularly mechanical magnetic disks, have been notoriously and comparatively slow for storage and access, and they are bound to cause a lot of issues with filesystems and OS operations with regard to lookups, finds, and sorting.

Distributed systems also pose an obvious challenge where a shared cache is necessary to ensure consistency across balanced requests. If server A updates its local cache and the next request returns a cache hit from server B, you can see varying results depending on the server. Using a network file server may reduce this, but it introduces some issues with permissions and network latency.

On the other hand, nothing is simpler than saving a version of a request to a file. That, along with disk-based caching's long history in other sectors of programming, made it a natural early choice.

Moreover, it's not entirely fair to suggest that disk-based caching's days are over. Faster drives, often SSDs, have reopened the potential for using non-ephemeral storage for quick access.

Let's take a quick look at how we can design a disk-based cache middleware solution for our API to reduce load and bottlenecks in heavy traffic.

The first consideration to take into account is what to cache. We would never want to allow the `PUT`, `POST`, and `DELETE` requests to cache for obvious reasons, as we don't want duplication of data nor erroneous responses to `DELETE` or `POST` requests that indicate that a resource has been created or deleted when in fact it hasn't.

So, we know that we're only caching the `GET` requests or listings of data. This is the only data we have that can be "outdated" in the sense that we can accept some staleness without making major changes in the way the application operates.

Let's start with our most basic request, `/api/users`, which returns a list of users in our system, and introduce some middleware for caching to a disk. Let's set it up as a skeleton to explain how we evaluate:

```
package diskcache

import
(
)

type CacheItem struct {

}
```

Our `CacheItem` struct is the only real element in the package. It consists of either a valid cache hit (and information about the cached element including the last modification time, contents, and so on) or a cache miss. A cache miss will return to our API that either the item does not exist or has surpassed the time-to-live (TTL). In this case, the `diskcache` package will then set the cache to file:

```
func SetCache() {

}
```

Here is where we'll do this. If a request has no cached response or the cache is invalid, we'll need to get the results back so that we can save it. This makes the middleware part a little trickier, but we'll show you how to handle this shortly. The following `GetCache()` function looks into our cache directory and either finds and returns a cache item (whether valid or not) or produces a false value:

```
func GetCache() (bool, CacheItem) {

}
```

The following `Evaluate()` function will be our primary point of entry, passing to `GetCache()` and possibly `SetCache()` later, if we need to create or recreate our cache entry:

```
func Evaluate(context string, value string, in ...[]string) (bool,
CacheItem) {


}
```

In this structure, we'll utilize a context (so that we can delineate between request types), the resulting value (for saving), and an open-ended variadic of strings that we can use as qualifiers for our cache entry. By this, we mean the parameters that force a unique cache file to be produced. Let's say we designate `page` and `search` as two such qualifiers. Page 1 requests will be different than page 2 requests and they will be cached separately. Page 1 requests for a search for Nathan will be different from page 1 requests for a search for Bob, and so on.

This point is very strict for hard files because we need to name (and look up) our cache files in a reliable and consistent way, but it's also important when we save caches in a datastore.

With all of this in mind, let's examine how we will discern a cacheable entry

Enabling filtering

Presently our API does not accept any specific parameters against any of our `GET` requests, which return lists of entities or specific details about an entity. Examples here include a list of users, a list of status updates, or a list of relationships.

You may note that our `UsersRetrieve()` handler presently returns the next page in response to a `start` value and a `limit` value. Right now this is hard-coded at a start value of `0` and a limit value of `10`.

In addition, we have a `Pragma: no-cache` header that is being set. We obviously don't want that. So, to prepare for caching, let's add a couple of additional fields that clients can use to find particular users they're looking for by attributes.

The first is a start and a limit, which dictates a pagination of sorts. What we now have is this:

```
start := 0
limit := 10

next := start + limit
```

Let's make this responsive to the request first by accepting a start:

```
start := 0
if len(r.URL.Query()["start"]) > 0 {
  start = r.URL.Query()["start"][0]
}
limit := 10
if len(r.URL.Query()["limit"]) > 0 {
  start = r.URL.Query()["limit"][0]
}
if limit > 50 {
  limit = 50
}
```

Now, we can accept a start value as well as a limit value. Note that we also put a cap on the number of results that we'll return. Any results that are more than 50 will be ignored and a maximum of 50 results will be returned.

Transforming a disk cache into middleware

Now we'll take the skeleton of `diskcache`, turn it into a middleware call, and begin to speed up our `GET` requests:

```
package diskcache

import
(
  "errors"
  "io/ioutil"
  "log"
  "os"
  "strings"
  "sync"
```

```
    "time"
  )

  const(
    CACHEDIR = "/var/www/cache/"
  )
```

This obviously represents a strict location for cache files, but it can also be branched into subdirectories that are based on a context, for example, our API endpoints in this case. So, `/api/users` in a `GET` request would map to `/var/www/cache/users/get/`. This reduces the volume of data in a single directory:

```
  var MaxAge int64  = 60
  var(
    ErrMissingFile = errors.New("File Does Not Exist")
    ErrMissingStats = errors.New("Unable To Get File Stats")
    ErrCannotWrite = errors.New("Cannot Write Cache File")
    ErrCannotRead = errors.New("Cannot Read Cache File")
  )

  type CacheItem struct {
    Name string
    Location string
    Cached bool
    Contents string
    Age int64
  }
```

Our generic `CacheItem` struct consists of the file's name, its physical location, the age in seconds, and the contents, as mentioned in the following code:

```
  func (ci *CacheItem) IsValid(fn string) bool {
    lo := CACHEDIR + fn
    ci.Location = lo

    f, err := os.Open(lo)
    defer f.Close()
    if err != nil {
      log.Println(ErrMissingFile)
      return false
    }

    st, err := f.Stat()
    if err != nil {
      log.Println(ErrMissingStats)
      return false
    }

    ci.Age := int64(time.Since(st.ModTime()).Seconds())
    return (ci.Age <= MaxAge)
  }
```

Our `IsValid()` method first determines whether the file exists and is readable, if it's older than the `MaxAge` variable. If it cannot be read or if it's too old, then we return false, which tells our `Evaluate()` entry point to create the file. Otherwise, we return true, which directs the `Evaluate()` function to perform a read of the existing cache file.

```
  func (ci *CacheItem) SetCache() {
    f, err := os.Create(ci.Location)
    defer f.Close()
    if err != nil {
      log.Println(err.Error())
    } else {
      FileLock.Lock()
      defer FileLock.Unlock()
      _, err := f.WriteString(ci.Contents)
      if err != nil {
        log.Println(ErrCannotWrite)
      } else {
        ci.Age = 0
      }
    }
    log.Println(f)
  }
```

In our imports section, you may note that the `sync` package is called; `SetCache()` should, in production at least, utilize a mutex to induce locking on file operations. We use `Lock()` and `Unlock()` (in a defer) to handle this.

```
  func (ci *CacheItem) GetCache() error {
    var e error
    d, err := ioutil.ReadFile(ci.Location)
    if err == nil {
      ci.Contents = string(d)
    }
    return err
```

```
}

func Evaluate(context string, value string, expireAge int64,
  qu ...string) (error, CacheItem) {


  var err error
  var ci CacheItem
  ci.Contents = value
  ci.Name = context + strings.Join(qu,"-")
  valid := ci.IsValid(ci.Name)
```

Note that our filename here is generated by joining the parameters in the `qu` variadic parameter. If we want to fine-tune this, we will need to sort the parameters alphabetically and this will prevent cache misses if the parameters are supplied in a different order.

