GO WEB DEVELOPMENT SUCCINCTLY

BY MARK LEWIN

SUCCINCTLY E-BOOK SERIES

Syncfusion®

Go Web Development Succinctly

By

Mark Lewin

Foreword by Daniel Jebaraj

Technical Reviewer: James McCaffrey
Copy Editor: John Elderkin
Acquisitions Coordinator: Hillary Bowling, online marketing manager, Syncfusion, Inc.
Proofreader: Jacqueline Bieringer, content producer, Syncfusion, Inc.

Table of Contents

The Story Behind the *Succinctly* Series of Books

Daniel Jebaraj, Vice President
Syncfusion, Inc.

Staying on the cutting edge

As many of you may know, Syncfusion is a provider of software components for the Microsoft platform. This puts us in the exciting but challenging position of always being on the cutting edge.

Whenever platforms or tools are shipping out of Microsoft, which seems to be about every other week these days, we have to educate ourselves, quickly.

Information is plentiful but harder to digest

In reality, this translates into a lot of book orders, blog searches, and Twitter scans.

While more information is becoming available on the Internet and more and more books are being published, even on topics that are relatively new, one aspect that continues to inhibit us is the inability to find concise technology overview books.

We are usually faced with two options: read several 500+ page books or scour the web for relevant blog posts and other articles. Just as everyone else who has a job to do and customers to serve, we find this quite frustrating.

The *Succinctly* series

This frustration translated into a deep desire to produce a series of concise technical books that would be targeted at developers working on the Microsoft platform.

We firmly believe, given the background knowledge such developers have, that most topics can be translated into books that are between 50 and 100 pages.

This is exactly what we resolved to accomplish with the *Succinctly* series. Isn't everything wonderful born out of a deep desire to change things for the better?

The best authors, the best content

Each author was carefully chosen from a pool of talented experts who shared our vision. The book you now hold in your hands, and the others available in this series, are a result of the authors' tireless work. You will find original content that is guaranteed to get you up and running in about the time it takes to drink a few cups of coffee.

Free forever

Syncfusion will be working to produce books on several topics. The books will always be free. Any updates we publish will also be free.

Free? What is the catch?

There is no catch here. Syncfusion has a vested interest in this effort.

As a component vendor, our unique claim has always been that we offer deeper and broader frameworks than anyone else on the market. Developer education greatly helps us market and sell against competing vendors who promise to “enable AJAX support with one click,” or “turn the moon to cheese!”

Let us know what you think

If you have any topics of interest, thoughts, or feedback, please feel free to send them to us at succinctly-series@syncfusion.com.

We sincerely hope you enjoy reading this book and that it helps you better understand the topic of study. Thank you for reading.

Please follow us on Twitter and “Like” us on Facebook to help us spread the word about the *Succinctly* series!

About the Author

Mark Lewin has been developing, teaching, and writing about software for more than 18 years. His main interests are web development in general and web mapping in particular. While working for ESRI, the world's largest GIS company, he acted as a consultant, trainer, and course author. He has been a frequent speaker at industry events, and he currently works with a wide variety of open-source mapping technologies and a handful of relevant JavaScript frameworks, including Node.js, Dojo, and JQuery.

Based in the U.K., Mark currently teaches ArcGIS Server/JavaScript development for Geospatial Training LLC and is the author of the *Google Maps API: Get Started* course for Pluralsight. By day, he writes MySQL courseware for Oracle.

Mark can be reached at mark@marklewin.com or on Twitter at @gisapps.

This is Mark's third e-book for Syncfusion. The first, *Leaflet.js Succinctly*, was published in March 2016. And the precursor to this e-book, *Go Succinctly*, will be published by the time you read this.

Acknowledgements

I'd like to thank my children for being little rays of sunshine, even on dark days. Words cannot express how I feel about you all.

And to my best friend, the "philosopher" David Peters. Everyone needs a friend to help them in the down times and keep them grounded in the good times, and David does that for me.

Chapter 1 Introduction

Who is this e-book for?

This book is for any developer who has a basic familiarity with the Go programming language and is interested in using Go to write web applications.

If you can write simple console applications with Go, you shouldn't have any problem understanding the contents of this short e-book.

If you're looking for a primer on the language itself, may I humbly recommend my book *Go Succinctly*, which will take you from zero to… well, if not exactly mastery, to an appreciation and understanding of the Go programming language. It also gives you links to other resources that will help you become familiar with Go.

Like all e-books in Syncfusion's *Succinctly* series, you can download Go Succinctly free of charge from the Syncfusion website.

Why use Go for web development?

Go is an excellent language for writing web applications, specifically for web services. In fact, it was designed specifically with the web in mind. After all, any modern programming language can hardly gain traction if it ignores the web.

So, which features of the Go programming language are particularly useful for web development? Here are a few:

Concurrency

Any decent-sized web server needs to run many thousands of tasks concurrently. Concurrency is complex and difficult in many languages because it is usually implemented as an afterthought. Concurrency, however, is built into the Go language. Indeed, concurrency is one of the main problems the Go language was designed to solve.

In Go, concurrency is achieved by using Goroutines, which are lightweight threads that allow developers to perform multiple operations asynchronously. These are incredibly useful in web applications. For example, when a user connects to your web server, you can simply spawn a Goroutine to handle any interactions with that client. It's very easy to do—merely prefix the function call with the **go** keyword. Better still, Goroutine scales incredibly well, and your Go web applications will purr along quite happily while servicing many thousands of users.

Modularity

Web applications, like many other modern applications, usually grow to include a lot of code. Keeping this code organized and efficient, so that it's easy to understand and maintain, is a challenge for today's developers. This is especially the case when several developers are working on the same application and each developer formats their code differently.

Go takes away many of these problems by imposing a specific method of structuring and formatting code. Functions, variables, constants, and type declarations are all expected to be in predictable places, and Go requires them all to be coded in a specific way.

If Go sounds draconian in this regard, consider this—what you lose in freedom of expression you gain in predictability; everything is where you expect it to be and formatted identically throughout.

Go makes it easy for you to satisfy its code formatting rules by providing you with the **`fmt`** package that you can build into your workflow to automatically Go-ify your code.

Compilation

Unlike many recent web development server-side languages, the Go programming language is compiled. This means that a problem such as a runtime error that might be difficult to track down is instead caught in the compilation step. Go's static typing system also helps you discover errors during development that might otherwise escape into production.

The `net/http` package

Go's **`net/http`** package is excellent and makes starting a web server while having full control of accepting requests and delivering responses very easy. Routing is handled by a multiplexer. You can either adopt the one in the standard library or select from several third-party options. In this book, we'll start by using Go's **`DefaultServeMux`**, then we'll consider a very capable alternative: **`gorilla/mux`**.

Setting up your development environment

I won't delve too deeply into setup here because I'm going to assume that you have some experience using Go. However, if you need to recap, I suggest referring to my e-book *Go Succinctly* or visiting the official Golang.org "Getting Started" page: https://golang.org/doc/install.

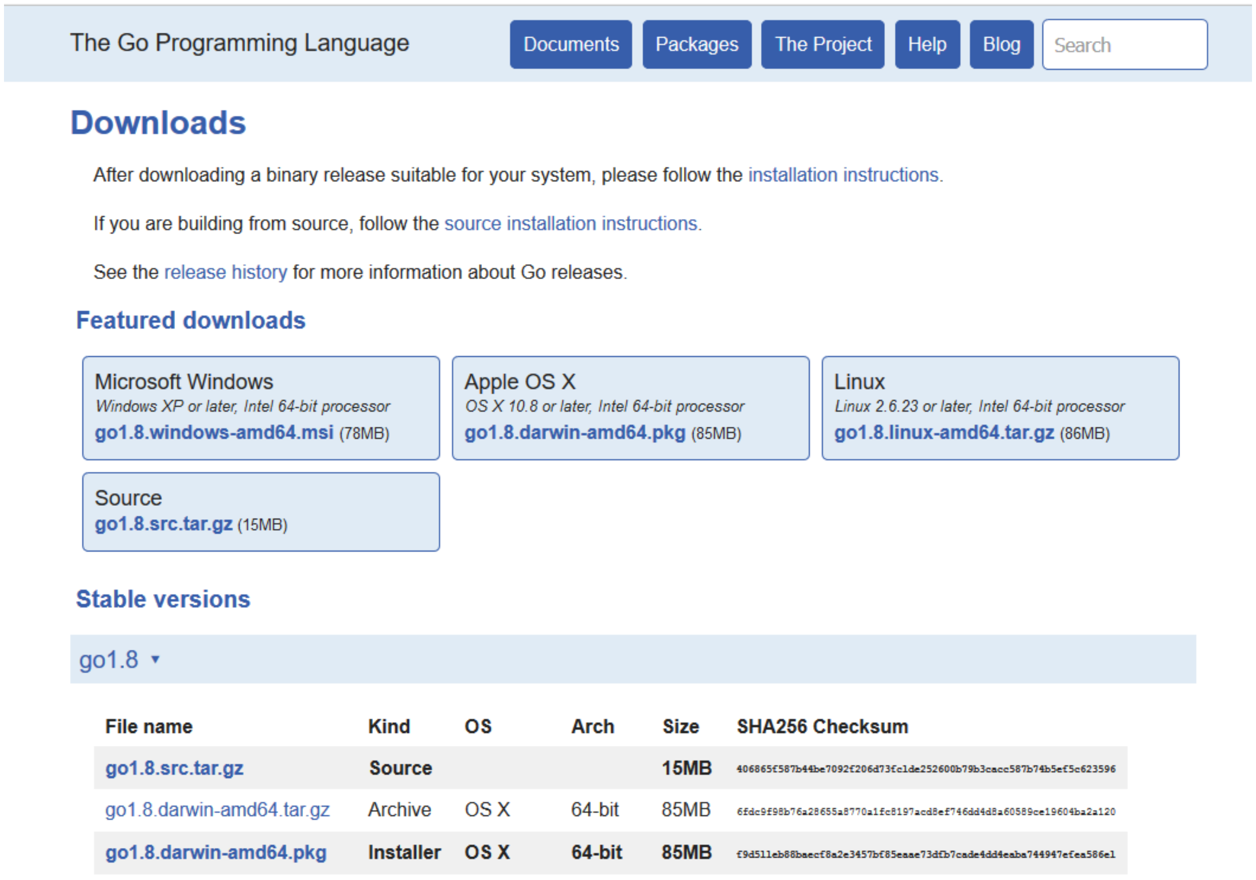
The steps involved in setting up your environment consist of:

- Installing the Go tools.
- Creating a workspace and setting the **`GOPATH`** environment variable.

Installing the Go tools

The exact steps you will need to follow depend on which platform you are using. Go binaries are available for Windows, Mac OS X, and *nix platforms.

You can download them here: https://golang.org/dl/, as shown in Figure 1.


The Go Programming Language

Documents · Packages · The Project · Help · Blog · Search

Downloads

After downloading a binary release suitable for your system, please follow the installation instructions.

If you are building from source, follow the source installation instructions.

See the release history for more information about Go releases.

Featured downloads

Microsoft Windows
Windows XP or later, Intel 64-bit processor
go1.8.windows-amd64.msi (78MB)

Apple OS X
OS X 10.8 or later, Intel 64-bit processor
go1.8.darwin-amd64.pkg (85MB)

Linux
Linux 2.6.23 or later, Intel 64-bit processor
go1.8.linux-amd64.tar.gz (86MB)

Source
go1.8.src.tar.gz (15MB)

Stable versions

go1.8 ▾

File name	Kind	OS	Arch	Size	SHA256 Checksum
go1.8.src.tar.gz	**Source**			**15MB**	406865f587b44be7092f206d73fc1de252600b79b3cacc587b74b5ef5c623596
go1.8.darwin-amd64.tar.gz	Archive	OS X	64-bit	85MB	6fdc9f98b76a28655a8770a1fc8197acd8ef746dd4d8a60589ce19604ba2a120
go1.8.darwin-amd64.pkg	**Installer**	**OS X**	**64-bit**	**85MB**	f9d511eb88baecf8a2e3457bf85eaae73dfb7cade4dd4eaba744947efea586e1

Figure 1: The Golang Downloads Webpage

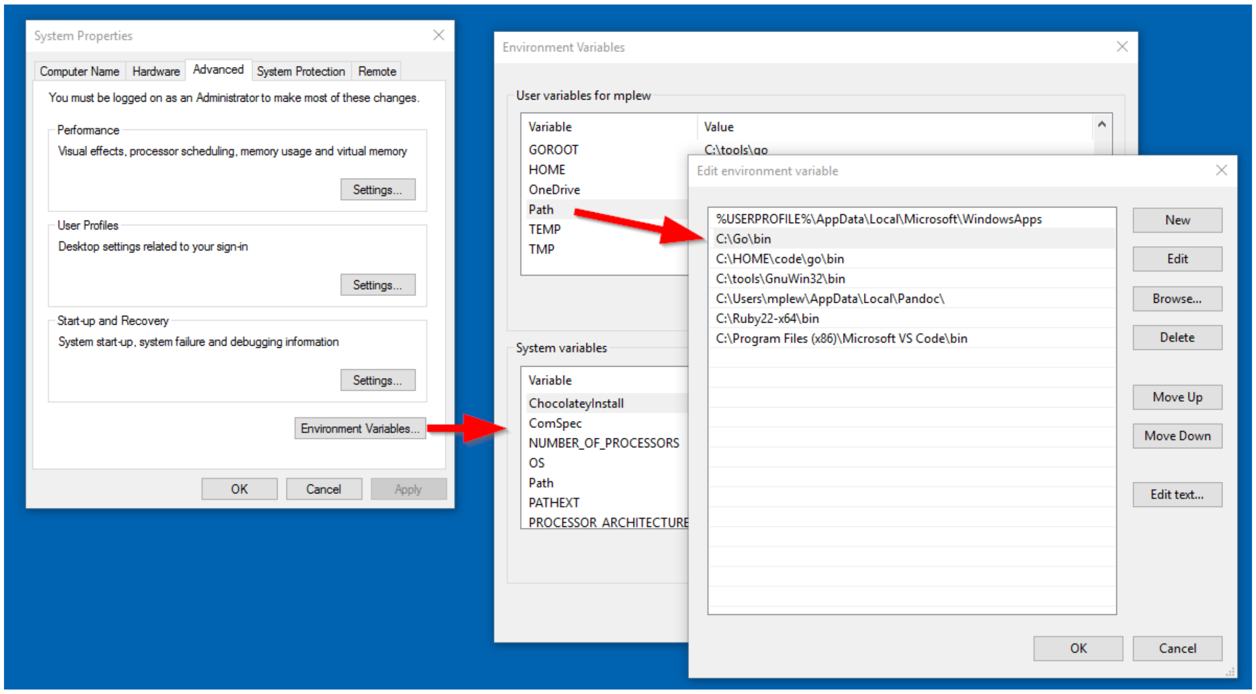
Tip: If you are upgrading to a later version of Go, you must uninstall the previous version first.

Windows

The easiest way to install the Go tools in Windows is to download the MSI installer, launch it, then follow the prompts. By default, Go installs everything in **`C:\Go`**, then it adds **`C:\Go\bin`** to your PATH environment variable.

Alternatively, if you would rather have more control over your environment variables, you can download the .zip file and extract it to a directory of your choice. In this case, you'll need to configure the environment variables yourself.

You can set environment variables through the Environment Variables button on the Advanced tab of the System control panel. Some versions of Windows provide this control panel through the Advanced System Settings option inside the System control panel.



You must be logged on as an Administrator to make most of these changes.

Performance
Visual effects, processor scheduling, memory usage and virtual memory

User Profiles
Desktop settings related to your sign-in

Start-up and Recovery
System start-up, system failure and debugging information

User variables for mplew

Variable	Value
GOROOT	C:\tools\go
HOME	
OneDrive	
Path	
TEMP	
TMP	

System variables

Variable
ChocolateyInstall
ComSpec
NUMBER_OF_PROCESSORS
OS
Path
PATHEXT
PROCESSOR_ARCHITECTURE

```
%USERPROFILE%\AppData\Local\Microsoft\WindowsApps
C:\Go\bin
C:\HOME\code\go\bin
C:\tools\GnuWin32\bin
C:\Users\mplew\AppData\Local\Pandoc\
C:\Ruby22-x64\bin
C:\Program Files (x86)\Microsoft VS Code\bin
```

Figure 2: Setting Go Environment Variables in Windows

You might need to close and re-open any command-line sessions in order for the changes to take effect.