Since we control the originating call, that's low-risk. However, since this is built as a shared library, it's important that the behavior should be fairly consistent.

```
  if !valid {
    ci.SetCache()
    ci.Cached = false
  } else {
    err = ci.GetCache()
    ci.Cached = true
  }

  return err, ci
}
```

We can test this pretty simply using a tiny example that just writes files by value:

```
package main

import
(
  "fmt"
  "github.com/nkozyra/api/diskcache"
)

func main() {
  err,c := diskcache.Evaluate("test","Here is a value that will
    only live for 1 minute",60)
  fmt.Println(c)
```

```
    if err != nil {
      fmt.Println(err)
    }
    fmt.Println("Returned value is",c.Age,"seconds old")
    fmt.Println(c.Contents)
  }
```

If we run this, then change the value of `Here is a value ...`, and run it again within 60 seconds, we'll get our cached value. This shows that our diskcache package saves and returns values without hitting what could otherwise be a backend bottleneck.

So, let's now put this in front of our `UsersRetrieve()` handler with some optional parameters. By setting our cache by `page` and `search` as cacheable parameters, we'll mitigate any load-based impact on our database.

Caching in distributed memory

Similar to disk-based caching, we're bound to a single entity key with simple in-memory caching although this is still a useful alternative to disk caching.

Replacing the disk with something like Memcache(d) will allow us to have very fast retrieval, but will provide us with no benefit in terms of keys. In addition, the potential for large amounts of duplication means that our memory storage that is generally smaller than physical storage might become an issue.

However, there are a number of ways to sneak into in-memory or distributed memory caching. We won't be showing you that drop-in replacement, but through a segue with one NoSQL solution, you can easily translate two types of caching into a strict, memory-only caching option.

Using NoSQL as a cache store

Unlike Memcache(d), with a datastore or a database we have the ability to do more complex lookups based on non-chained parameters.

For example, in our `diskcache` package, we chain together parameters such as `page` and `search` in such a way that our key (in this case a filename) is something like `getusers_1_nathan.cache`.

It is essential that these keys are generated in a consistent and reliable way for lookup since any change results in a cache miss instead of an expected hit, and we will need to rebuild our cached request, which will completely eliminate the intended benefit.

For databases, we can do very high-detail column lookups for cache requests, but, given the nature of relational databases, this is not a good solution. After all, we built the caching layer very specifically to avoid hitting common bottlenecks such as a RDBMS.

For the sake of an example, we'll again utilize MongoDB as a way to compile and lookup our cache files with high throughput and availability and with the extra flexibility that is afforded to parameter-dependent queries.

In this case, we'll add a basic document with just a page, search, contents, and a modified field. The last field will serve as our timestamp for analysis.

Despite `page` being a seemingly obvious integer field, we'll create it as a string in MongoDB to avoid type conversion when we do queries.

```
package memorycache
```

For obvious reasons, we'll call this `memorycache` instead of memcache to avoid any potential confusion.

```
import
(
  "errors"
  "log"
  mgo "gopkg.in/mgo.v2"
  bson "gopkg.in/mgo.v2/bson"
  _ "strings"
  "sync"
  "time"
)
```

We've supplanted any OS and disk-based packages with the MongoDB ones. The BSON package is also included as part of making specific `Find()` requests.

> In a production environment, when looking for a key-value store or a memory store for such intents, one should be mindful of the locking mechanisms of the solution and their impact on your read/write operations.

```
const(
  MONGOLOC = "localhost"
)

var MaxAge int64  = 60
var Session mgo.Session
var Collection *mgo.Collection

var(
  ErrMissingFile = errors.New("File Does Not Exist")
  ErrMissingStats = errors.New("Unable To Get File Stats")
  ErrCannotWrite = errors.New("Cannot Write Cache File")
  ErrCannotRead = errors.New("Cannot Read Cache File")

  FileLock sync.RWMutex
)

type CacheItem struct {
  Name string
  Location string
  Contents string
  Age int64
  Parameters map[string] string
}
```

It's worth noting here that MongoDB has a time-to-live concept for data expiration. This might remove the necessity to manually expire content but it may not be available in alternative store platforms.

```
type CacheRecord struct {
  Id bson.ObjectId `json:"id,omitempty" bson:"_id,omitempty"`
  Page string
  Search string
  Contents string
  Modified int64
}
```

Note the literal identifiers in the `CacheRecord` struct; these allow us to generate MongoDB IDs automatically. Without this, MongoDB will complain about duplicate indices on `_id_`. The following `IsValid()` function literally returns information about a file in our `diskcache` package. In a `memorycache` version, we will only return one piece of information, whether or not a record exists within the requested age.

```
func (ci *CacheItem) IsValid(fn string) bool {
  now := time.Now().Unix()
  old := now - MaxAge

  var cr CacheRecord
  err := Collection.Find(bson.M{"page":"1", "modified":
    bson.M{"$gt":old} }).One(&cr)
  if err != nil {
    return false
  } else {
    ci.Contents = cr.Contents
    return true
  }

  return false

}
```

Note also that we're not deleting old records. This may be the logical next step to keep cache records snappy.

```
func (ci *CacheItem) SetCache() {
  err := Collection.Insert(&CacheRecord{Id: bson.NewObjectId(),
    Page:ci.Parameters["page"],Search:ci.Parameters
    ["search"],Contents:ci.Contents,Modified:time.Now().Unix()})
  if err != nil {
    log.Println(err.Error())
  }
}
```

Whether or not we find a record, we insert a new one in the preceding code. This gives us the most recent record when we do a lookup and it also allows us to have some sense of revision control in a way. You can also update the record to eschew revision control.

```
func init() {
  Session, err := mgo.Dial(MONGOLOC)
  if err != nil {
    log.Println(err.Error())
  }
  Session.SetMode(mgo.Monotonic, true)
  Collection = Session.DB("local").C("cache")
  defer Session.Ping()
}

func Evaluate(context string, value string, expireAge int64, param
  map[string]string) (error, CacheItem) {

  MaxAge = expireAge
  defer Session.Close()

  var ci CacheItem
  ci.Parameters = param
  ci.Contents = value
  valid := ci.IsValid("bah:")
  if !valid {
    ci.SetCache()
  }

  var err error

  return err, ci
}
```

This operates in much the same way as `diskcache` except that, instead of a list of unstructured parameter names, we provide key/value pairs in the `param` hash map.

So, the usage changes a little bit. Here's an example:

```
package main

import
(
  "fmt"
  "github.com/nkozyra/api/memorycache"
)

func main() {
  parameters := make( map[string]string )
  parameters["page"] = "1"
  parameters["search"] = "nathan"
  err,c := memorycache.Evaluate("test","Here is a value that will
    only live for 1 minute",60,parameters)
  fmt.Println(c)
  if err != nil {
    fmt.Println(err)
  }
  fmt.Println("Returned value is",c.Age,"seconds old")
  fmt.Println(c.Contents)
}
```

When we run this, we'll set our content in the datastore and this will last for 60 seconds before it becomes invalid and recreates the cache contents in a second row.

Implementing a cache as middleware

To place this cache in the middleware chain for all of our `GET` requests, we can implement the strategy that we outlined above and add a caching middleware element.

Using our example from earlier, we can implement this at the front of the chain using our `middleware()` function:

```
Routes.HandleFunc("/api/users", middleware(DiskCache,
  UsersRetrieve,DiskCacheSave) ).Methods("GET")
```

This allows us to execute a `DiskCache()` handler before the `UsersRetrieve()` function. However, we'll also want to save our response if we don't have a valid cache, so we'll also call `DiskCacheSave()` at the end. The `DiskCache()` middleware handler will block the chain if it receives a valid cache. Here's how that works:

```
func DiskCache(h http.HandlerFunc) http.HandlerFunc {
  start := 0
  q := r.URL.Query()
  if len(r.URL.Query()["start"]) > 0 {
    start = r.URL.Query()["start"][0]
  }
  limit := 10
  if len(r.URL.Query()["limit"]) > 0 {
    limit = q["limit"][0]
  }
  valid, check := diskcache.Evaluate("GetUsers", "", MaxAge,
    start, limit)
  fmt.Println("Cache valid",valid)
  if check.Cached  {
    return func(w http.ResponseWriter, r *http.Request) {
      fmt.Fprintln(w, check.Contents)
    }
  } else {
    return func(w http.ResponseWriter, r *http.Request) {
      h.ServeHTTP(w, r)
    }
  }
}
```

If we get `check.Cached` as true, we simply serve the contents. Otherwise, we continue on.

One minor modification to our primary function is necessary to transfer the contents to our next function right before writing the function:

```
    r.CacheContents = string(output)
    fmt.Fprintln(w, string(output))
  }
```

And then, `DiskCacheSave()` can essentially be a duplicate of `DiskCache`, except that it will set the actual contents from the `http.Request` function:

```
  func DiskCacheSave(h http.HandlerFunc) http.HandlerFunc {
    start := 0
    if len(r.URL.Query()["start"]) > 0 {
      start = r.URL.Query()["start"][0]
    }
    limit := 10
    if len(r.URL.Query()["limit"]) > 0 {
      start = r.URL.Query()["limit"][0]
    }
    valid, check := diskcache.Evaluate("GetUsers", r.CacheContents,
      MaxAge, start, limit)
    fmt.Println("Cache valid",valid)
    return func(w http.ResponseWriter, r *http.Request) {
      h.ServeHTTP(w, r)
    }

  }
```

Using a frontend caching proxy in front of Go

Another tool in our toolbox is utilizing front-end caching proxies (as we did in *Chapter 7, Working with Other Web Technologies*) as our request-facing cache layer.

In addition to traditional web servers such as Apache and Nginx, we can also employ services that are intended almost exclusively for caching, either in place of, in front of, or in parallel with the said servers.

Without going too deeply into this approach, we can replicate some of this functionality with better performance from outside the application. We'd be remiss if we didn't at least broach this. Tools such as Nginx, Varnish, Squid, and Apache have built-in caching for reverse-proxy servers.

For production-level deployments, these tools are probably more mature and better suited for handling this level of caching.

You can find more information on Nginx and reverse proxy caching at `http://nginx.com/resources/admin-guide/caching/`.

Varnish and Squid are both built primarily for caching at this level as well. More detail on Varnish and Squid can be found at `https://www.varnish-cache.org/` and `http://www.squid-cache.org/` respectively.

Rate limiting in Go

Introducing caching to our API is probably the simplest way to demonstrate effective middleware strategy. We're able to now mitigate the risk of heavy traffic and move toward

One particularly useful place for this kind of middleware functionality in a web service is rate limiting.

Rate limiting exists in APIS with high traffic to allow consumers to use the application without potentially abusing it. Abuse in this case can just mean excessive access that can impact performance, or it can mean creating a deterrent for large-scale data acquisition.

Often, people will utilize APIs to create local indices of entire data collections, effectively spidering a site through an API. With most applications, you'll want to prevent this kind of access.

In either case, it makes sense to impose some rate limiting on certain requests within our application. And, importantly, we'll want this to be flexible enough so that we can do it with varying limits depending on the request time.

We can do this using a number of factors, but the two most common methods are as follows:

- Through the corresponding API user credentials
- Through the IP address of the request

In theory, we can also introduce rate limits on the underlying user by making a request per proxy. In a real-world scenario, this would reduce the risk of a third-party application being penalized for its user's usage.

The important factor is that we discover rate-limit-exceeded notations before delving into more complex calls, as we want to break the middleware chain at precisely that point if the rate limit has been exceeded.

For this rate-limiting middleware example, we'll again use MongoDB as a request store and a limit based on a calendar day from midnight to midnight. In other words, our limit per user resets every day at 12:01 a.m.

Storing actual requests is just one approach. We can also read from web server logs or store them in memory. However, the most lightweight approach is keeping them in a datastore.

```
package ratelimit

import
(
  "errors"
  "log"
  mgo "gopkg.in/mgo.v2"
  bson "gopkg.in/mgo.v2/bson"
  _ "strings"
  "time"
)

const(
  MONGOLOC = "localhost"
)
```

This is simply our MongoDB host or hosts. Here, we have a struct with boundaries for the beginning and end of a calendar day:

```
type Requester struct {
  Id bson.ObjectId `json:"id,omitempty" bson:"_id,omitempty"`
  IP string
  APIKey string
  Requests int
  Timestamp int64
  Valid bool
}

type DateBounds struct {
  Start int64
  End int64
}
```

The following `CreateDateBounds()` function calculates today's date and then adds `86400` seconds to the returned `Unix()` value (effectively 1 day).

```
var (
  MaxDailyRequests = 15
  TooManyRequests = errors.New("You have exceeded your daily limit
    of requests.")
  Bounds DateBounds
  Session mgo.Session
  Collection *mgo.Collection
)

func createDateBounds() {
  today := time.Now()
  Bounds.Start = time.Date(today.Year(), today.Month(),
    today.Day(), 0, 0, 0, 0, time.UTC).Unix()
  Bounds.End = Bounds.Start + 86400
}
```