Mac OS X

Download the **`.pkg`** file and follow the prompts. The package installs the Go distribution to **`/usr/local/go`** and adds **`/usr/local/go/bin`** to your **`PATH`**.

Linux, Unix, and FreeBSD

You'll notice that *nix-based systems expect a bit more from you. But as a *nix user, you are more than up to the challenge.

Download the **`.tar.gz`** file and extract it to **`/usr/local`** as the root user, or get it via **`sudo`**:

```
sudo tar -C /usr/local -xzf go$VERSION.$OS-$ARCH.tar.gz
```

You next need to add **`/usr/local/go/bin`** permanently to your **`PATH`**. Put the following line in your **`/etc/profile`** (for all users) or **`~/.profile`** (for only you) to make this a permanent thing:

```
export PATH=$PATH:/usr/local/go/bin
```

Custom installation locations

Those methods decide where Go will be installed. If you don't like to be bossed around that way, you must tell Go where to find itself by setting the **`GOROOT`** environment variable. If you're happy with the default location, then don't specify **`GOROOT`**. You'll only confuse things.

Creating a workspace

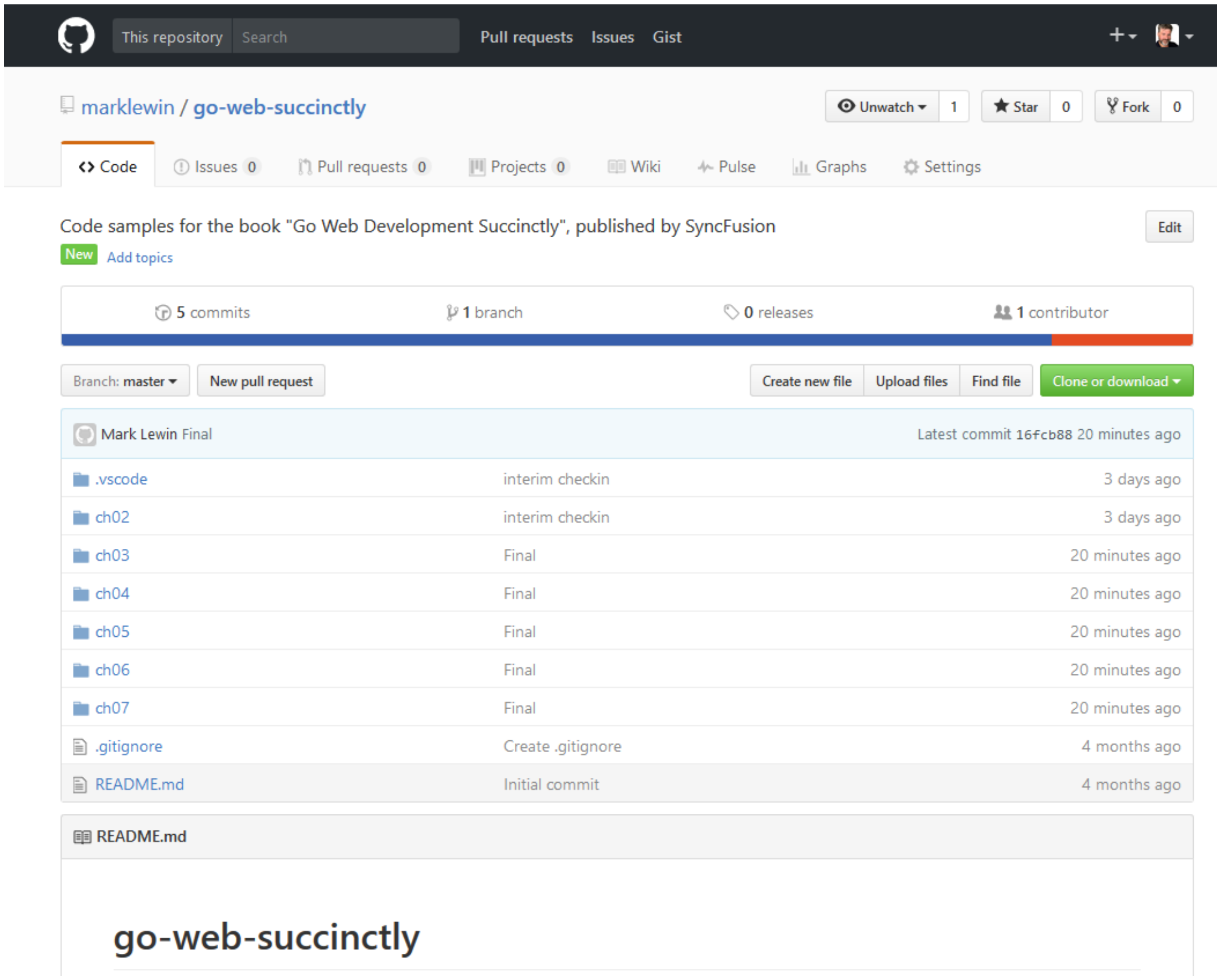
The final step is telling Go where any code you write, as well as any third-party libraries that you download with **`go get`**, will reside. Do this by setting the **`GOPATH`** environment variable using the same techniques described above.

Go will create **`bin`**, **`pkg`**, and **`src`** files in this location:

- **`bin`**: Contains executables
- **`pkg`**: Contains package objects
- **`src`**: Contains Go source code

Code examples

All code examples in this book can be found on GitHub at https://github.com/marklewin/go-web-succinctly.git.



Code samples for the book "Go Web Development Succinctly", published by SyncFusion

Mark Lewin Final — Latest commit 16fcb88 20 minutes ago

.vscode	interim checkin	3 days ago
ch02	interim checkin	3 days ago
ch03	Final	20 minutes ago
ch04	Final	20 minutes ago
ch05	Final	20 minutes ago
ch06	Final	20 minutes ago
ch07	Final	20 minutes ago
.gitignore	Create .gitignore	4 months ago
README.md	Initial commit	4 months ago

README.md

go-web-succinctly

Figure 3: The Github Repository for This E-Book's Code Samples

You can identify each sample by the chapter and topic it matches in this e-book.

You can either download each file individually or clone the repository by using the following commands:

```
$ git clone git://github.com/marklewin/go-web-succinctly.git

$ git pull origin
```

Chapter 2 Serving and Routing

Go's **`net`** package facilitates all network communications in Go programs, whether it's over HTTP, TCP/IP, WebSockets, or any other standard network protocol.

Of course, because this e-book addresses web development, we are primarily concerned with HTTP, which means the main subpackage we'll be using is **`net/http`**.

In this chapter, we'll look at the basic requirements of any Go web application—serving and routing—and how you can use **`net/http`** and complementary packages to implement them.

Go as a simple web server

Back in the old days, web servers did little more than serve up files that resided in a directory on that server. If that's all we want now, we can do something very similar in Go by using the **`net/http package`** in just a few lines of code.

Code Listing 1: Go as a Web Server

```
package main

import (
      "net/http"
)

func main() {
      http.ListenAndServe(":8999",
                              http.FileServer(http.Dir("/var/www")))
}
```

It's very basic, but our program fulfills the core responsibilities of any web server—namely, listening to a request and serving a response. What's more, because it doesn't have to worry about all the other issues that a more traditional web server must, the program is lightning fast.

In this example, we call the **`http.ListenAndServe()`** function to send all requests on port 8999 to an **`http.FileServer`** handler method, which in turn accepts the directory on the server that we want to serve files from. Easy!

If you run the program using **`go run <`*`program_name.go`*`>`** and enter **`http://localhost:8999/`** in your browser's address bar, followed by the name of a file that exists in **`/var/www`** on your server, you'll see that file's contents displayed—as in Figure 4.

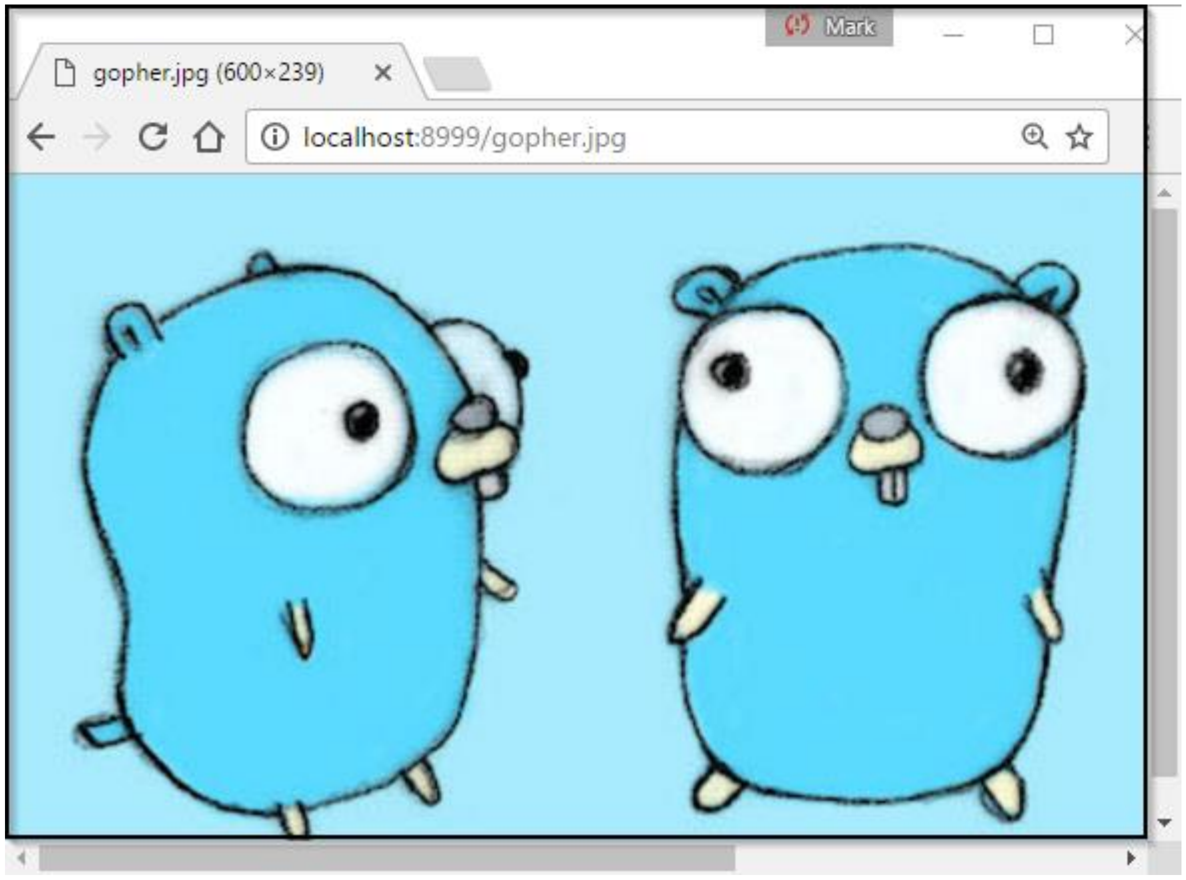


Figure 4: Go Serving Gophers

However, most modern web applications require a bit more than this from the server. And keep in mind, you're not always going to be sending static content. Increasingly, you will be called upon to generate content dynamically, perhaps from the contents of a database. In this case, a physical file location doesn't make much sense.

And suppose you have a more complex site structure in which your application's files are spread among various subdirectories? For example, **`www.mysite.com/about/aboutus.html`** and **`www.mysite.com/blog/blog.html`**? This approach won't work. So, you'll need better control over the URLs your application can accept. You can achieve this by using the **`net/http`** package's routing capabilities.

Simple serving and routing

Go relies on two main components to process HTTP requests—a multiplexor and handlers. A multiplexor (or "mux") is essentially an HTTP request router. In Go's **`net/http`** package, the multiplexor functionality is provided by **`ServeMux`** and the default serve mux is **`DefaultServeMux`**.

Intuitive, eh?

ServeMux compares incoming requests against a list of predefined URL paths, then calls the appropriate handler (a function that you define) for each path when there is a match.

Let's first have a look at the code, then we'll examine what's going on.

Code Listing 2: Basic Serving and Routing Using Net/http

```go
package main

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

func main() {
	http.HandleFunc("/", showInfo)
	http.HandleFunc("/site", serveFile)
	err := http.ListenAndServe(":8999", nil)
	if err != nil {
		log.Fatal("ListenAndServe: ", err)
	}
}

func showInfo(w http.ResponseWriter, r *http.Request) {
	fmt.Fprintln(w, "Current time: ", time.Now())
	fmt.Fprintln(w, "URL Path: ", html.EscapeString(r.URL.Path))
}

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

Notice that the **main()** function, which is the entry point into the program, sets up some server routes by using the **http.HandleFunc(*route, handler*)** method to map a URL route to a function that will respond to any requests coming in that match that route.

The program next calls the **http.ListenAndServe()** method, which starts an HTTP server with a specified address (in this case, the local machine on port 8999) and a mux. The mux in this instance is **nil**, which tells Go to use **DefaultServeMux**.

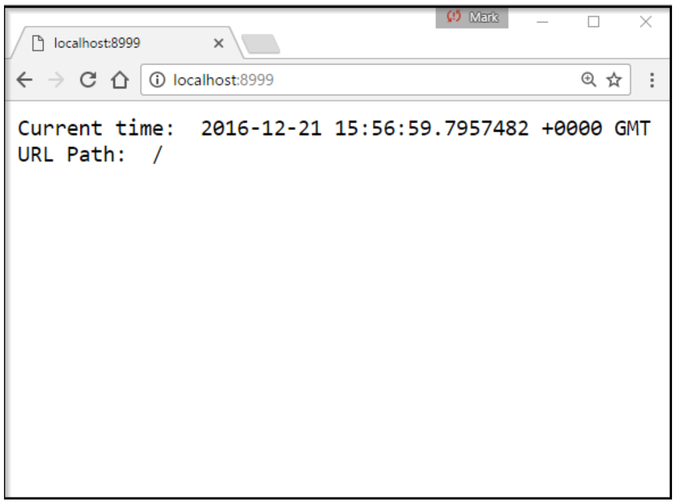
The handler functions do different things depending on whether the URL accessed is at the root of the site ("**/**") or at "**/site**". However, in both cases their method signatures must implement both **http.ResponseWriter** and **http.Request**. If a handler that does not implement both of these gets called via **http.HandleFunc()**, you'll see that Go will raise a compile-time error.

If the user visits **`/site`**, the **`http.ServeFile()`** method handles the request and returns the **`index.html`** page that resides in the same directory as the application. You can use **`http.ServeFile()`** to send any static file in the response.

If the user visits the root location, **`showInfo()`** gets called and will display the current time and whichever path that user entered beyond the root URL. The application extracts this information by using the **`URL.Path`** property of the **`http.Request`** object that gets passed to the handler. The response is generated by **`http.ResponseWriter`**.

Start the program by issuing **`go run <program_name.go>`** from your IDE or simply via the command line. Next, visit **`localhost:8999`** in your browser. Try various combinations to test its functionality.

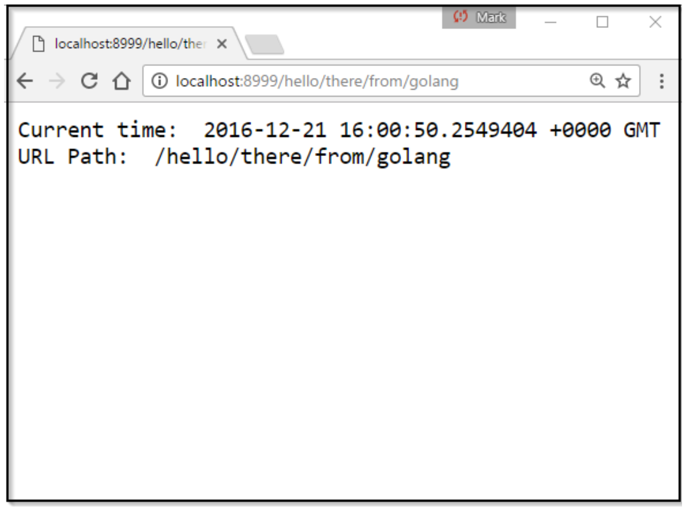
For example, entering **`http://localhost:8999`** results in the dynamically generated page in Figure 5.



```
Current time:  2016-12-21 15:56:59.7957482 +0000 GMT
URL Path:  /
```

Figure 5: Visiting the Root of the Web Application

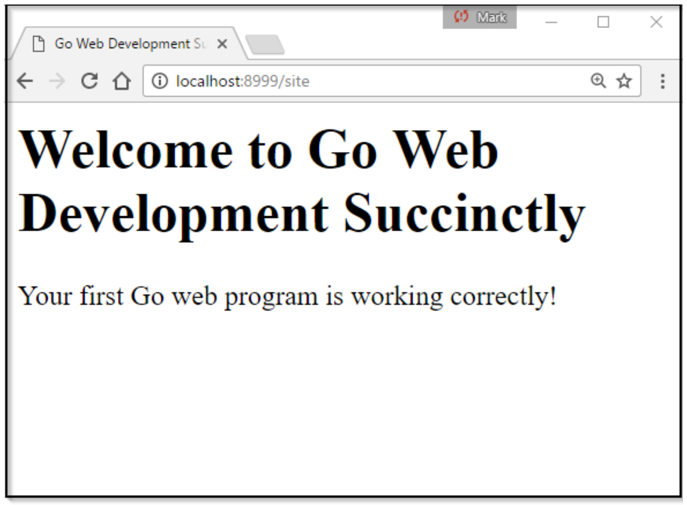
Entering any subroute (for example, **`localhost:8999/hello/there/from/golang`**) displays the time and date and the subroute entered, as we see in Figure 6.



```
Current time:  2016-12-21 16:00:50.2549404 +0000 GMT
URL Path:  /hello/there/from/golang
```

Figure 6: Visting a Subroute

Entering **`localhost:8999/site`** displays the **`index.html`** page.



Welcome to Go Web Development Succinctly

Your first Go web program is working correctly!

Figure 7: Visiting/Site

Let's expand our ability to serve static files in this example. Suppose that the URL in the request starts with **`/static/`** and we want to strip the **`/static`** part of the URL in order to look for the file referenced in the remaining path in the **`/var/www`** directory. We can use the **`StripPrefix`** function to achieve this, as we see in Code Listing 3.

Code Listing 3: Using StripPrefix

```
package main

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

func main() {
	http.HandleFunc("/", showInfo)
	files := http.FileServer(http.Dir("/var/www"))
	http.Handle("/site/", http.StripPrefix("/site/", files))
	err := http.ListenAndServe(":8999", nil)
	if err != nil {
		log.Fatal("ListenAndServe: ", err)
	}
}

func showInfo(w http.ResponseWriter, r *http.Request) {
	fmt.Fprintln(w, "Current time: ", time.Now())
	fmt.Fprintln(w, "URL Path: ", html.EscapeString(r.URL.Path))
}
```

Note that in this example we're not calling **`HandleFunc`**, but **`Handle`**. That's because the **`FileServer`** function returns its own handler that we can pass to the mux using **`Handle`** (instead of explicitly creating our own handler function).

So, you can see that it's straightforward to get a site up and running that is self-serving (no separate web server required) and lets you respond to some simple requests.

Middleware

In web development, middleware is code that sits between the web request and your route handler. Middleware consists of reusable bits of code you can use to perform tasks that must occur either before the handler is called or afterward.

The term "middleware" is often used with Go programming, but you might also see similar terms used with other web languages and technologies, such as "interceptor," "hooking," and "request filtering."

For example, you might want to check the status of a database connection or authenticate a user before routing the request. You might also want to compress the content of a response or limit the amount of times a specific handler is called—perhaps as part of some restriction that you place on users who access your web service free of charge.

Creating middleware is simply a matter of chaining handlers and handler functions, and it's something you will see—and use—a lot in Go web development.

The basic idea is that you pass a handler function—let's call it *f2*—into another handler function—let's call this one *f1*—as a parameter. Handler *f1* gets called when the route that triggers it is visited. *f1* does some work, then calls *f2*.

Of course, you could have *f1* call *f2* directly. However, this is not ideal because we typically want to achieve a clear separation of concerns, and therefore our handler code should really be limited to processing the request and not doing whatever *f2* is designed to do.

Here's the basic idea—instead of dealing with the response as part of your handler, you simply pass the next handler in the chain to it.

Functions that accept another handler function as a parameter and return a new one can perform tasks before or after the handler is called (or both before and after) and can even ultimately choose not to call the original handler at all (if that is your intention).

Consider the example in Code Listing 4.

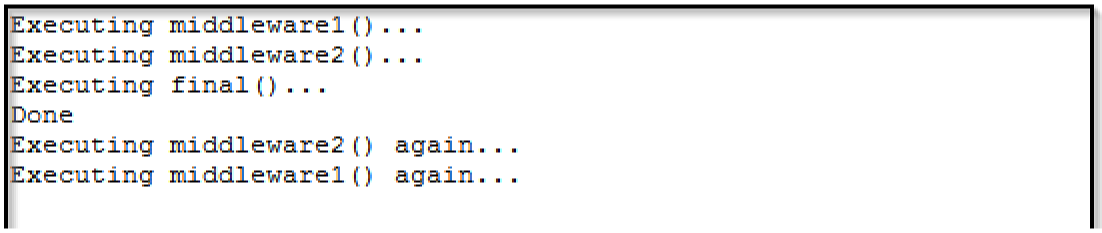
Code Listing 4: Middleware Example 1

```
package main

import (
	"fmt"
	"net/http"
)

func middleware1(next http.Handler) http.Handler {
	return http.HandlerFunc(func(w http.ResponseWriter, r
*http.Request) {
		fmt.Fprintln(w, "Executing middleware1()...")
		next.ServeHTTP(w, r)
		fmt.Fprintln(w, "Executing middleware1() again...")
	})
}

func middleware2(next http.Handler) http.Handler {
	return http.HandlerFunc(func(w http.ResponseWriter, r
*http.Request) {
		fmt.Fprintln(w, "Executing middleware2()...")
		if r.URL.Path != "/" {
			return
		}
		next.ServeHTTP(w, r)
		fmt.Fprintln(w, "Executing middleware2() again...")
	})
}

func final(w http.ResponseWriter, r *http.Request) {
	fmt.Fprintln(w, "Executing final()...")
	fmt.Fprintln(w, "Done")
}

func main() {
	finalHandler := http.HandlerFunc(final)
	http.Handle("/", middleware1(middleware2(finalHandler)))
	http.ListenAndServe(":8999", nil)
}
```

Look at the **`main()`** function. Here, we are intercepting requests to our root web directory with the **`middleware1`** handler. This handler accepts as a parameter another handler called **`middleware2`**. That in turn has the **`final`** handler as a parameter.

When someone visits our web application, it will call **`middleware1`**, which displays a message and, when it hits **`next.serveHTTP`**, executes the code in **`middleware2`**. The **`middleware2`** function in turn calls **`final`**, which executes and then returns control to **`middleware2`**, which completes its tasks and returns control to **`middleware1`**.

The output will look like Figure 8.



```
Executing middleware1()...
Executing middleware2()...
Executing final()...
Done
Executing middleware2() again...
Executing middleware1() again...
```

Figure 8: The Output of Middleware Example 1

Figure 8 offers a completely artificial example, but it serves to illustrate how much control middleware can give you.

Go often uses middleware internally. For example, many functions in the **`net/http`** package, such as **`StripPrefix`**, are textbook examples of what middleware is—they wrap your handler and perform additional operations on requests or responses.

The concept of middleware can be somewhat difficult to understand, so Code Listing 5 shows another example.

Code Listing 5: Middleware Example 2

```
package main

import (
	"net/http"
)

type AfterMiddleware struct {
	handler http.Handler
}
```

```
func (a *AfterMiddleware) ServeHTTP(w http.ResponseWriter, r
*http.Request) {
	a.handler.ServeHTTP(w, r)
	w.Write([]byte(" +++ Hello from middleware! +++ "))
}

func myHandler(w http.ResponseWriter, r *http.Request) {
	w.Write([]byte(" *** Hello from myHandler! *** "))
}

func main() {
	mid := &AfterMiddleware{http.HandlerFunc(myHandler)}

	println("Listening on port 8999")
	http.ListenAndServe(":8999", mid)
}
```

In this bit of code, we want our middleware to execute when the **`myHandler`** function writes the response body—and to append some data to it.

First, we create a type for the middleware that we call **`AfterMiddleware`**. This consists of a single field—the **`http.Handler`** we want our middleware to respond to.

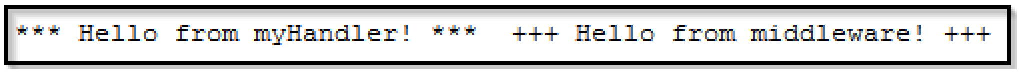
The **`http.Handler`** interface requires only that we implement the **`ServeHTTP`** interface:

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

We do this as follows:

```
func (a *AfterMiddleware) ServeHTTP(w http.ResponseWriter, r *http.Request) {
      a.handler.ServeHTTP(w, r)
      w.Write([]byte(" +++ Hello from middleware! +++ "))
}
```

Now, the response consists of whatever **`myHandler`** wrote and is followed by the output of our middleware in Figure 9.



```
*** Hello from myHandler! ***  +++ Hello from middleware! +++
```

Figure 9: The Output of Middleware Example 2

We cannot cover every aspect of middleware here, but it is a very powerful feature that is well worth investigating. This article and video lecture by Mat Ryer is an excellent introduction to the topic.

We'll revisit the concept of middleware in Chapter 7, when we create some eminently more useful logging middleware.

More advanced serving and routing with the Gorilla Web Toolkit

The routing functionality offered by Go's built-in **`net/http`** package only gets you so far. If the range of possible URLs is complex and if, for example, you want to be able to use regular expressions or variables to match URLs, you will probably want to consider a third-party solution.

The Gorilla Web Toolkit is one such solution. Gorilla consists of 22 packages, including:

- **`gorilla/context`** stores global request variables.
- **`gorilla/mux`** is a powerful URL router and dispatcher.
- **`gorilla/reverse`** produces reversible regular expressions for regexp-based muxes.
- **`gorilla/rpc`** implements RPC over HTTP with codec for JSON-RPC.
- **`gorilla/schema`** converts form values to a struct.
- **`gorilla/securecookie`** encodes and decodes authenticated and optionally encrypted cookie values.
- **`gorilla/sessions`** saves cookie and filesystem sessions and allows custom session back ends.
- **`gorilla/websocket`** implements the WebSocket protocol defined in RFC 6455.

As you can see from the list of packages, there is more to Gorilla than merely an alternative mux for Go routing and serving. In fact, it provides a whole range of different tools to assist you in your web development efforts.

But here, we're interested in the mux. The **`gorilla/mux`** module implements the **`http.Handler`** interface so that it is compatible with the standard Golang. **`http.serveMux`**.

In addition, the **`gorilla/mux`** module gives you the ability to:

- Match requests based on URL host, path, path prefix, schemes, header and query values, HTTP methods, or custom matchers that you define.
- Register URLs so that you can build or "reverse" them, thereby maintaining references to resources.
- Use "subrouters"—nested routes that are only tested if their parent routes match. In this way, you can define "groups" of routes that all have something in common, such as a host or a path prefix. This optimizes request matching by undertaking some tests only if they are appropriate to the group (rather than executing tests against all incoming requests).

Installing and referencing the `gorilla/mux` package

If you are only interested in the more advanced routing capability offered by Gorilla, you can install **`gorilla/mux`** in your **`GOPATH`** by using **`go get`**. For example, assuming that you have **`git`** installed:

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

Next, you must import it into your application, like so:

```
package main

import (
	github.com/gorilla/mux
	...
)
```

Using gorilla/mux

For a bit of fun, let's use **gorilla/mux** to create a handler that matches incoming requests for product IDs based on a regular expression. If the product ID is a single digit long, we have a match and can route to the appropriate page. If not, we route to an error page.

We'll first need to import the **gorilla/mux** module, which we do in the normal way:

```
import(
	. . .
	github.com/gorilla/mux
)
```

Next, we'll need to tell Go to use **gorilla/mux** instead of its own **DefaultServeMux:**

```
func main() {
	router = mux.NewRouter()
}
```

When we've done that, we can use the handler functions we are familiar with, but in the context of **router** and with the extra capabilities **gorilla/mux** provides, such as searching for a URL parameter with a regular expression:

```
router.HandleFunc("/product/{id:[0-9]+}", pageHandler)
```

That line of code creates a handler function that attempts to match a URL that consists of **/product/** with a number from zero to nine, inclusive, which it refers to as **id**. If there is a match, it calls the **pageHandler** function.

In our **pageHandler** function, we need a way to examine the exact string that **HandleFunc** matched. We can use Gorilla's **mux.Vars** function to do this, passing in the request as a single parameter and getting back a map of route variables. We can reference the route variable we're interested in by its name—**id**.

Having retrieved the product ID, we can next check for a matching HTML page. Go's **os** module provides the **Stat** and **IsNotExist** functions to help us with that. If the file exists on the server, we use **http.ServeFile** in order to send it to the browser. And if it doesn't exist there, we'll return another page telling the user that the product is invalid.

Code Listing 6 shows the full code.

Code Listing 6: Serving and Routing Using gorilla/mux

```
package main

import (
	"log"
	"net/http"
	"os"

	"github.com/gorilla/mux"
)

func pageHandler(w http.ResponseWriter, r *http.Request) {
	vars := mux.Vars(r)
	productID := vars["id"]
	log.Printf("Product ID:%v\n", productID)

	fileName := productID + ".html"

	if _, err := os.Stat(fileName); os.IsNotExist(err) {
		log.Printf("no such product")
		fileName = "invalid.html"
	}

	http.ServeFile(w, r, fileName)
}

func main() {
	router := mux.NewRouter()
	router.HandleFunc("/product/{id:[0-9]+}", pageHandler)
	http.Handle("/", router)
	http.ListenAndServe(":8999", nil)
}
```

This only scratches the surface of what **`gorilla/mux`** can do. For example, you can match against:

Path prefixes:

`router.PathPrefix("/products/")`

HTTP methods:

`router.Methods("GET", "POST")`

URL schemes:

`router.Schemes("https")`

Header values (for example, was the request an AJAX request?):

```
router.Headers("X-Requested-With", "XMLHttpRequest")
```

Query values:

```
router.Queries("key", "value")
```

Custom matching functions that you define:

```
router.MatcherFunc(func(r *http.Request, match *RouteMatch) bool {
      // do something
})
```

You can also combine multiple matchers in a single route by chaining function calls:

```
router.HandleFunc("/products", productHandler).
  Host("www.example.com").
  Methods("GET").
  Schemes("https")
```

Or, you can use subroutes to group multiple routes together. For example, the following code looks for a host name of www.example.com, but it will check the subroutes only if it gets a match on the host name:

```
router := mux.NewRouter()
subrouter := route.Host("www.example.com").Subrouter()

// Register the subroutes
subrouter.HandleFunc("/products/", AllProductsHandler)
subrouter.HandleFunc("/products/{name}", ProductHandler)
subrouter.HandleFunc("/reviews/{category}/{id:[0-9]+}"), ReviewsHandler)
```

The **gorilla/mux** package is a very capable alternative to **DefaultServeMux** and well worth investigating. You can find out more at http://www.gorillatoolkit.org/pkg/mux.

Of course, with Go being such a vibrant ecosystem, there are many other choices if you don't like Gorilla. I prefer Gorilla because it's easy to get my head around, and it's always been able to do whatever I ask it to do.

Other popular third-party routers include:

- **httprouter:** Lightning fast, but less capable than Gorilla. For example, you cannot include regular expressions in your routes.
- **Httptreemux:** A fast, flexible, tree-based HTTP router, inspired by **httprouter**.
- **Pat:** Simple to use and quite popular. If you're a Ruby-ist familiar with Sinatra and Rails, you'll find Pat's approach very familiar.

Returning errors

Things don't always work as we intend. Requests get made for pages that no longer exist, things get moved from one place to another, and sometimes the connection drops.

The HTTP protocol defines 61 different status codes so that we can determine whether or not a request was successful.

Here are a few of the most common ones:

- **`200 OK`**: Hooray! All is well.
- **`404 Not found`**: Whatever resource you're looking for, you won't find it here.
- **`301 Moved Permanently`**: Used as a permanent redirect to another page.
- **`301 Moved Temporarily`**: Used as a temporary redirect to another page.
- **`500 Internal Server Error`**: Something unexpected has gone wrong. This is a "catch all."

The **`net/http`** package provides a function called **`Error`** that you can use to handle error status codes. Your job as a developer is to pick one that makes sense for the type of error you are reporting.

You can raise an error by passing the **`ResponseWriter`**, a string message, and the status code. For example, this raises a **`404 Not Found`** error:

```
http.HandleFunc("/", func(w http.ResponseWriter, r *http.Request) {
    http.Error(w, "Something has gone wrong", 500)
})
```

However, it's better practice to use the various helper functions that **`net/http`** provides for this purpose rather than the generic **`http.Error()`** function.

For example:

```
// Return 404 Not Found
http.NotFound(w, req)

// Return 301 Permanently Moved
http.Redirect(w, req, “http://somewhereelse.com”, 301)

// Return 302 Temporarily Moved
http.Redirect(w, req “http://temporarylocation”, 302)
```

Chapter 3 Accessing Data

In the previous chapter, we saw how Go can accept requests from different URLs and route them to an appropriate handler. So far, however, all of our responses to those requests have served only static content, which is not enough for today's web applications. In this chapter, we'll see how we can connect our Go web applications to a data source and serve dynamic data instead.

Getting a driver for your database

Go allows you to use any database for which a driver is available. A definitive list of drivers can be found at https://github.com/golang/go/wiki/SQLDrivers.

Unless you're using something quite exotic, you should be able to find a driver for your database in the wiki.

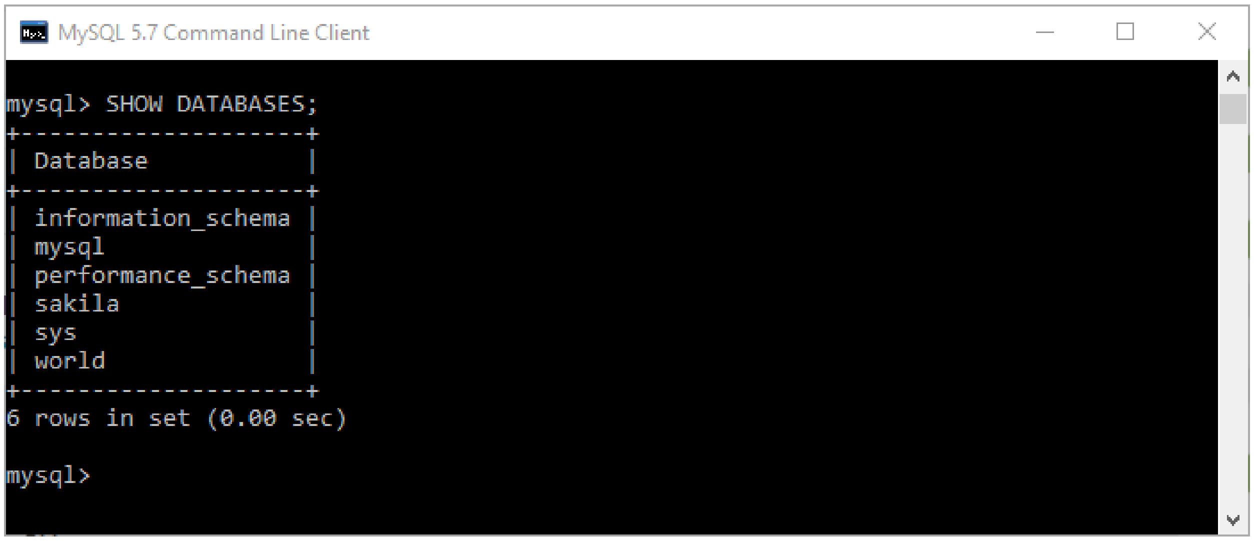
MySQL is a fantastic open-source database and, although it has been around for quite a while, it continues to be extremely popular, so we'll use that for our examples in this chapter. However, whichever database you choose, the steps you need to take will be similar, if not identical, to those shown here.

You can download the free community edition of MySQL Server from http://www.mysql.com. Follow the installation instructions for your platform. If you want to follow along with the examples in this chapter, you should also install the sample "world" database. There's an option in the Windows installer for this, but with Mac and Linux you must download and install it manually from https://dev.mysql.com/doc/index-other.html.

Simply connect to the MySQL server using the command-line client (or MySQL Workbench, which you can also install if you prefer a GUI environment), then execute the following statement:

```
mysql> SOURCE < /path/to/world.sql
```

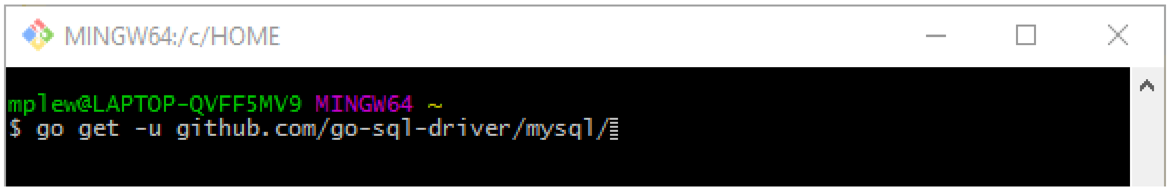
You can verify that the "world" database is installed by issuing the command in Figure 10 at the **mysql>** prompt.



```
mysql> SHOW DATABASES;
+--------------------+
| Database           |
+--------------------+
| information_schema |
| mysql              |
| performance_schema |
| sakila             |
| sys                |
| world              |
+--------------------+
6 rows in set (0.00 sec)

mysql>
```

Figure 10: Verifying "World" Database Installation

When you've installed the database server, you need to install the driver for it. You can do this by using **`go get`**. I'm using the **`go-sql-driver`** in Figure 11.



```
mplew@LAPTOP-QVFF5MV9 MINGW64 ~
$ go get -u github.com/go-sql-driver/mysql/
```

Figure 11: Installing the MySQL Driver for Go

When we've imported the driver specific to the database server, we'll next need to reference both it and the generic **`database/sql`** package in our **`import`** statement. Note that we are preceding the driver import with an underscore (_). This tells the Go compiler that the package we are referencing is complementary to another package—in this instance, the **`database/sql`** package. We're using it only for its initialization capability—what Go calls "side effects."

```
import (
	"database/sql"
	"log"

	_ "github.com/go-sql-driver/mysql"
)
```

Next, we need to specify the server host, port number, initial database, user name, and password that we'll use to build the connection string.

```
db, err := sql.Open("mysql", "root:password@tcp(127.0.0.1:3306)/world")
```

There are three tables in the "world" database—Country, City, and CountryCode, the last of which links cities to countries. It's a good idea to create types for each of the database tables in your application so that it will be easier to implement changes in the schema later.

Here is our Go struct for rows in the City table:

```
type City struct {
      Name        string
      CountryCode string
      District    string
      Population  uint32
}
```

Next, let's have a variable in which we store a handle to our database when we're connected:

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

Finally, we're ready to connect to the database. We build up the connection string by serializing the server host, port number, initial database, and the user name and password strings.

Next, we attempt to connect by calling the **`database/sql.Open`** method, passing in the type of driver and the connection string:

```
dbConn := fmt.Sprintf("%s:%s@tcp(%s)/%s", dbUser, dbPass, dbHost, dbDatabase)
db, err := sql.Open("mysql", dbConn)
```

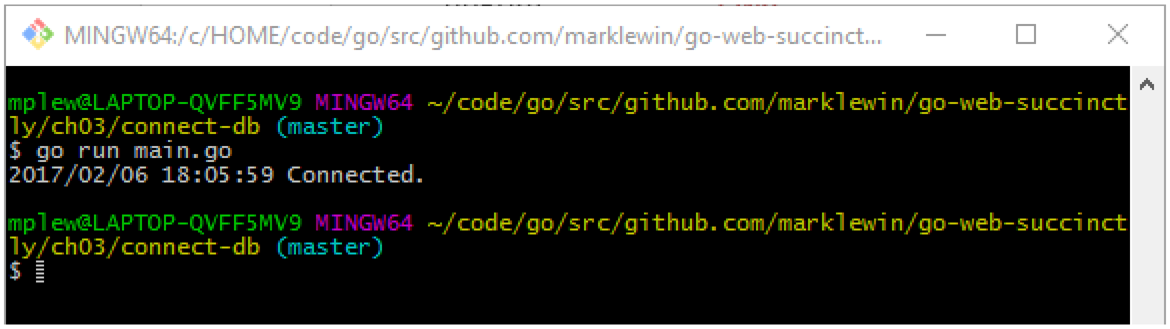
This returns an instance of **`sql.DB`** (about which we will have more to say later).

The full code is here in Code Listing 7.

Code Listing 7: Connecting to the Database

```
package main

import (
      "database/sql"
      "log"

      _ "github.com/go-sql-driver/mysql"
)

var database *sql.DB

func main() {
      db, err := sql.Open("mysql",
"root:password@tcp(127.0.0.1:3306)/world")
      if err != nil {
            log.Println("Could not connect!")
      }
      database = db
      log.Println("Connected.")
}
```

If everything goes to plan, we should receive a message at the terminal prompt telling us that we are now connected to the MySQL server.



```
MINGW64:/c/HOME/code/go/src/github.com/marklewin/go-web-succinct...
mplew@LAPTOP-QVFF5MV9 MINGW64 ~/code/go/src/github.com/marklewin/go-web-succinct
ly/ch03/connect-db (master)
$ go run main.go
2017/02/06 18:05:59 Connected.

mplew@LAPTOP-QVFF5MV9 MINGW64 ~/code/go/src/github.com/marklewin/go-web-succinct
ly/ch03/connect-db (master)
$
```

Figure 12: Verifying Connection to the Database

Now that we can connect to the “world” database, we can execute queries against it and display the results to our users. But first, let’s examine **`sql.DB`**.

sql.DB

Note from the previous code example that we are using the **`database`** variable that contains **`sql.DB`** to access the database.

The **`sql.DB`** is not the database connection—that’s stored in **`dbConn`**—instead, it’s the **`database/sql`** package’s abstraction of the database. In our example, it refers to the MySQL database, but because we can use **`database/sql`** to interface with many different data sources, it could just as easily be a local file or some sort of in-process data store such as **`memcache`**.

`sql.DB` takes care of a lot of things on your behalf, including the opening, closing, and pooling of database connections.

Don’t keep opening and closing databases unless your application requires it. The **`sql.DB`** object is designed to be long-lived, and all of your interaction with the database will come via **`sql.DB`**. However, you must be certain to close connections by using **`sql.DB.Close`** when you are finished with them so that they can be returned to the pool.

Tip: If you need to give short-lived function access to `sql.DB`, pass it to the function as a parameter instead of creating a new connection within the function.

We’re getting ahead of ourselves. We haven’t done anything with our database yet.

Let’s rectify that now.

Retrieving data from the database

The following code queries the “world” database’s City table based on whatever the user enters in the browser’s address bar after http://localhost:8999/.

Code Listing 8: Querying the Database

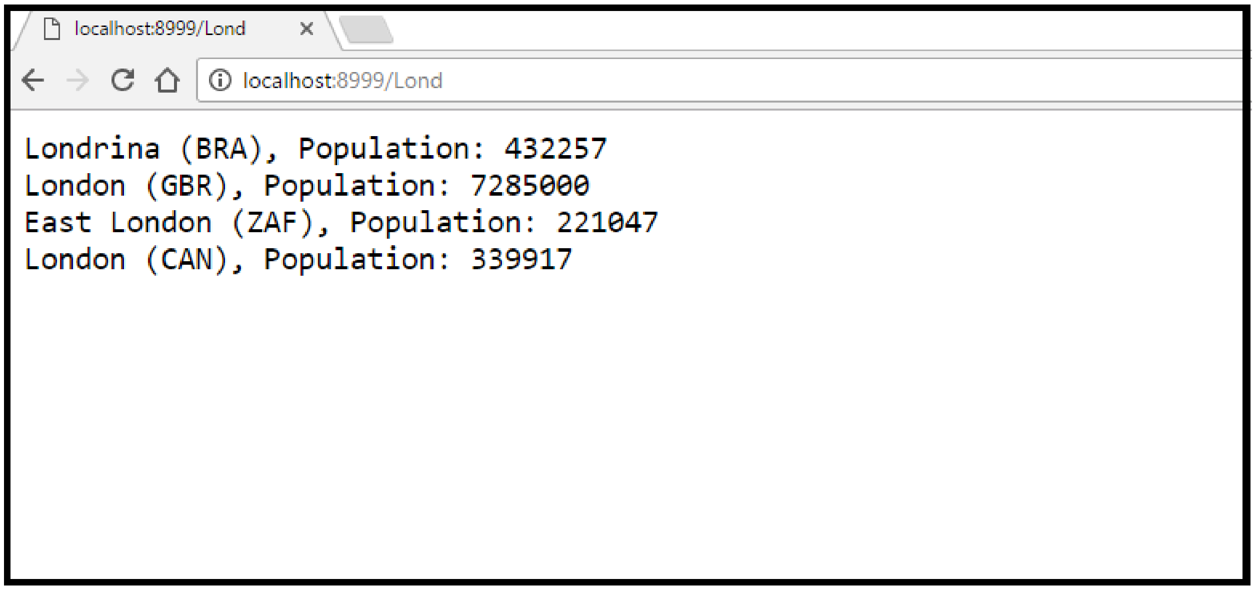
```go
package main

import (
	"database/sql"
	"fmt"
	"log"
	"net/http"

	_ "github.com/go-sql-driver/mysql"
)

type City struct {
	Name        string
	CountryCode string
	Population  int
}

var database *sql.DB

func main() {

	db, err := sql.Open("mysql",
	                         "root:password@tcp(127.0.0.1:3306)/world")
	if err != nil {
		log.Println("Could not connect!")
	}
	database = db
	log.Println("Connected.")

	http.HandleFunc("/", showCity)
	http.ListenAndServe(":8999", nil)
}

func showCity(w http.ResponseWriter, r *http.Request) {
	city := City{}
	queryParam := "%" + r.URL.Path[1:] + "%"
	rows, err := database.Query("SELECT Name, CountryCode,
	         Population FROM city WHERE Name LIKE ?", queryParam)
	if err != nil {
		log.Fatal(err)
	}
	defer rows.Close()
```

```
        for rows.Next() {
                err := rows.Scan(&city.Name, &city.CountryCode,
                                                        &city.Population)
                if err != nil {
                        log.Fatal(err)
                }
                fmt.Fprintf(w, "%s (%s), Population: %d \n", city.Name,
                                            city.CountryCode, city.Population)
        }
        err = rows.Err()
        if err != nil {
                log.Fatal(err)
        }
}
```

What we're interested in here resides in the **`showCity`** function that gets called when the user visits the root of the web application.

In order to query the database, we call the **`Query`** method in **`sql.DB`** and pass in the required SQL. The nice thing about **`Query`** (and its counterpart, **`QueryRow`**, which retrieves only a single row) is that it is parameterized, which helps protect against SQL injection.

The query returns a collection of rows that we iterate over by using the **`Next`** method. For each row, we execute **`Scan`** to map the columns in the row to the **`City`** type we declared earlier. Next, it's a simple matter of displaying the results to the user, as in Figure 13.