With the following `RegisterRequest()` function, we're simply logging another request to the API. Here again, we're only binding the request to the IP, adding an authentication key, user ID, or

```
func (r *Requester) CheckDaily() {

  count, err := Collection.Find(bson.M{"ip": r.IP, "timestamp":
    bson.M{"$gt":Bounds.Start, "$lt":Bounds.End } }).Count()
  if err != nil {
    log.Println(err.Error())
  }
  r.Valid = (count <= MaxDailyRequrests)
}

func (r Requester) RegisterRequest() {
  err := Collection.Insert(&Requester{Id: bson.NewObjectId(),
    IP: r.IP, Timestamp: time.Now().Unix()})
  if err != nil {
    log.Println(err.Error())
  }

}
```

The following code is a simple, standard initialization setup, except for the `createDateBounds()` function, which simply sets the start and end of our lookup:

```
func init() {
  Session, err := mgo.Dial(MONGOLOC)
  if err != nil {
    log.Println(err.Error())
  }
  Session.SetMode(mgo.Monotonic, true)
  Collection = Session.DB("local").C("requests")
  defer Session.Ping()
  createDateBounds()
}
```

The following `CheckRequest()` function acts as the coordinating function for the entire process; it determines whether any given request exceeds the daily limit and returns the `Valid` status property:

```
func CheckRequest(ip string) (bool) {
  req := Requester{ IP: ip }
  req.CheckDaily()
  req.RegisterRequest()

  return req.Valid
}
```

Implementing rate limiting as middleware

Unlike the cache system, turning the rate limiter into middleware is much easier. Either the IP address is rate-limited, or it's not and we move on.

Here's an example for updating users:

```
  Routes.HandleFunc("/api/users/{id:[0-9]+}",
    middleware(RateLimit,UsersUpdate)).Methods("PUT")
```

And then, we can introduce a `RateLimit()` middleware call:

```
func RateLimit(h http.HandlerFunc) http.HandlerFunc {
  return func(w http.ResponseWriter, r *http.Request) {
    if (ratelimit.CheckRequest(r.RemoteAddr) == false {
      fmt.Fprintln(w,"Rate limit exceeded")
    } else {
      h.ServeHTTP(w,r)
    }
  }
}
```

This allows us to block the middleware chain if our `ratelimit.CheckRequest()` call fails and prevents any more processing-intensive parts of our API from being called.

Implementing SPDY

If there's one thing you can say about Google's vast ecosystem of products, platforms, and languages, it's that there's a perpetual, consistent focus on one thing that spans all of them—a need for speed.

> We briefly mentioned the SPDY pseudo-protocol in *Chapter 7, Working with Other Web Technologies*. You can read more about SPDY from its whitepaper at `http://www.chromium.org/spdy/spdy-whitepaper`.

As Google (the search engine) quickly scaled from being a student project to the most popular site on Earth to the de facto way people find anything anywhere, scaling the product and its underlying infrastructure became key.

And, if you think about it, this search engine is heavily dependent on sites being available; if the sites are fast, Google's spiders and indexers will be faster and the results will be more current.

Much of this is behind Google's *Let's Make the Web Faster* campaign, which aims to help developers on both the backend and frontend by being cognizant of and pushing toward speed as the primary consideration.

Google is also behind the SPDY pseudo-protocol that augments HTTP and operates as a stopgap set of improvements, many of which are finding their way into the standardization of HTTP/2.

There are a lot of SPDY implementations that are written for Go, and SPDY seems to be a particularly popular project to embrace as it's not yet supported directly in Go. Most implementations are interchangeable drop-in replacements for `http` in `net/http`. In most practical cases, you can get these benefits by simply leaving SPDY to a reverse proxy such as HAProxy or Nginx.

Here are a few SPDY implementations that implement both secure and nonsecure connections and that are worth checking out and comparing:

The `spdy.go` file from Solomon Hykes: `https://github.com/shykes/spdy-go`

The `spdy` file from Jamie Hall: `https://github.com/SlyMarbo`

We'll first look at `spdy.go` from the preceding list. Switching our `ListenAndServe` function is the easiest first step, and this approach to implement SPDY is fairly common.

Here's how to use `spdy.go` as a drop-in replacement in our `api.go` file:

```
wg.Add(1)
go func() {
  //http.ListenAndServeTLS(SSLport, "cert.pem", "key.pem",
    Routes)
  spdy.ListenAndServeTLS(SSLport, "cert.pem", "key.pem", Routes)
  wg.Done()
}()
```

Pretty simple, huh? Some SPDY implementations make serving pages through the SPDY protocol in lieu of `HTTP/HTTP` semantically indistinguishable.

For some Go developers, this counts as an idiomatic approach. For others, the protocols are different enough that having separate semantics is logical. The choice here depends on your preference.

However, there are a few other considerations to take into account. First, SPDY introduces some additional features that we can utilize. Some of these are baked-in like header compression.

Detecting SPDY support

For most clients, detecting SPDY is not something that you need to worry about too much, as SPDY support relies on TLS/SSL support.

Summary

In this chapter, we worked at a few concepts that are important to highly-performant APIs. These primarily included rate limiting and disk and memory caching that were executed through the use of custom-written middleware.

Utilizing the examples within this chapter, you can implement any number of middleware-reliant services to keep your code clean and introduce better security, faster response times, and more features.

In the next and final chapter, we'll focus on security-specific concepts that should lock in additional concerns with rate limits, denial-of-service detection, and mitigating and preventing attempts at code and SQL injections.

11
Security

Before we begin this chapter, it's absolutely essential to point out one thing—though security is the topic of the last chapter of this book, it should never be the final step in application development. As you develop any web service, security should be considered prominently at every step. By considering security as you design, you limit the impact of top-to-bottom security audits after an application's launch.

With that being said, the intent here is to point out some of the larger and more rampant security flaws and look at ways in which we can allay their impact on our web service using standard Go and general security practices.

Of course, out of the box, Go provides some wonderful security features that are disguised as solely good programming practices. Using all the included packages and handling all the errors are not only useful for developing good habits, but they also help you to secure your application.

However, no language can offer perfect security nor can it stop you from shooting yourself in the foot. In fact, the most expressive and utilitarian languages often make that as easy as possible.

There's also a large trade-off when it comes to developing your own design as opposed to using an existing package (as we've done throughout this book), be it for authentication, database interfaces, or HTTP routing or middleware. The former can provide quick resolution and less exposure of errors and security flaws.

There is also some security through obscurity that is offered by building your own application, but swift responses to security updates and a whole community whose eyes are on your code beats a smaller, closed-source project.

In this chapter, we'll look at:

- Handling error logging for security purposes
- Preventing brute-force attempts
- Logging authentication attempts
- Input validation and injection mitigation
- Output validation

Lastly, we'll look at a few production-ready frameworks to look at the way they handle API and web service integrations and associated security.

Handling error logging for security

A critical step on the path to a secure application involves the use of comprehensive logging. The more data you have, the better you can analyze potential security flaws and look at the way your application is used.

Even so, the "log it all" approach can be somewhat difficult to utilize. After all, finding the needles in the haystack can be particularly difficult if you have all the hay.

Ideally, we'd want to log all errors to file and have the ability to segregate other types of general information such as SQL queries that are tied to users and/or IP addresses.

In the next section, we'll look at logging authentication attempts but only in memory/an application's lifetime to detect brute-force attempts. Using the log package more extensively allows us to maintain a more persistent record of such attempts.

The standard way to create log output is to simply set the output of the general log, `Logger`, like this:

```
dbl, err := os.OpenFile("errors.log", os.O_CREATE | os.RDWR | os.O_
APPEND, 0666)
  if err != nil {
    log.Println("Error opening/creating database log file")
  }
defer dbl.Close()

log.SetOutput(dbl)
```

This allows us to specify a new file instead of our default `stdout` class for logging our database errors for analyzing later.

However, if we want multiple log files for different errors (for example, database errors and authentication errors), we can break these into separate loggers:

```
package main

import (
  "log"
  "os"
)

var (
  Database       *log.Logger
  Authentication *log.Logger
  Errors         *log.Logger
)

func LogPrepare() {
  dblog, err := os.OpenFile("database.log",
    os.O_CREATE|os.O_APPEND|os.O_WRONLY, 0666)
  if err != nil {
    log.Println(err)
  }
  authlog, err := os.OpenFile("auth.log",
    os.O_CREATE|os.O_APPEND|os.O_WRONLY, 0666)
  if err != nil {
    log.Println(err)
  }
  errlog, err := os.OpenFile("errors.log",
    os.O_CREATE|os.O_APPEND|os.O_WRONLY, 0666)
  if err != nil {
    log.Println(err)
  }

  Database = log.New(dblog, "DB:", log.Ldate|log.Ltime)
  Authentication = log.New(authlog, "AUTH:", log.Ldate|log.Ltime)
  Errors = log.New(errlog, "ERROR:",
    log.Ldate|log.Ltime|log.Lshortfile)
}
```

Here, we instantiate separate loggers with specific formats for our log files:

```
func main() {
  LogPrepare()

  Database.Println("Logging a database item")
  Authentication.Println("Logging an auth attempt item")
  Errors.Println("Logging an error")

}
```

By building separate logs for elements of an application in this manner, we can divide and conquer the debugging process.

As for logging SQL, we can make use of the `sql.Prepare()` function instead of using `sql.Exec()` or `sql.Query()` to keep a reference to the query before executing it.

The `sql.Prepare()` function returns a `sql.Stmt` struct, and the query itself, which is represented by the variable query, is not exported. You can, however, use the struct's value itself in your log file:

```
d, _ := db.Prepare("SELECT fields FROM table where column=?")
Database.Println(d)
```

This will leave a detailed account of the query in the log file. For more detail, IP addresses can be appended to the `Stmt` class for more information.

Storing every transactional query to a file may, however, end up becoming a drag on performance. Limiting this to data-modifying queries and/or for a short period of time will allow you to identify potential issues with security.

There are some third-party libraries for more robust and/or prettier logging. Our favorite is go-logging, which implements multiple output formats, partitioned debugging buckets, and expandable errors with attractive formatting. You can read more about these at `https://github.com/op/go-logging` or download the documentation via the `go get github.com/op/go-logging` command.

Preventing brute-force attempts

Perhaps the most common, lowest-level attempt at circumventing the security of any given system is the brute-force approach.

From the point of view of an attacker, this makes some sense. If an application designer allows an infinite amount of login attempts without penalty, then the odds of this application enforcing a good password-creation policy are low.

This makes it a particularly vulnerable application. And, even if the password rules are in place, there is still a likelihood to use dictionary attacks to get in.

Some attackers will look at rainbow tables in order to determine a hashing strategy, but this is at least in some way mitigated by the use of unique salts per account.

Brute-force login attacks were actually often easier in the offline days because most applications did not have a process in place to automatically detect and lock account access attempts with invalid credentials. They could have, but then there would also need to be a retrieval authority process—something like "e-mail me my password".

With services such as our social network, it makes a great deal of sense to either lock accounts or temporarily disable logins after a certain point.

The first is a more dramatic approach, requiring direct user action to restore an account; often, this also entails greater support systems.

The latter is beneficial because it thwarts brute-force attempts by greatly slowing the rate of attempts, and rendering most attacks useless for all practical purposes without necessarily requiring user action or support to restore access.

Knowing what to log

One of the hardest things to do when it comes to logging is deciding what it is that you need to know. There are several approaches to this, ranging from logging everything to logging only fatal errors. All the approaches come with their own potential issues, which are largely dependent on a trade-off between missing some data and wading through an impossible amount of data.

The first consideration that we'll need to make is what we should log in memory—only failed authentications or attempts against API keys and other credentials.

It may also be prudent to note login attempts against nonexistent users. This will tell us that someone is likely doing something nefarious with our web service.

Next, we'll want to set a lower threshold or the maximum amount of login attempts before we act.

Let's start by introducing a `bruteforcedetect` package:

```
package bruteforcedetect

import
(
)

var MaxAttempts = 3
```

We can set this directly as a package variable and modify it from the calling application, if necessary. Three attempts are likely lower than what we'd like for a general invalid login threshold, particularly one that automatically bans the IP:

```
type Requester struct {
  IP string
  LoginAttempts int
  FailedAttempts int
  FailedInvalidUserAttempts int
}
```

Our `Requester` struct will maintain all incremental values associated with any given IP or hostname, including general attempts at a login, failed attempts, and failed attempts wherein the requested user does not actually exist in our database:

```
func Init() {

}

func (r Requester) Check() {

}
```

We don't need this as middleware as it needs to react to just one thing—authentication attempts. As such, we have a choice as it relates to storage of authentication attempts. In a real-world environment, we may wish to grant this process more longevity than we will here. We could store these attempts directly into memory, a datastore, or even to disk.

However, in this case, we'll just let this data live in the memory space of this application by creating a map of the `bruteforce.Requester` struct. This means that if our server reboots, we lose these attempts. Similarly, it means that multiple server setups won't necessarily know about attempts on other servers.

Both these problems can be easily solved by putting less ephemeral storage behind the logging of bad attempts, but we'll keep it simple for this demonstration.

In our `api.go` file, we'll bring in `bruteforce` and create our map of `Requesters` when we start the application:

```
package main

import (
…
    "github.com/nkozyra/api/bruteforce"
)

var Database *sql.DB
var Routes *mux.Router
var Format string
var Logins map[string] bruteforce.Requester
```

And then, of course, to take this from being a nil map, we'll initialize it when our server starts:

```
func StartServer() {

  LoginAttempts = make(map[string] bruteforce.Requester)
  OauthServices.InitServices()
```

We're now ready to start logging our attempts.

If you've decided to implement middleware for login attempts, make the adjustment here by simply putting these changes into the middleware handler instead of the separate function named `CheckLogin()` that we originally called.