```
Londrina (BRA), Population: 432257
London (GBR), Population: 7285000
East London (ZAF), Population: 221047
London (CAN), Population: 339917
```

Figure 13: Displaying the Query Results

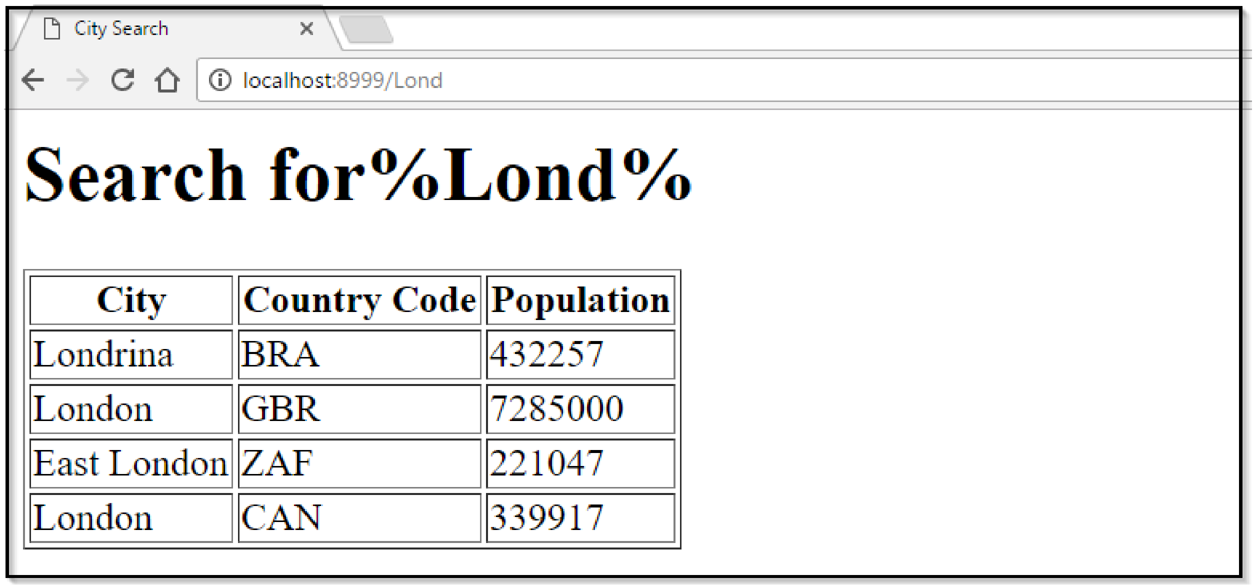
Tidying up the output

All we have now is some rather ugly text being dumped out to the browser. Let’s make it a little bit more pleasant to look at by displaying the query results in an HTML table. We do this simply by rewriting the code in our handler, as in Code Listing 9.

Code Listing 9: Using HTML to Format the Query Results

```
func showCity(w http.ResponseWriter, r *http.Request) {
	city := City{}
	queryParam := "%" + r.URL.Path[1:] + "%"
	rows, err := database.Query("SELECT Name, CountryCode, Population
FROM city WHERE Name LIKE ?", queryParam)
	if err != nil {
		log.Fatal(err)
	}
	defer rows.Close()

	html := "<html><head><title>City
Search</title></head><body><h1>Search for" + queryParam + "</h1><table
border='1'><tr><th>City</th><th>Country
Code</th><th>Population</th></tr>"

	for rows.Next() {
		err := rows.Scan(&city.Name, &city.CountryCode,
						&city.Population)
		if err != nil {
			log.Fatal(err)
		}
		html +=
fmt.Sprintf("<tr><td>%s</td><td>%s</td><td>%d</td></tr>", city.Name,
city.CountryCode, city.Population)
	}
	err = rows.Err()
	if err != nil {
		log.Fatal(err)
	} else {
		html += "</table></body></html>"
		fmt.Fprintln(w, html)
	}
}
```

When we execute it, we get Figure 14.



Search for%Lond%

City	Country Code	Population
Londrina	BRA	432257
London	GBR	7285000
East London	ZAF	221047
London	CAN	339917

Figure 14: Displaying the Query Results in an HTML Table

Not exactly pretty, but a slight improvement. Of course, we can style this to our hearts' content with some judicious use of CSS.

But wait a minute—at what expense does this come to our code? Our nice, neat program is now a mess of inline HTML. If it becomes more complex over time, we'd find it very difficult to maintain. What's more, each time we want to make a trivial change, we would have to recompile our code.

We have now committed one of web development's great cardinal sins—mixing logic and presentation in our source code. Thankfully, there's a much better way to format the display. We'll examine that in the next chapter.

Chapter 4 Templates

In the previous chapter, we connected to a data source, then displayed the results of queries to the user in HTML format. This HTML code was embedded directly within our source code, which is not desirable (except for very simple scenarios) because it makes the application very difficult to maintain.

Go comes to the rescue with a rather good templating engine. Not only does this enable us to separate our program code from the presentation markup, but it also provides some logical constructs such as loops, variables, and even functions that allow us to offload presentation logic to the template.

Introducing templates

First, let's define a template.

Go template functionality is provided by the **`template`** package, which includes a number of methods, including **`Parse`** and **`ParseFile`**, for loading a template from a string or file, then **`Execute`** for merging the specific content from our application with the more general content provided by the template itself.

Note that templates typically exist within their own files, but this is not always the case. We can use template functionality directly within our program logic by encoding the template as a string, then using the **`Parse`** method to read from it. This is only a slight improvement on encoding everything as HTML, so in these examples we'll use separate template files and **`ParseFile`** to read from them.

First, our template. As you can see, this is simply an HTML file with some special syntax so that our program knows what to insert and where to insert it.

Code Listing 10: Our HTML File

```
<html>
<head>
<title>Hello!</title>
</head>
<body>
  <h1>A warm hello to....</h1>
  <p>
    {{.Name}}
  </p>
</body>
</html>
```

Everything should look familiar except the **{{.Name}}** entry. To Go templates, that is a variable. It's contained within double curly braces, and its name (**Name**) is preceded by a dot.

The dot signifies the scope or context of the variable. In our case, the **Name** variable refers to the **Name** field of a **Person** struct, which will become apparent when you see the definition for it that we'll use in our program logic:

```
type Person struct {
	Name string
}
```

We can pass a **Person** struct to the template variable, and it will pull out the **Name** field and replace the variable definition in the template with it.

Note that any fields you refer to in the template must be exported. That is, they must begin with a capital letter. Unexported fields cause issues, as we'll see in a bit.

Code Listing 11 shows the full program code.

Code Listing 11: Using the Template

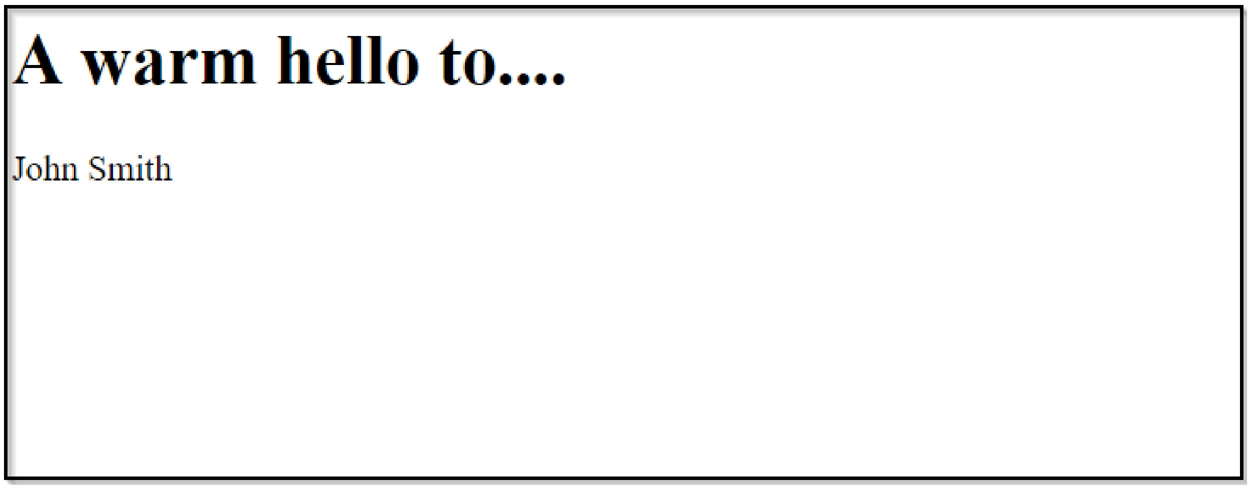
```
package main

import (
	"log"
	"net/http"
	"text/template"
)

type Person struct {
	Name string
}

func main() {
	http.HandleFunc("/", handler)
	err := http.ListenAndServe(":8999", nil)
	if err != nil {
		log.Fatal("ListenAndServe: ", err)
	}
}

func handler(w http.ResponseWriter, r *http.Request) {
	p := Person{Name: "John Smith"}
	t, _ := template.ParseFiles("hello.html")

	t.Execute(w, p)
}
```

The only code of note here is in the **handler** function that is invoked when a user visits the root of our web application at http://localhost:8999.

Handler creates a variable **`p`** of type **`Person`** and assigns “John Smith” to its **`Name`** field. We then use **`template.ParseFiles`**, passing in the name of the template file and assigning the results to variable **`t`**.

Having parsed the template file, we next need to perform the merge, which we do by calling **`Execute`**, which passes the **`http.ResponseWriter`** and the data item we want to merge with the template as parameters.

When we launch the application in a browser, we get Figure 15.


A warm hello to....

John Smith

Figure 15: The Output of Our Simple Template

Reworking the data access application using a template

Let’s put our newfound knowledge of Go templates to work. First, we need to understand one of the many capabilities of a template—the ability to iterate over a set of results, then apply the appropriate formatting to each of them.

In order to achieve this, we must use the **`{{range}} {{end}}`** construct. We’ll use the following template to build out our table. Everything between **`{{range}}`** and **`{{end}}`** will be repeated once for every member of the object we pass into it.

Code Listing 12: The Template for Our “Cities” Database Application

```
<html>
<head>
<title>City Search</title>
</head>
<body>
  <h1>Search Results</h1>
  <table border='1'>
    <tr>
      <th>City</th>
      <th>Country Code</th>
      <th>Population</th>
    </tr>
    {{range .}}
      <tr>
        <td>{{.Name}}</td>
        <td>{{.CountryCode}}</td>
        <td>{{.Population}}</td>
      </tr>
    {{end}}
  </table>
</body>
</html>
```

See the dot notation that appears just after the range keyword? That means “any object.”

Something important to note about using ranges in Go templates—they will only accept a single object. “What?” I hear you say. “Why would something that is supposed to iterate through a set of values only accept one?”

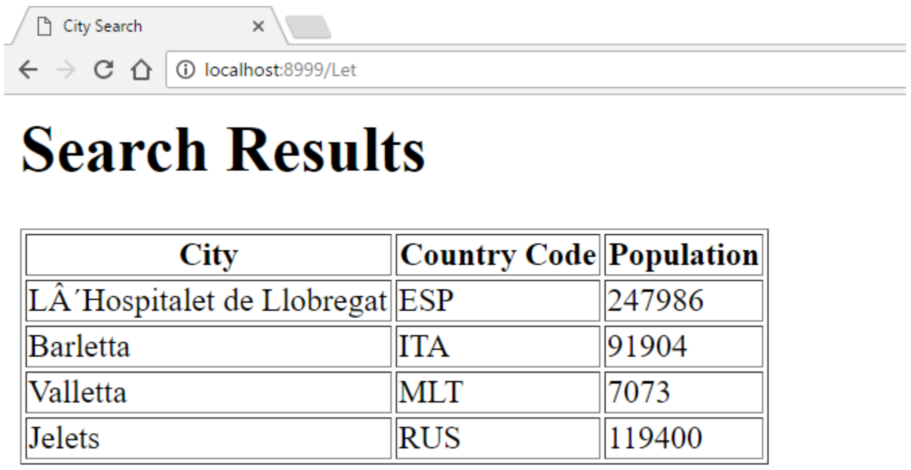
And you’d be right. But the range allows us to pass in an array or slice of values, then it will dig into that object to pull out its members.

So, in our database application, we simply need to do a bit of refactoring in order to ensure that each of our query results gets added to a slice—let’s call it **`Cities`**—and that we pass that slice into our template.

Code Listing 13: Using the Template in Our “Cities” Application

```
package main

import (
	"database/sql"
	"html/template"
	"log"
	"net/http"

	_ "github.com/go-sql-driver/mysql"
)

type City struct {
	Name        string
	CountryCode string
	Population  int
}

var database *sql.DB

func main() {
	db, err := sql.Open("mysql",
"root:password@tcp(127.0.0.1:3306)/world")
	if err != nil {
		log.Println("Could not connect!")
	}
	database = db
	log.Println("Connected.")

	http.HandleFunc("/", showCity)
	http.ListenAndServe(":8999", nil)
}

func showCity(w http.ResponseWriter, r *http.Request) {
	var Cities = []City{}
	queryParam := "%" + r.URL.Path[1:] + "%"
	cities, err := database.Query("SELECT Name, CountryCode,
		    Population FROM city WHERE Name LIKE ?", queryParam)
	if err != nil {
		log.Fatal(err)
	}
	defer cities.Close()

	for cities.Next() {
		theCity := City{}
		cities.Scan(&theCity.Name, &theCity.CountryCode,
		                                   &theCity.Population)
		Cities = append(Cities, theCity)
	}

	t, _ := template.ParseFiles("results.html")

	t.Execute(w, Cities)
}
```

Next, we can execute the program and search for cities just as we did before.



Search Results

City	Country Code	Population
LÂ´Hospitalet de Llobregat	ESP	247986
Barletta	ITA	91904
Valletta	MLT	7073
Jelets	RUS	119400

Figure 16: The Cities Application with Template

Let’s consider some of the benefits of using templates. First, our code is much neater. We don’t need to wade through long strings of tags in order to work out what’s going on.

Second, we can change the template at any time, no recompilation necessary.

Third, the template syntax is simple enough that we could hand it over to someone who knows nothing about Go, or about web development in general, but who might have a better handle on design than we do.

What else can we do with Go templates? Quite a lot, as it happens. Check out the official documentation for the definitive list. In the meantime, there are several features that are so useful it would be remiss of me not to at least mention them—embedded methods and conditionals.

Using embedded methods in templates

Let’s say we want to do something with our templated output. Perhaps we’d like to format the Population column in the preceding example in order to use a comma as a thousands separator.

Sure, we could add an extra string field in the **`City`** structure to display formatted output and write the formatted **`Population`** column figure to it each time we retrieve a row from the table. But, technically, that’s presentation, right? And haven’t we been keen to get as much presentation logic out of our main code as possible?

Also, this is especially cumbersome because Go’s **`fmt`** package doesn’t provide this functionality out of the box, which means we’ll have to roll our own.

So, let’s see if we can offload some of that responsibility to the template. In fact, we can do that by writing a method in our application that we’ll get the template to invoke. The best way to do that is to add a method directly to our **`City`** structure, as in Code Listing 14.

Code Listing 14: Adding a Custom Function to the City Struct

```
func (c City) FormatPopulation(n int, sep rune) string {

	s := strconv.Itoa(n)

	startOffset := 0
	var buff bytes.Buffer

	if n < 0 {
		startOffset = 1
		buff.WriteByte('-')
	}

	l := len(s)

	commaIndex := 3 - ((l - startOffset) % 3)

	if commaIndex == 3 {
		commaIndex = 0
	}

	for i := startOffset; i < l; i++ {

		if commaIndex == 3 {
			buff.WriteRune(sep)
			commaIndex = 0
		}
		commaIndex++

		buff.WriteByte(s[i])
	}

	return buff.String()

}
```

Note that the function accepts an integer value and a separator and returns a string (a tip of the hat to Ivan Tung for this function, which saved me having to write it myself!).

You can think of the Go **rune** keyword as an alias for a Unicode character, which is sometimes called a “code point” in Go terminology.

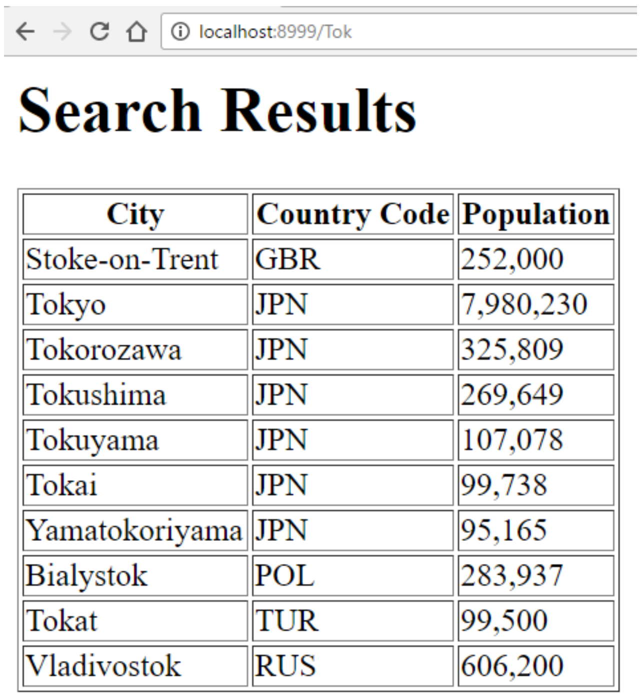
In order to call this function within our template, we simply specify its name followed by any parameters (not in parentheses), as shown in Code Listing 15.

Code Listing 15: Calling the FormatPopulation Function from the Template