No matter what happens with our authentication—be it a valid user, valid authentication; a valid user, invalid authentication; or an invalid user—we want to add this to our `LoginAttempts` function of the respective `Requester` struct.

We'll bind each `Requester` map to either our IP or hostname. In this case, we will use the IP address.

The net package has a function called SplitHostPort that properly explodes our RemoteAddr value from the http.Request handler, as follows:

```
ip,_,_ := net.SplitHostPort(r.RemoteAddr)
```

You can also just use the entire r.RemoteAddr value, which may be more comprehensive:

```
func CheckLogin(w http.ResponseWriter, r *http.Request) bool {
  if val, ok := Logins[r.RemoteAddr]; ok {
    fmt.Println("Previous login exists",val)
  } else {
    Logins[r.RemoteAddr] = bruteforce.Requester{IP: r.RemoteAddr,
      LoginAttempts:0, FailedAttempts: 0,
      FailedValidUserAttempts: 0, }
  }

  Logins[r.RemoteAddr].LoginAttempts += 1
```

This means that no matter what, we invoke another attempt to the tally.

Since CheckLogin() will always create the map's key if it doesn't exist, we're free to safely evaluate on this key further down the authentication pipeline. For example, in our UserLogin() handler, which processes an e-mail address and a password from a form and checks against our database, we first call UserLogin() before checking the submitted values:

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

  w.Header().Set("Access-Control-Allow-Origin", "*")
  fmt.Println("Attempting User Login")

  Response := UpdateResponse{}
 CheckLogin(w,r)
```

If we check against our maximum login attempts following the CheckLogin() call, we'll never allow database lookups after a certain point.

In the following code of the `UserLogin()` function, we compare the hash from the submitted password to the one stored in the database and return an error on an unsuccessful match. Let's use that to increment the `FailedAttempts` value:

```
if (dbPassword == expectedPassword) {
  // ...
} else {
  fmt.Println("Incorrect password!")
  _, httpCode, msg := ErrorMessages(401)
  Response.Error = msg
  Response.ErrorCode = httpCode
  Logins[r.RemoteAddr].FailedAttempts =
    Logins[r.RemoteAddr].FailedAttempts + 1
  http.Error(w, msg, httpCode)
}
```

This simply increases our general `FailedAttempts` integer value with each invalid login per IP.

However, we're not yet doing anything with this. To inject it as a blocking element, we'll need to evaluate it after the `CheckLogin()` call to initialize the map's hash if it does not exist yet:

In the preceding code, you may notice that the mutable `FailedAttempts` value that is bound by `RemoteAddr` could theoretically be susceptible to a race condition, causing unnatural increments and premature blocking. A mutex or similar locking mechanism may be used to prevent this behavior.

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

  w.Header().Set("Access-Control-Allow-Origin", "*")
  fmt.Println("Attempting User Login")

if Logins[r.RemoteAddr].Check() == false {
  return
}
```

This call to `Check()` prevents banned IPs from even accessing our database at the login endpoint, which can still cause additional strain, bottlenecks, and potential service disruptions:

```
      Response := UpdateResponse{}
      CheckLogin(w,r)
      if Logins[r.RemoteAddr].Check() == false {
        _, httpCode, msg := ErrorMessages(403)
        Response.Error = msg
        Response.ErrorCode = httpCode
        http.Error(w, msg, httpCode)
        return
      }
```

And, to update our `Check()` method from a brute-force attack, we will use the following code:

```
    func (r Requester) Check() bool {
      return r.FailedAttempts <= MaxAttempts
    }
```

This supplies us with an ephemeral way to store information about login attempts, but what if we want to find out whether someone is simply testing account names along with passwords, ala "guest" or "admin?"

To do this, we'll just add an additional check to `UserLogin()` to see whether the requested e-mail account exists. If it does, we'll just continue. If it does not exist, we'll increment `FailedInvalidUserAttempts`. We can then make a decision about whether we should block access to the login portion of `UserLogin()` at a lower threshold:

```
      var dbPassword string
      var dbSalt string
      var dbUID int
      var dbUserCount int
      uexerr := Database.QueryRow("SELECT count(*) from users where
        user_email=?",email).Scan(&dbUserCount)
      if uexerr != nil {

      }
      if dbUserCount > 0 {
        Logins[r.RemoteAddr].FailedInvalidUserAttempts =
          Logins[r.RemoteAddr].FailedInvalidUserAttempts + 1
      }
```

If we decide that the traffic is represented by fully failed authenticated attempts (for example, invalid users), we can also pass that information to IP tables or our front-end proxy to block the traffic from even getting to our application.

Handling basic authentication in Go

One area at which we didn't look too deeply in the authentication section of *Chapter 7, Working with Other Web Technologies*, was basic authentication. It's worth talking about as a matter of security, particularly as it can be a very simple way to allow authentication in lieu of OAuth, direct login (with sessions), or keys. Even in the latter, it's entirely possible to utilize API keys as part of basic authentication.

The most critical aspect of basic authentication is an obvious one—**TLS**. Unlike methods that involve passing keys, there's very little obfuscation involved in the basic authentication header method, as beyond Base64 encoding, everything is essentially cleartext.

This of course enables some very simple man-in-the-middle opportunities for nefarious parties.

In *Chapter 7*, *Working with Other Web Technologies*, we explored the concept of creating transaction keys with shared secrets (similar to OAuth) and storing valid authentication via sessions.

We can grab usernames and passwords or API keys directly from the `Authorization` header and measure attempts on the API by including a check for this header at the top of our `CheckLogin()` call:

```
func CheckLogin(w http.ResponseWriter, r *http.Request) {
  bauth := strings.SplitN(r.Header["Authorization"][0], " ", 2)
  if bauth[0] == "Basic" {
    authdata, err := base64.StdEncoding.DecodeString(bauth[1])
    if err != nil {
      http.Error(w, "Could not parse basic auth",
        http.StatusUnauthorized)
      return
    }
      authparts := strings.SplitN(string(authdata),":",2)
      username := authparts[0]
      password := authparts[1]
    }else {
      // No basic auth header
    }
```

In this example, we can allow our `CheckLogin()` function to utilize either the data posted to our API to obtain username and password combinations, API keys, or authentication tokens, or we can also ingest that data directly from the header.

Handling input validation and injection mitigation

If a brute-force attack is a rather inelegant exercise in persistence, one in which the attacker has no access, input or injection attacks are the opposite. At this point, the attacker has some level of trust from the application, even if it is minimal.

SQL injection attacks can happen at any level in the application pipeline, but cross-site scripting and cross-site request forgeries are aimed less at the application and more at other users, targeting vulnerabilities to expose their data or bring other security threats directly to the application or browser.

In this next section, we'll examine how to keep our SQL queries safe through input validation, and then move onto other forms of input validation as well as output validation and sanitization.