```
<html>
```

```
<head>
<title>City Search</title>
</head>
<body>
  <h1>Search Results</h1>
  <table border='1'>
    <tr>
      <th>City</th>
      <th>Country Code</th>
      <th>Population</th>
    </tr>
    {{range .}}
      <tr>
        <td>{{.Name}}</td>
        <td>{{.CountryCode}}</td>
        <td>{{.FormatPopulation .Population}}</td>
      </tr>
    {{end}}
  </table>
</body>
</html>
```

When we execute the application, we get nicely formatted population values, as in Figure 17.



localhost:8999/Tok

Search Results

City	Country Code	Population
Stoke-on-Trent	GBR	252,000
Tokyo	JPN	7,980,230
Tokorozawa	JPN	325,809
Tokushima	JPN	269,649
Tokuyama	JPN	107,078
Tokai	JPN	99,738
Yamatokoriyama	JPN	95,165
Bialystok	POL	283,937
Tokat	TUR	99,500
Vladivostok	RUS	606,200

Figure 16: The Population Values, Formatted by the Template

Using conditionals in templates

You can offload the evaluation of conditional expressions to your template using **`{{if}}`** **`{{else}}`** **`{{end}}`**.

Go supports several functions that support basic types, such as **`eq`** (equals), **`ne`** (not equal to), or **`gt`** (greater than) that you can use for building expressions.

The use of conditional expressions in templates is best demonstrated by an example.

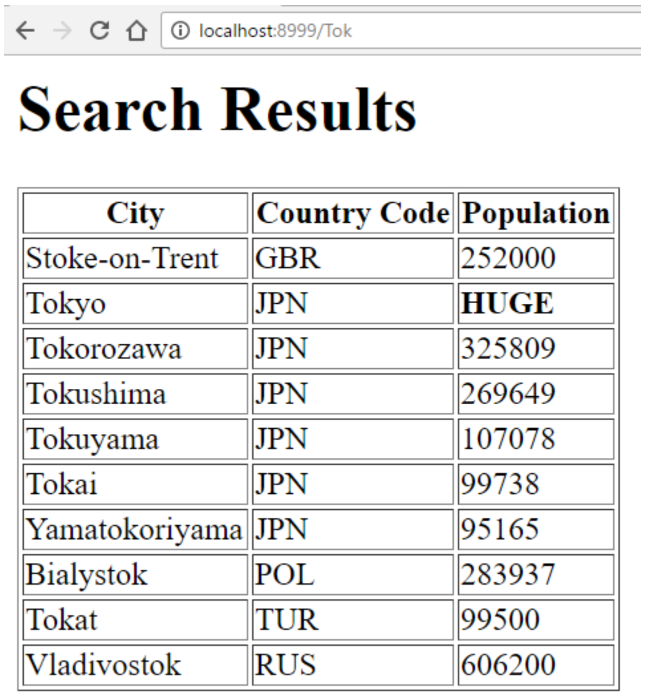
Let’s say that we’re only interested in populations of less than 5,000,000. Everything else we’ll count as huge and won’t bother displaying the actual figure.

Code Listing 16 shows our template.

Code Listing 16: Template with Conditional Expression

```
<html>
<head>
<title>City Search</title>
</head>
<body>
  <h1>Search Results</h1>
  <table border='1'>
    <tr>
      <th>City</th>
      <th>Country Code</th>
      <th>Population</th>
    </tr>
    {{range .}}
      <tr>
        <td>{{.Name}}</td>
        <td>{{.CountryCode}}</td>
        <td>{{if gt .Population 5000000}} <b>HUGE</b>
            {{else}} {{.Population}}
            {{end}}
        </td>
      </tr>
    {{end}}
  </table>
</body>
</html>
```

Figure 18 shows the result.



localhost:8999/Tok

Search Results

City	Country Code	Population
Stoke-on-Trent	GBR	252000
Tokyo	JPN	**HUGE**
Tokorozawa	JPN	325809
Tokushima	JPN	269649
Tokuyama	JPN	107078
Tokai	JPN	99738
Yamatokoriyama	JPN	95165
Bialystok	POL	283937
Tokat	TUR	99500
Vladivostok	RUS	606200

Figure 17: Displaying Population Values Using Conditionals in the Template

Chapter 5 Creating a RESTful JSON API

We've learned a lot already, so let's see if we can take all that knowledge and do something useful with it.

In Chapter 1, we examined how Go is an excellent choice for creating web services for many reasons—for example, its ability to scale massively through its lightweight threading model, its ease with modularizing code, and its ability to integrate with common tools such as cryptographic libraries, secure web protocols, and, of course, HTTP servers.

In this chapter, I want to demonstrate how we can create a simple RESTful web service that can accept JSON requests and return JSON responses in order to facilitate CRUD (Create, Read, Update, and Delete) operations on a database.

(Actually, I'm going a cheat a little here. In order to minimize the complexity of our application, I'm going to create a CRD application. The Update functionality will be left as an exercise for the reader!)

This application will use what we've learned so far about serving, routing, and accessing a database. In building it, I'll show you how you might design and develop a typical web service. It won't be perfect or production-ready, but it should give you a good idea of how easy it is to build these sorts of services in Go.

RESTful APIs

If you've been a developer, or if you've spent any time with developers, you have no doubt heard about REST. Given the amount of hype it has attracted in recent years, you would think that it, too, was a recent concept. But it's not. In fact, it's as old as the web itself.

REST is simply a response to the hundreds of different protocols that have been developed over the years that have aimed to get computers talking to each other over networks using the same language.

Some of these protocols have included SOAP (which, because of its reliance on XML as a transport mechanism, requires a fair amount of data and computing power and began to fall out of favor as mobile devices became ubiquitous), JMS (which is specific to Java applications and therefore not really geared for widespread adoption), and XML-RPC (which, like SOAP, uses XML but without implementing any of the standards that SOAP has).

As with all these approaches, the aim behind REST was to make sharing data easy for computers while at the same time being transparent enough that human observers can understand what they are doing.

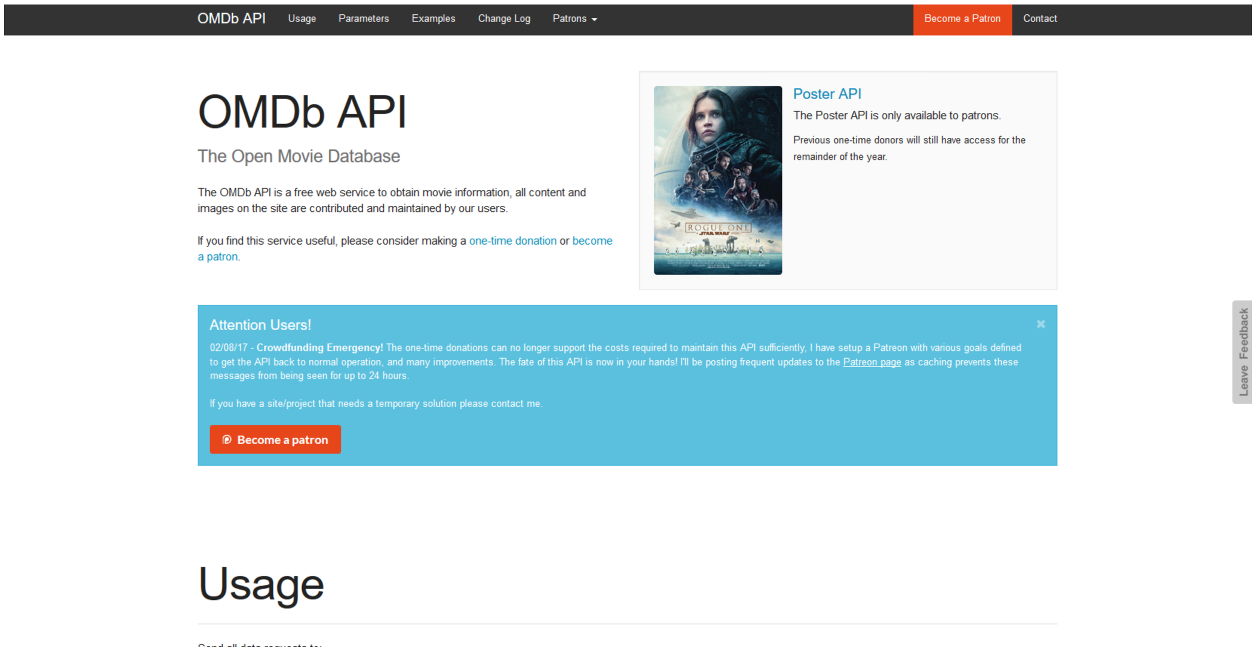
What REST offers, however, is the ability to do this while remaining very lightweight. Its methods are familiar to developers because it uses the same, established methods employed by the web itself.

How so?

Well, consider a website that you access in a browser. You enter a URL to visit the site, and the program behind the site can analyze that URL in order to understand your intention. In a static site, you're simply entering URLs that point to file resources on the web server, but in a site that employs web services, you can effectively use the URL as a command line and enter requests not only for resources, but also for specific operations that the program's API exposes.

Consider the Open Movie Database website (http://www.omdbapi.com) that provides a free REST API to access information about movies. You need only to craft the URL so that it specifies what information you want it to give you.

If you start at https://www.omdbapi.com/, you get the home page in Figure 19.



OMDb API

The Open Movie Database

The OMDb API is a free web service to obtain movie information, all content and images on the site are contributed and maintained by our users.

If you find this service useful, please consider making a one-time donation or become a patron.

Poster API

The Poster API is only available to patrons.

Previous one-time donors will still have access for the remainder of the year.

Attention Users!

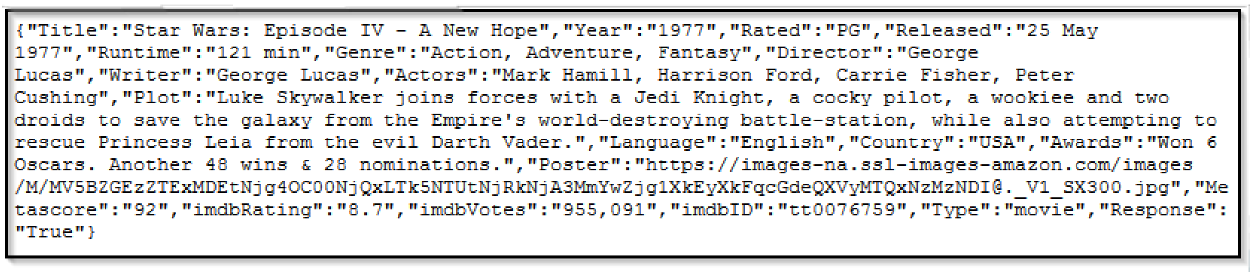
02/08/17 - **Crowdfunding Emergency!** The one-time donations can no longer support the costs required to maintain this API sufficiently, I have setup a Patreon with various goals defined to get the API back to normal operation, and many improvements. The fate of this API is now in your hands! I'll be posting frequent updates to the Patreon page as caching prevents these messages from being seen for up to 24 hours.

If you have a site/project that needs a temporary solution please contact me.

Usage

Figure 18: The Open Movie Database API Home Page

If you specify a movie using the **t** (title) parameter, you get something a little more interesting: https://www.omdbapi.com/?t=star%20wars.



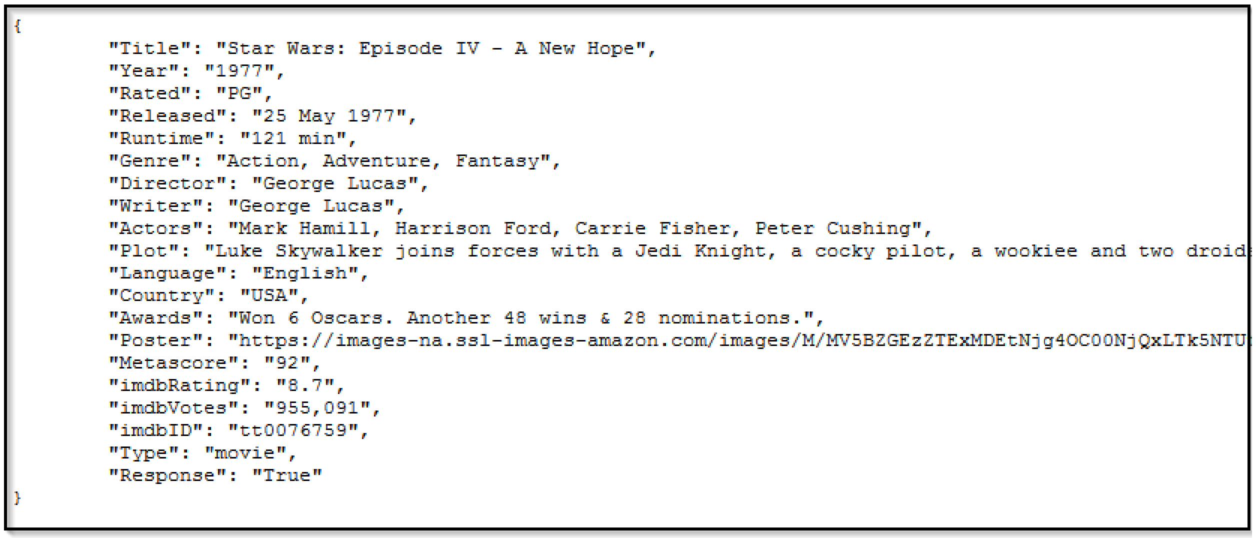
```
{"Title":"Star Wars: Episode IV - A New Hope","Year":"1977","Rated":"PG","Released":"25 May
1977","Runtime":"121 min","Genre":"Action, Adventure, Fantasy","Director":"George
Lucas","Writer":"George Lucas","Actors":"Mark Hamill, Harrison Ford, Carrie Fisher, Peter
Cushing","Plot":"Luke Skywalker joins forces with a Jedi Knight, a cocky pilot, a wookiee and two
droids to save the galaxy from the Empire's world-destroying battle-station, while also attempting to
rescue Princess Leia from the evil Darth Vader.","Language":"English","Country":"USA","Awards":"Won 6
Oscars. Another 48 wins & 28 nominations.","Poster":"https://images-na.ssl-images-amazon.com/images
/M/MV5BZGEzZTExMDEtNjg4OC00NjQxLTk5NTUtNjRkNjA3MmYwZjg1XkEyXkFqcGdeQXVyMTQxNzMzNDI@._V1_SX300.jpg","Me
tascore":"92","imdbRating":"8.7","imdbVotes":"955,091","imdbID":"tt0076759","Type":"movie","Response":
"True"}
```

Figure 19: Accessing the Open Movie Database API

That information is JSON, or JavaScript Object Notation. Despite the name, it's not specific to JavaScript. In fact, it's a common way of encoding data in a very lightweight format, and it's used by applications written in many different languages, including Go.

Apart from being lightweight and easily parsed by client applications, JSON is also human readable. Well, just barely. You can tidy up the response either by viewing it in your browser's developer console or by installing an extension that prettifies a JSON response.

Figure 21 shows the output after I've installed the JSON Formatter add-on in Firefox.



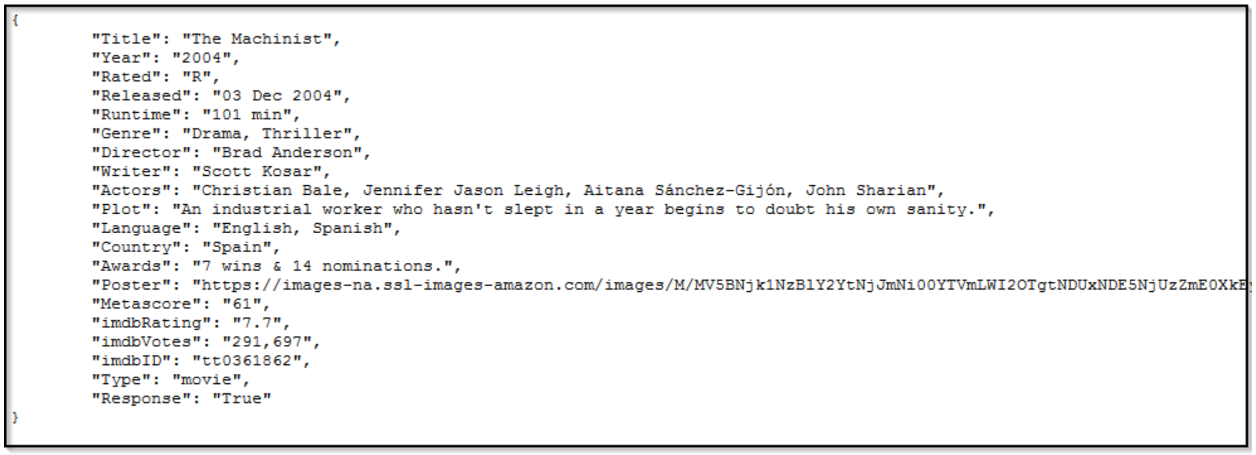
```
{
        "Title": "Star Wars: Episode IV - A New Hope",
        "Year": "1977",
        "Rated": "PG",
        "Released": "25 May 1977",
        "Runtime": "121 min",
        "Genre": "Action, Adventure, Fantasy",
        "Director": "George Lucas",
        "Writer": "George Lucas",
        "Actors": "Mark Hamill, Harrison Ford, Carrie Fisher, Peter Cushing",
        "Plot": "Luke Skywalker joins forces with a Jedi Knight, a cocky pilot, a wookiee and two droid
        "Language": "English",
        "Country": "USA",
        "Awards": "Won 6 Oscars. Another 48 wins & 28 nominations.",
        "Poster": "https://images-na.ssl-images-amazon.com/images/M/MV5BZGEzZTExMDEtNjg4OC00NjQxLTk5NTU
        "Metascore": "92",
        "imdbRating": "8.7",
        "imdbVotes": "955,091",
        "imdbID": "tt0076759",
        "Type": "movie",
        "Response": "True"
}
```

Figure 20: The JSON Response, Prettified

As you can see, JSON is simply a bunch of key/value pairs. Everything inside the curly braces is a JSON object, and objects themselves can contain other objects and arrays of values.

However, this isn't a JSON tutorial. You can find out about JSON almost anywhere on the web, and there isn't a lot to it. The takeaway here is that APIs exist that can parse a URL and return data in the JSON format that can be consumed by a client application. Some sites can even accept JSON, too, but the Open Movie Database isn't one of them.

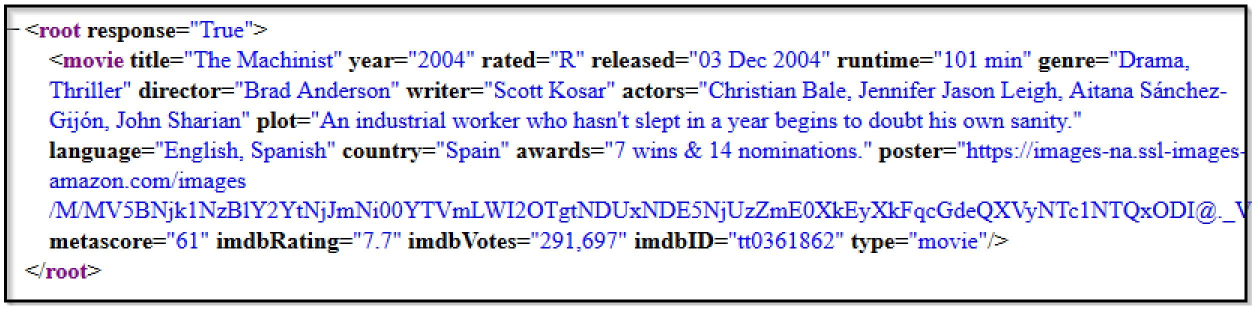
We can also add other criteria. Figure 22 looks for a film called *The Machinist*, includes a short plot summary, and returns the results as JSON (the default): http://www.omdbapi.com/?t=the+machinist&y=&plot=short&r=json.



```
{
        "Title": "The Machinist",
        "Year": "2004",
        "Rated": "R",
        "Released": "03 Dec 2004",
        "Runtime": "101 min",
        "Genre": "Drama, Thriller",
        "Director": "Brad Anderson",
        "Writer": "Scott Kosar",
        "Actors": "Christian Bale, Jennifer Jason Leigh, Aitana Sánchez-Gijón, John Sharian",
        "Plot": "An industrial worker who hasn't slept in a year begins to doubt his own sanity.",
        "Language": "English, Spanish",
        "Country": "Spain",
        "Awards": "7 wins & 14 nominations.",
        "Poster": "https://images-na.ssl-images-amazon.com/images/M/MV5BNjk1NzBlY2YtNjJmNi00YTVmLWI2OTgtNDUxNDE5NjUzZmE0XkE
        "Metascore": "61",
        "imdbRating": "7.7",
        "imdbVotes": "291,697",
        "imdbID": "tt0361862",
        "Type": "movie",
        "Response": "True"
}
```

Figure 21: Open Movie Database API Results for The Machinist

Figure 23 sets the **r** (response) parameter to XML: http://www.omdbapi.com/?t=the+machinist&y=&plot=short&r=xml.



```
- <root response="True">
    <movie title="The Machinist" year="2004" rated="R" released="03 Dec 2004" runtime="101 min" genre="Drama,
    Thriller" director="Brad Anderson" writer="Scott Kosar" actors="Christian Bale, Jennifer Jason Leigh, Aitana Sánchez-
    Gijón, John Sharian" plot="An industrial worker who hasn't slept in a year begins to doubt his own sanity."
    language="English, Spanish" country="Spain" awards="7 wins & 14 nominations." poster="https://images-na.ssl-images-
    amazon.com/images
    /M/MV5BNjk1NzBlY2YtNjJmNi00YTVmLWI2OTgtNDUxNDE5NjUzZmE0XkEyXkFqcGdeQXVyNTc1NTQxODI@._V
    metascore="61" imdbRating="7.7" imdbVotes="291,697" imdbID="tt0361862" type="movie"/>
  </root>
```

Figure 22: Changing the Output to XML Format

That's pretty concise as far as XML goes. A lot of XML is deeply nested and is a nightmare to parse, hence the preference for JSON in modern APIs.

Play around with the Open Movie Database API for a bit. I'll wait!

So, that's an example of a RESTful API. You're effectively using the URL itself to make calls to the web service.

Our RESTful web service

We're going to build something similar in this chapter.

You've probably already thought: "I could do that with Go routers!" And, indeed, you can. So, all you need to get going is some data. And we've already been playing around with MySQL's "world" database, so let's use that.

Our application will allow users to enter any of the following URL paths and respond in the way described:

- **`http://localhost:8999/city`**: Return a list of all cities in the City table in JSON format.
- **`http://localhost:8999/city/1028`**: Return details of the city with ID 1028 in JSON format.
- **`http://localhost:8999/cityadd/`**: Allow the user to **`POST`** a JSON representation of a new city and get a JSON result object back.
- **`http://localhost:8999/citydel/1028`**: Allow the user to delete the city with ID 1028 and get a JSON result object back.

In order to make this work while keeping our code as modular as possible, we're going to create the following files:

- **`main.go`**
- **`handlers.go`**
- **`router.go`**
- **`routes.go`**
- **`database.go`**
- **`city.go`**
- **`dbupdate.go`**

Those module names should be self-explanatory. Let's dive into it.

Serving and routing

Let's define our routes first. We're going to use **`gorilla/mux`** instead of **`net/http`**'s **`DefaultServeMux`** because **`gorilla/mux`** is generally nicer to work with. And instead of simply defining them one at a time, we're going to store route details in a struct in order to make it easier to add new routes when we need them.

Here is **`routes.go`**, which defines each of the routes and their handlers in a slice (roughly analogous to arrays in other programming languages) of **`Route`** types, called **`Routes`**.

Code Listing 17: routes.go

```go
package main

import "net/http"

type Route struct {
      Name        string
      Method      string
      Pattern     string
      HandlerFunc http.HandlerFunc
}

type Routes []Route

var routes = Routes{
      Route{
            "HomePage",
            "GET",
            "/",
            HomePage,
      },
      Route{
            "CityList",
            "GET",
            "/city",
            CityList,
      },
      Route{
            "CityDisplay",
            "GET",
            "/city/{id}",
            CityDisplay,
      },
      Route{
            "CityAdd",
            "POST",
            "/cityadd",
            CityAdd,
      },
      Route{
            "CityDelete",
            "GET",
            "/citydel/{id}",
            CityDelete,
      },
}
```

And here in Code Listing 18 is **`router.go`**, which loops through the **`Routes`** and creates handlers for them.

Code Listing 18: router.go

```
package main

import "github.com/gorilla/mux"

func NewRouter() *mux.Router {

      router := mux.NewRouter().StrictSlash(true)
      for _, route := range routes {
            router.
                  Methods(route.Method).
                  Path(route.Pattern).
                  Name(route.Name).
```

```
			Handler(route.HandlerFunc)
	}

	return router
}
```

In order to get all those routes up and running and ready to serve requests, we simply need to create an instance of **`NewRouter`** in our **`main`** function, but we'll get around to that in a bit.

Let's begin by having a look at each of the handlers in **`handlers.go`**.

First is the **`CityList`** handler, which gets called when a user visits http://localhost:8999/city.

Code Listing 19: The CityList Handler

```
func CityList(w http.ResponseWriter, r *http.Request) {
	// Query the database
	jsonCities := dbCityList()

	...
}
```

`CityList` calls **`dbCityList`** in our **`database.go`** module in order to query the database and return a list of cities in JSON format. In order to work with JSON in Go applications, you need the **`encoding/json`** package. Then you can call **`json.Marshal`** to turn a struct into JSON and **`json.Unmarshal`** to turn JSON into a struct.

Code Listing 20: Encoding the Query Response as JSON

```
// Find all cities and return as JSON.
func dbCityList() []byte {
	var cities Cities
	var city City

	cityResults, err := database.Query("SELECT * FROM city")
	if err != nil {
		log.Fatal(err)
	}
	defer cityResults.Close()

	for cityResults.Next() {
		cityResults.Scan(&city.Id, &city.Name,
		   &city.CountryCode, &city.District, &city.Population)
		cities = append(cities, city)
	}

	jsonCities, err := json.Marshal(cities)
	if err != nil {
```

```
            fmt.Printf("Error: %s", err)
      }
      return jsonCities
}
```

As in Chapter 3, we're using the **Query** method to pass SQL to our database server, then calling **Next** on the result set to iterate through each row that the query returns, and we're also calling **Scan** to map the columns in each row to fields in our **City** struct.

Back in our handler, **CityList**, we're writing out the appropriate JSON headers in the response, then returning the JSON we received from **dbCityList**, as in Code Listing 21.

Code Listing 21: Returning the JSON Response Containing a List of Cities

```
func CityList(w http.ResponseWriter, r *http.Request) {
      // Query the database.
      jsonCities := dbCityList()

      // Format the response.
      w.Header().Set("Content-Type", "application/json")
      w.WriteHeader(http.StatusOK)
      w.Write(jsonCities)
}
```

Let's look at another handler—**CityDisplay**. This handler accepts an **id** parameter for a specific city that we then pass to the **dbCityDisplay** function in **database.go**. If all goes well, we receive a JSON object that describes only that city and returns it in our response, as in Code Listing 22.

Code Listing 22: Sending a JSON Response for a Single City

```
func CityDisplay(w http.ResponseWriter, r *http.Request) {
      // Get URL parameter with the city ID to search for.
      vars := mux.Vars(r)
      cityId, _ := strconv.Atoi(vars["id"])

      // Query the database.
      jsonCity := dbCityDisplay(cityId)

      // Send the response.
      w.Header().Set("Content-Type", "application/json")
      w.WriteHeader(http.StatusOK)
      w.Write(jsonCity)
}
```

Code Listing 23 shows the **dbCityDisplay** function that provides the JSON data for a specific city.

Code Listing 23: Finding a Single City by Using QueryRow

```
// Find a single city based on ID and return as JSON.
func dbCityDisplay(id int) []byte {
      var city City

      err := database.QueryRow("SELECT * FROM city WHERE ID=?",
id).Scan(&city.Id, &city.Name, &city.CountryCode, &city.District,
&city.Population)
      if err != nil {
            log.Fatal(err)
      }

      jsonCity, err := json.Marshal(city)
      if err != nil {
            fmt.Printf("Error: %s", err)
      }
      return jsonCity
}
```

Note that **`dbCityDisplay`** expects to return only a single row from the database, so we're using **`QueryRow`** instead of **`Query`**. We don't need to defer the closing of the record set because that's done explicitly when the query returns with either a single record or with no record. We need to chain the call to **`Scan`** to the call to **`QueryRow`** because otherwise the recordset will be closed before we have a chance to work with it.

`CityAdd` is a little more complex because we must read the JSON from the body of the request, unmarshall it into a Go struct, then pass it to **`dbCityAdd`** to process it.

Code Listing 24: The CityAdd Function

```go
func CityAdd(w http.ResponseWriter, r *http.Request) {
	var city City

	// Read the body of the request.
	body, err := ioutil.ReadAll(io.LimitReader(r.Body, 1048576))
	if err != nil {
		panic(err)
	}
	if err := r.Body.Close(); err != nil {
		panic(err)
	}

	// Convert the JSON in the request to a Go type.
	if err := json.Unmarshal(body, &city); err != nil {
		w.Header().Set("Content-Type", "application/json")
		w.WriteHeader(422) // can't process!
		if err := json.NewEncoder(w).Encode(err); err != nil {
			panic(err)
		}
	}

	// Write to the database.
	addResult := dbCityAdd(city)

	// Format the response.
	w.Header().Set("Content-Type", "application/json")
	w.WriteHeader(http.StatusCreated)
	w.Write(addResult)
}
```

Notice how we’re setting a 1MB limit on the amount of data we’ll accept from the client. The last thing we want is some malicious user sending us a terabyte of data and crashing our server!

Tip: Always limit the amount of data you will allow a client to send to your application.

When we have the JSON in a Go struct, we can prepare a statement in SQL, then execute it with the new city record we want to add. The benefit of preparing a statement in this way, by using **`db.Prepare`**, is that it helps protect against SQL injection if the same malicious user tries to sabotage us using another favorite hacker’s technique.

Code Listing 25: Using a Prepared Statement for the INSERT Operation

```
// Create a new city based on the information supplied.
func dbCityAdd(city City) []byte {

	var addResult DBUpdate

	// Create prepared statement.
	stmt, err := database.Prepare("INSERT INTO City(Name, CountryCode, 
District, Population) VALUES(?,?,?,?)")
	if err != nil {
		log.Fatal(err)
	}

	// Execute the prepared statement and retrieve the results.
	res, err := stmt.Exec(city.Name, city.CountryCode,
				city.District, city.Population)
	if err != nil {
		log.Fatal(err)
	}
	lastId, err := res.LastInsertId()
	if err != nil {
		log.Fatal(err)
	}
	rowCnt, err := res.RowsAffected()
	if err != nil {
		log.Fatal(err)
	}

	// Populate DBUpdate struct with last Id and num rows affected.
	addResult.Id = lastId
	addResult.Affected = rowCnt

	// Convert to JSON and return.
	newCity, err := json.Marshal(addResult)
	if err != nil {
		fmt.Printf("Error: %s", err)
	}
	return newCity
}
```

Note also that the **INSERT** operation on the database retrieves some useful information—namely the ID of the recently added record and the number of records affected. We're storing this in a struct called **DBUpdate** so that we can return it to the **CityAdd** handler and, subsequently, to the client (who may wish to act on the information).

The last of our handlers is **CityDelete**. Like **CityDisplay**, it requires users to enter the ID of the city with which they want to work.

Code Listing 26: The CityDelete Function

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

	// Get URL parameter with the city ID to delete.
	vars := mux.Vars(r)
	cityId, _ := strconv.ParseInt(vars["id"], 10, 64)

	// Query the database.
	deleteResult := dbCityDelete(cityId)

	// Send the response.
	w.Header().Set("Content-Type", "application/json")
	w.WriteHeader(http.StatusOK)
	w.Write(deleteResult)
}
```

The corresponding **`dbCityDelete`** function takes that ID and prepares a **`DELETE`** statement with it. The database responds with the number of rows affected, and we can use the same **`DBUpdate`** struct to present that information to the user.