Using best practices for SQL

There are a few very big security loopholes when it comes to using a relational database, and most of them apply to other methods of data storage. We've looked at a few of these loopholes such as properly and uniquely salting passwords and using secure sessions. Even in the latter, there is always some risk of session fixation attacks, which allow shared or persistent shared sessions to be hijacked.

One of the more pervasive attack vectors, which modern database adapters tend to eliminate, are injection attacks.

Injection attacks, particularly SQL injections, are among the most prevalent and yet most avoidable loopholes that can expose sensitive data, compromise accountability, and even make you lose control of entire servers.

A keen eye may have caught it, but earlier in this book, we deliberately built an unsafe query into our `api.go` file that can allow SQL injection.

Here is the line in our original `CreateUser()` handler:

```
sql := "INSERT INTO users set user_nickname='" + NewUser.Name +
  "', user_first='" + NewUser.First + "', user_last='" +
  NewUser.Last + "', user_email='" + NewUser.Email + "'"
q, err := database.Exec(sql)
```

It goes without saying, but constructing queries as a straight, direct SQL command is frowned upon in almost all languages.

A good general rule of thumb is to treat all externally produced data, including user input, internal or administrator user input, and external APIs as malicious. By being as suspicious as possible of user-supplied data, we improve the odds of catching potentially harmful injections.

Most of our other queries utilized the parameterized `Query()` function that allows you to add variadic parameters that correspond to the `?` tokens.

Remember that since we store the user's unique salt in the database (at least in our example), losing access to the MySQL database means that we also lose the security benefits of having a password salt in the first place.

This doesn't mean that all accounts' passwords are exposed in this scenario, but at this point, having direct login credentials for users would only be useful for exploiting other services if the users maintain poor personal password standards, that is, sharing passwords across services.

Validating output

Normally, the idea of output validation seems foreign, particularly when the data is sanitized on the input side.

Preserving the values as they were sent and only sanitizing them when they are output may make some sense, but it increases the odds that said values might not be sanitized on the way out to the API consumer.

There are two main ways in which a payload can be delivered to the end user, either in a stored attack where we, as the application, keep the vector verbatim on our server, or in a reflected attack wherein some code is appended via another method such as an e-mail message that includes the payload.

APIs and web services can sometimes be especially susceptible to not only **XSS** (short form for **Cross-Site Scripting**) but also **CSRF** (short form for **Cross-Site Request Forgery**).

We'll briefly look at both of these and the ways in which we can limit their efficacy within our web service.

Protection against XSS

Anytime we're dealing with user input that will later be translated into output for the consumption of other users, we need to be wary of Cross-Site Scripting or Cross-Site Request Forgery in the resulting data payload.

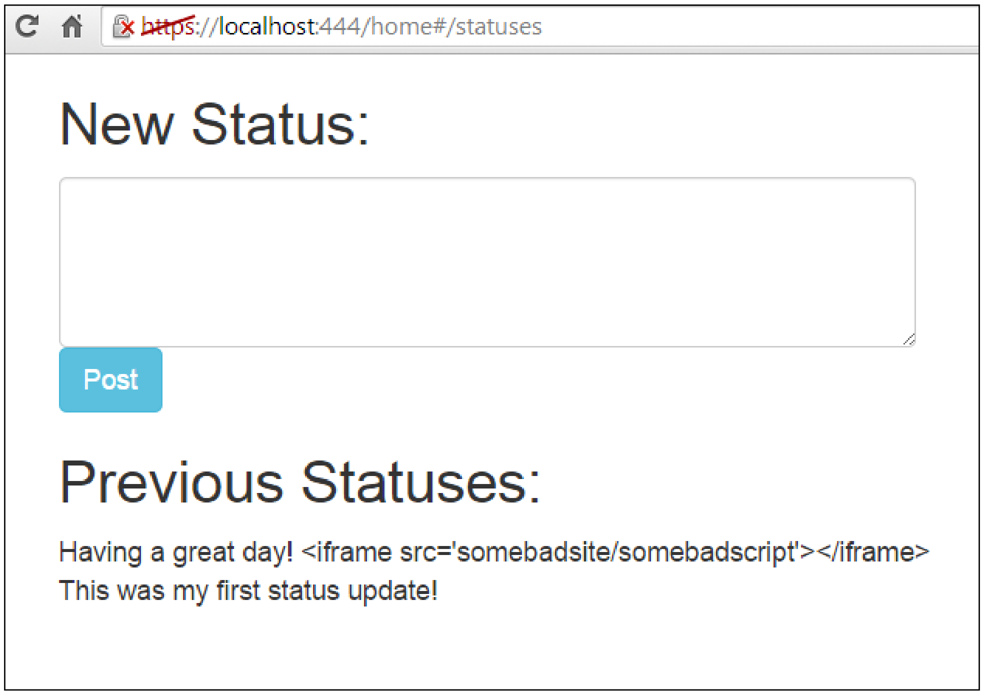
This isn't necessarily a matter solely for output validation. It can and should be addressed at the input stage as well. However, our output is our last line of defense between one user's arbitrary text and another user's consumption of that text.

Traditionally, this is best illustrated through something like the following nefarious piece of hypothetical code. A user hits our `/api/statuses` endpoint with a `POST` request, after authenticating it via whatever method is selected, and posts the following status:

```
url -X POST -H "Authorization: Basic dGVzdDp0ZXN0" -H "Cache-
  Control: no-cache" -H "Postman-Token: c2b24964-c12d-c183-dd7f-
  5c1365f5ae81" -H "Content-Type: multipart/form-data; boundary=---
  --WebKitFormBoundary7MA4YWxkTrZu0gW" -F "status=Having a great
  day! <iframe src='somebadsite/somebadscript'></iframe>"
  https://localhost/api/statuses
```

If presented in a template, as in our interface example, then this is a problem that will be mitigated automatically by using Go's template engine.

Let's take the preceding example data and see what it looks like on our interface's user profile page:



New Status:

Previous Statuses:

Having a great day! <iframe src='somebadsite/somebadscript'></iframe>
This was my first status update!

The `html/template` package automatically escapes the HTML output to prevent code injection, and it requires an override to allow any HTML tags to come through as originally entered.

However, as an API provider, we are agnostic towards the type of consuming application language and support or care given to sanitation of input.

The onus on escaping data is a matter that needs some consideration, that is, should the data that your application provides to clients come pre-sanitized or should it come with a usage note about sanitizing data? The answer in almost all cases is the first option, but depending on your role and the type of data, it could go either way. On the other hand, unsanitizing the data in certain situations (for example, APIs) on the frontend means potentially having to reformat data in many different ways.

Earlier in this chapter, we showed you some input validation techniques for allowing or disallowing certain types of data (such as characters, tags, and so on), and you can apply some of these techniques to an endpoint such as `/statuses`.

It makes more sense, however, to allow this data; but, sanitize it either before saving it to a database/datastore or returning it via an API endpoint. Here are two ways in which we can use the `http/template` package to do either.

First, when we accept data via the `/api/statuses` endpoint, we can utilize one or more of the functions in `html/template` to prevent certain types of data from being stored. The functions are as follows:

- `template.HTMLEscapeString`: This encodes HTML tags and renders the resulting string as non-HTML content
- `template.JSEscapeString()`: This encodes JavaScript-specific pieces of a string to prevent proper rendering

For the purpose of keeping this simple for potential output through HTML, we can just apply `HTMLEscapeString()` to our data, which will disable any JavaScript calls from executing:

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

  Response := CreateResponse{}
  UserID := r.FormValue("user")
  Status := r.FormValue("status")
  Token := r.FormValue("token")
  ConsumerKey := r.FormValue("consumer_key")

  Status = template.HTMLEscapeString(Status)
```

This makes the data escape on the input (`StatusCreate`) side. If we want to add JavaScript escaping (which, as noted earlier, may not be necessary), it should come before the HTML escaping, as noted here:

```
  Status = template.JSEscapeString(Status)
  Status = template.HTMLEscapeString(Status)
```

If in lieu of escaping on the input side, we wish to do it on the output side, the same template escape calls can be made as part of the respective status request API calls, like `/api/statuses`:

```
func StatusRetrieve(w http.ResponseWriter, r *http.Request) {
  var Response StatusResponse
  w.Header().Set("Access-Control-Allow-Origin", "*")
  loggedIn := CheckLogin(w, r)
  if loggedIn {

  } else {
    statuses,_ := Database.Query("select * from user_status where
      user_id=? order by user_status_timestamp desc",Session.UID)
    for statuses.Next() {

      status := Status{}
```

```
        statuses.Scan(&status.ID, &status.UID, &status.Time,
          &status.Text)
        status.Text = template.JSEscapeString(status.Text)
        status.Text = template.HTMLEscapeString(status.Text)
        Response.Statuses = append(Response.Statuses, status)
    }
```

If we want to attempt to detect and log attempts to pass specific HTML elements into input elements, we can create a new logger for XSS attempts and capture any text that matches a `<script>` element, a `<iframe>` element, or any other element.

Doing this can be as complex as a tokenizer or a more advanced security package or as simple as a regular expression match, as we will see in the following examples. First, we will look at the code in our logging setup:

```
var (
  Database       *log.Logger
  Authentication *log.Logger
  Errors         *log.Logger
  Questionable *log.Logger
)
```

And the changes in our initialization code are as follows:

```
questlog, err := os.OpenFile("injections.log",
  os.O_CREATE|os.O_APPEND|os.O_WRONLY, 0666)
if err != nil {
  log.Println(err)
}
Questionable = log.New(questlog, "XSS:", log.Ldate|log.Ltime)
```

And then, make the following changes back in our application's `StatusCreate` handler:

```
isinject, _ := regexp.MatchString("<(script|iframe).*",Status)
if isinject  {

}
```

Detecting tags this way, through regular expressions, is not airtight nor is it intended to be. Remember that we'll be sanitizing the data either on the input side or the output side, so if we can catch attempts through this method, it will give us some insight into potentially malicious attempts against our application.

If we want to be more idiomatic and comprehensive, we can simply sanitize the text and compare it with the original. If the two values do not match, we can surmise that HTML was included.

This does mean that we'll get a positive for innocuous HTML tags such as bold tags or table tags.

Using server-side frameworks in Go

We would be remiss if, while detailing how to build a web service from scratch, we didn't at least touch upon integrating or exclusively using some existing frameworks.

Although you'll never get the same experience by plugging in such a framework as you would by designing one from scratch, for practical purposes, there's often no reason to reinvent the wheel when you want to start a project.

Go has quite a few readily available and mature web/HTML frameworks, but it also has a handful of noteworthy frameworks that are specifically designed for web services with some of the delivery methods and additional hooks that you might expect to see.

By some measurements, it's fair to describe Gorilla as a framework; however, as the name implies, it's a little basic.

Whether you use an existing framework or choose to build your own (either for the experience or to completely customize it due to business requirements), you should probably consider doing a bit

We'll briefly look at a few of these frameworks and how they can simplify the development of small web-based projects.

Tiger Tonic

Tiger Tonic is specifically an API-centric framework, so we'll mention it first in this section. This takes a very idiomatic Go approach to developing JSON web services.

Responses are primarily intended to be in JSON only and multiplexing should seem pretty familiar to the style introduced by Gorilla.

Tiger Tonic also provides some quality logging features that allow you to funnel logs directly into the Apache format for more detailed analysis. Most importantly, it handles middleware in a way that allows some conditional operations based on the results of the middleware itself.

You can read more about Tiger Tonic at `https://github.com/rcrowley/go-tigertonic` or download the documentation using the `go get github.com/rcrowley/go-tigertonic` command.

Martini

The web framework Martini is one of the more popular web frameworks for the relatively young Go language, largely due to its similarity in design to both the `Node.js` framework Express and the popular Ruby-on-Rails framework Sinatra.

Martini also plays extraordinarily well with middleware, so much so that it's often brought in exclusively for this purpose. It also comes with a few standard middleware handlers like `Logger()` that takes care of logging in and out and `Recovery()` that recovers from panics and returns HTTP errors.

Martini is built for a large swath of web projects, and it may include more than what is necessary for a simple web service; however, it is an excellent all-inclusive framework that's worth checking out.

You can read more about Martini at `https://github.com/go-martini/martini` or download the documentation using the `go get github.com/go-martini/martini` command.

Goji

Unlike Martini, which is quite comprehensive and far-reaching, the Goji framework is minimalistic and lean. The primary advantages of Goji are its incredibly quick routing system, a low overhead for additional garbage collection, and robust middleware integrations.

Goji uses Alice for middleware, which we briefly touched on in an earlier chapter.

You can read more about the Goji micro framework at `https://goji.io/` and download it with the `go get github.com/zenazn/goji` and `go get github.com/zenazn/goji/web` commands.

Beego

Beego is a more complex type of framework that has quickly become one of the more popular Go frameworks for web projects.

Beego has a lot of features that can come in handy for a web service, despite the additional feature set that is largely intended for rendered web pages. The framework comes with its own sessions, routing, and cache modules, and also includes a live monitoring process that allows you to analyze your project dynamically.

> You can read more about Beego at `http://beego.me/` or download it using the `go get github.com/astaxie/beego` command.

Summary

In this final chapter, we looked at how to keep our web service as airtight as possible from common security issues and looked at solutions to mitigate issues if and when a breach happens.

As APIs scale both in popularity and scope, it is paramount to ensure that users and their data are safe.

We hope you have been (and will be) able to utilize these security best practices and tools to improve the overall reliability and speed of your application.

While our primary project—the social network—is by no means a complete or comprehensive project, we've broken down aspects of such a project to demonstrate routing, caching, authentication, display, performance, and security.

If you wish to continue expanding the project, feel free to augment, fork, or clone the example at `https://github.com/nkozyra/masteringwebservices`. We'd love to see the project continue to serve as a demonstration of features and best practices related to web services and APIs in Go.

Index