Code Listing 27: The dbCityDelete Function

```
// Delete the city with the supplied ID.
func dbCityDelete(id int64) []byte {
	var deleteResult DBUpdate

	// Create prepared statement.
	stmt, err := database.Prepare("DELETE FROM City WHERE ID=?")
	if err != nil {
		log.Fatal(err)
	}

	// Execute the prepared statement and retrieve the results.
	res, err := stmt.Exec(id)
	if err != nil {
		log.Fatal(err)
	}
	rowCnt, err := res.RowsAffected()
	if err != nil {
		log.Fatal(err)
	}

	// Populate DBUpdate struct with last Id and num rows affected.
	deleteResult.Id = id
	deleteResult.Affected = rowCnt

	// Convert to JSON and return.
	deletedCity, err := json.Marshal(deleteResult)
	if err != nil {
		fmt.Printf("Error: %s", err)
	}
	return deletedCity
}
```

And that's really it!

The complete application

Let's next examine a number of code listing examples that demonstrate the complete application as we separate the various concerns into these modules:

- **`main.go`**
- **`router.go`**
- **`routes.go`**
- **`handlers.go`**
- **`database.go`**
- **`city.go`**
- **`dbupdate.go`**

Code Listing 28: main.go

```
package main

import (
	"log"
	"net/http"
)

func main() {

	router := NewRouter()
	dbConnect()

	log.Fatal(http.ListenAndServe(":8999", router))
}
```

Code Listing 29: router.go

```
package main

import "github.com/gorilla/mux"

func NewRouter() *mux.Router {

      router := mux.NewRouter().StrictSlash(true)
      for _, route := range routes {
            router.
                  Methods(route.Method).
                  Path(route.Pattern).
                  Name(route.Name).
                  Handler(route.HandlerFunc)
      }

      return router
}
```

Code Listing 30: routes.go

```
package main

import "net/http"

type Route struct {
      Name        string
      Method      string
      Pattern     string
      HandlerFunc http.HandlerFunc
}

type Routes []Route

var routes = Routes{
      Route{
            "HomePage",
            "GET",
            "/",
            HomePage,
      },
      Route{
            "CityList",
            "GET",
            "/city",
            CityList,
      },
      Route{
```

```
			"CityDisplay",
			"GET",
			"/city/{id}",
			CityDisplay,
		},
		Route{
			"CityAdd",
			"POST",
			"/cityadd",
			CityAdd,
		},
		Route{
			"CityDelete",
			"GET",
			"/citydel/{id}",
			CityDelete,
		},
}
```

Code Listing 31: handlers.go

```
package main

import (
	"encoding/json"
	"fmt"
	"io"
	"io/ioutil"
	"net/http"
	"strconv"

	"github.com/gorilla/mux"
)

func HomePage(w http.ResponseWriter, r *http.Request) {
	fmt.Fprintln(w, "Welcome to the City Database!")
}

func CityList(w http.ResponseWriter, r *http.Request) {
	// Query the database.
	jsonCities := dbCityList()

	// Format the response.
	w.Header().Set("Content-Type", "application/json")
	w.WriteHeader(http.StatusOK)
	w.Write(jsonCities)
}

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

```go
	// Get URL parameter with the city ID to search for.
	vars := mux.Vars(r)
	cityId, _ := strconv.Atoi(vars["id"])

	// Query the database.
	jsonCity := dbCityDisplay(cityId)

	// Format the response.
	w.Header().Set("Content-Type", "application/json")
	w.WriteHeader(http.StatusOK)
	w.Write(jsonCity)
}

func CityAdd(w http.ResponseWriter, r *http.Request) {
	var city City

	// Read the body of the request.
	body, err := ioutil.ReadAll(io.LimitReader(r.Body, 1048576))
	if err != nil {
		panic(err)
	}
	if err := r.Body.Close(); err != nil {
		panic(err)
	}

	// Convert the JSON in the request to a Go type.
	if err := json.Unmarshal(body, &city); err != nil {
		w.Header().Set("Content-Type", "application/json")
		w.WriteHeader(422) // can't process!
		if err := json.NewEncoder(w).Encode(err); err != nil {
			panic(err)
		}
	}

	// Write to the database.
	addResult := dbCityAdd(city)

	// Format the response.
	w.Header().Set("Content-Type", "application/json")
	w.WriteHeader(http.StatusCreated)
	w.Write(addResult)
}

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

	// Get URL parameter with the city ID to delete.
	vars := mux.Vars(r)
	cityId, _ := strconv.ParseInt(vars["id"], 10, 64)
	// Query the database.
	deleteResult := dbCityDelete(cityId)

	// Format the response.
	w.Header().Set("Content-Type", "application/json")
	w.WriteHeader(http.StatusOK)
	w.Write(deleteResult)
}
```

Code Listing 32: database.go

```
package main

import (
	"database/sql"
	"encoding/json"
	"fmt"
	"log"

	_ "github.com/go-sql-driver/mysql"
)

var database *sql.DB

// Connect to the "world" database.
func dbConnect() {
	db, err := sql.Open("mysql",
		"root:password@tcp(127.0.0.1:3306)/world")
	if err != nil {
		log.Println("Could not connect!")
	}
	database = db
	log.Println("Connected.")
}

// Find all cities and return as JSON.
func dbCityList() []byte {
	var cities Cities
	var city City

	cityResults, err := database.Query("SELECT * FROM city")
	if err != nil {
		log.Fatal(err)
	}
	defer cityResults.Close()

	for cityResults.Next() {
		cityResults.Scan(&city.Id, &city.Name,
		  &city.CountryCode, &city.District, &city.Population)
		cities = append(cities, city)
	}

	jsonCities, err := json.Marshal(cities)
	if err != nil {
		fmt.Printf("Error: %s", err)
	}
	return jsonCities
}

// Find a single city based on ID and return as JSON.
func dbCityDisplay(id int) []byte {
	var city City

	err := database.QueryRow("SELECT * FROM city WHERE ID=?",
id).Scan(&city.Id, &city.Name, &city.CountryCode, &city.District,
&city.Population)
	if err != nil {
		log.Fatal(err)
	}

	jsonCity, err := json.Marshal(city)
	if err != nil {
		fmt.Printf("Error: %s", err)
	}
	return jsonCity
}

// Create a new city based on the information supplied.
func dbCityAdd(city City) []byte {

	var addResult DBUpdate

	// Create prepared statement.
	stmt, err := database.Prepare("INSERT INTO City(Name, CountryCode,
District, Population) VALUES(?,?,?,?)")
	if err != nil {
		log.Fatal(err)
	}

	// Execute the prepared statement and retrieve the results.
	res, err := stmt.Exec(city.Name, city.CountryCode, city.District,
city.Population)
	if err != nil {
		log.Fatal(err)
	}
	lastId, err := res.LastInsertId()
	if err != nil {
		log.Fatal(err)
	}
	rowCnt, err := res.RowsAffected()
	if err != nil {
		log.Fatal(err)
	}

	// Populate DBUpdate struct with last Id and num rows affected.
	addResult.Id = lastId
	addResult.Affected = rowCnt

	// Convert to JSON and return.
	newCity, err := json.Marshal(addResult)
	if err != nil {
		fmt.Printf("Error: %s", err)
	}
	return newCity
}

// Delete the city with the supplied ID.
func dbCityDelete(id int64) []byte {
	var deleteResult DBUpdate

	// Create prepared statement.
	stmt, err := database.Prepare("DELETE FROM City WHERE ID=?")
	if err != nil {
		log.Fatal(err)
	}

	// Execute the prepared statement and retrieve the results.
	res, err := stmt.Exec(id)
	if err != nil {
		log.Fatal(err)
	}
	rowCnt, err := res.RowsAffected()
	if err != nil {
		log.Fatal(err)
	}

	// Populate DBUpdate struct with last Id and num rows affected.
	deleteResult.Id = id
	deleteResult.Affected = rowCnt

	// Convert to JSON and return.
	deletedCity, err := json.Marshal(deleteResult)
	if err != nil {
		fmt.Printf("Error: %s", err)
	}
	return deletedCity
}
```

Code Listing 33: city.go

```
package main

type City struct {
      Id          int    `json:"id"`
      Name        string `json:"name"`
      CountryCode string `json:"country"`
      District    string `json:"district"`
      Population  int    `json:"pop"`
}

type Cities []City
```

Code Listing 34: The dbUpdate Struct in dbupdate.go

```
package main

type DBUpdate struct {
      Id       int64 `json:"id"`
      Affected int64 `json:"affected"`
}
```

Running the application

We can test the application from within a browser, but that will involve a bit of fiddling around with the developer console and more clicks than necessary.

Instead, let's use a command-line utility called **`curl`** ("client URL").

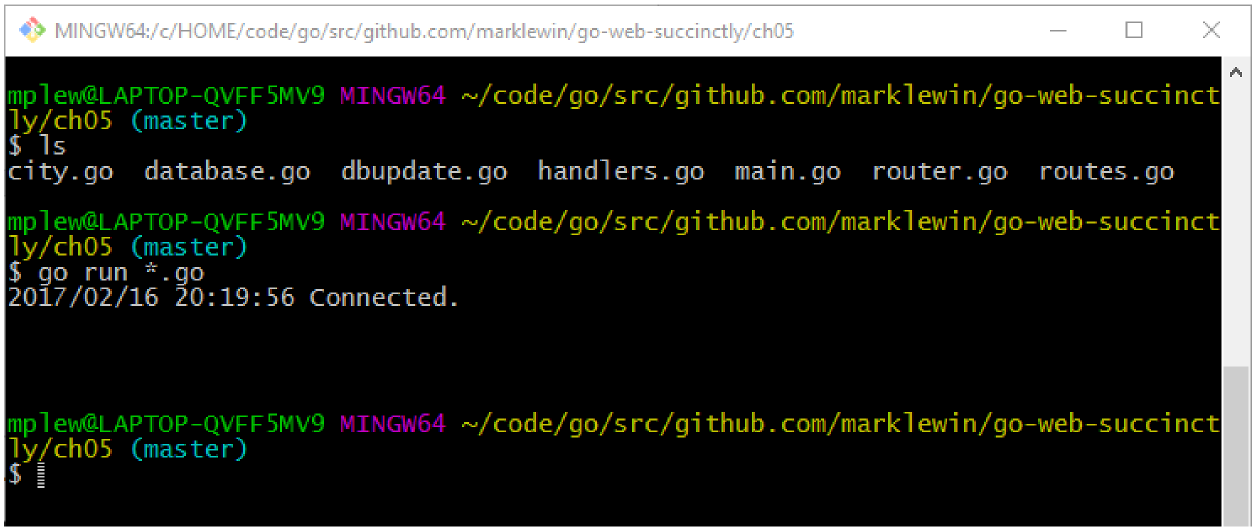
If you have a Mac or Linux machine, **`curl`** is probably already available to you. If you have Windows, perhaps the easiest way to access it is by downloading the Git Bash shell that contains **`curl`** and a wealth of other useful Linux tools. There's even a native Bash shell for Windows—see https://msdn.microsoft.com/en-us/commandline/wsl/about. It's handy to have around even if you don't use Git. Another alternative (although, in my opinion, a more bloated alternative) is Cygwin.

You can download Git Bash at https://git-for-windows.github.io/.

Displaying all cities

Open two shell windows.

Run the application in the first shell by executing **`go run *.go`** in the same folder as your application modules, as in Figure 24.


MINGW64:/c/HOME/code/go/src/github.com/marklewin/go-web-succinctly/ch05

```
mplew@LAPTOP-QVFF5MV9 MINGW64 ~/code/go/src/github.com/marklewin/go-web-succinct
ly/ch05 (master)
$ ls
city.go  database.go  dbupdate.go  handlers.go  main.go  router.go  routes.go

mplew@LAPTOP-QVFF5MV9 MINGW64 ~/code/go/src/github.com/marklewin/go-web-succinct
ly/ch05 (master)
$ go run *.go
2017/02/16 20:19:56 Connected.



mplew@LAPTOP-QVFF5MV9 MINGW64 ~/code/go/src/github.com/marklewin/go-web-succinct
ly/ch05 (master)
$
```

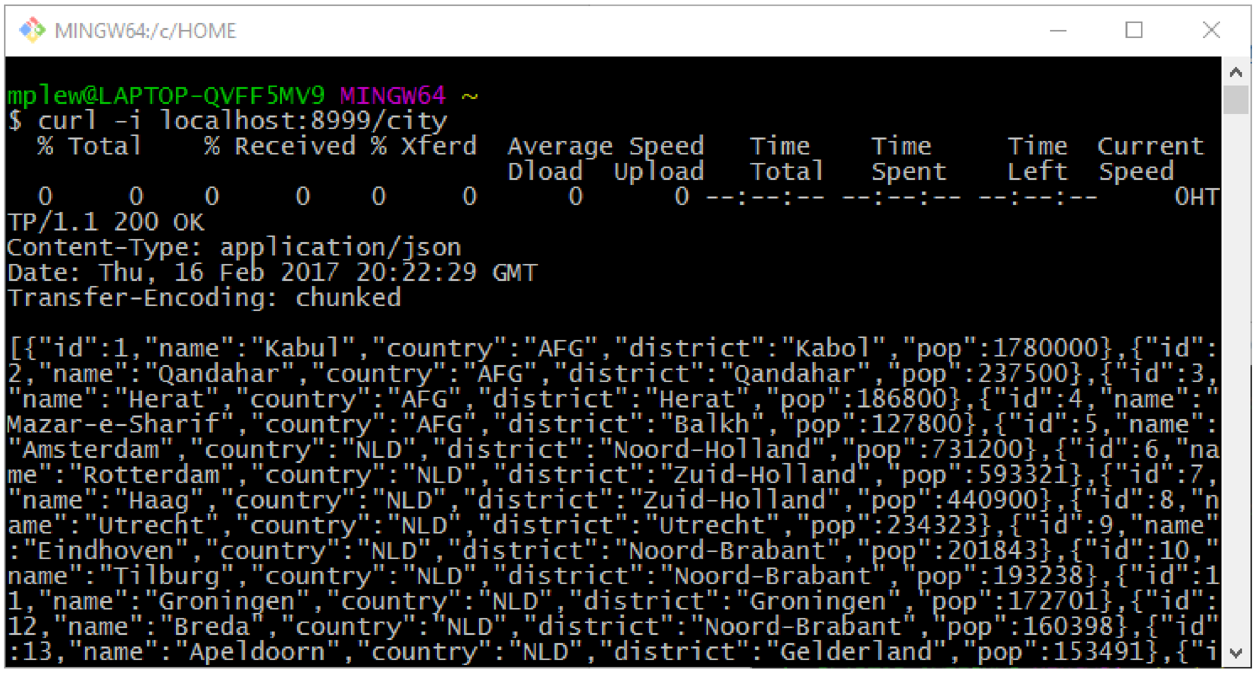
Figure 23: Running the Application

Next, enter the following command in the second shell window:

`curl -i localhost:8999/city`

The **`-i`** flag instructs **`curl`** to include the HTTP header.

You should see a JSON representation of every city in the City table, as shown in Figure 25.



```
mplew@LAPTOP-QVFF5MV9 MINGW64 ~
$ curl -i localhost:8999/city
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
  0     0    0     0    0     0      0      0 --:--:-- --:--:-- --:--:--     0HT
TP/1.1 200 OK
Content-Type: application/json
Date: Thu, 16 Feb 2017 20:22:29 GMT
Transfer-Encoding: chunked

[{"id":1,"name":"Kabul","country":"AFG","district":"Kabol","pop":1780000},{"id":
2,"name":"Qandahar","country":"AFG","district":"Qandahar","pop":237500},{"id":3,
"name":"Herat","country":"AFG","district":"Herat","pop":186800},{"id":4,"name":"
Mazar-e-Sharif","country":"AFG","district":"Balkh","pop":127800},{"id":5,"name":
"Amsterdam","country":"NLD","district":"Noord-Holland","pop":731200},{"id":6,"na
me":"Rotterdam","country":"NLD","district":"Zuid-Holland","pop":593321},{"id":7,
"name":"Haag","country":"NLD","district":"Zuid-Holland","pop":440900},{"id":8,"n
ame":"Utrecht","country":"NLD","district":"Utrecht","pop":234323},{"id":9,"name"
:"Eindhoven","country":"NLD","district":"Noord-Brabant","pop":201843},{"id":10,"
name":"Tilburg","country":"NLD","district":"Noord-Brabant","pop":193238},{"id":1
1,"name":"Groningen","country":"NLD","district":"Groningen","pop":172701},{"id":
12,"name":"Breda","country":"NLD","district":"Noord-Brabant","pop":160398},{"id"
:13,"name":"Apeldoorn","country":"NLD","district":"Gelderland","pop":153491},{"i
```

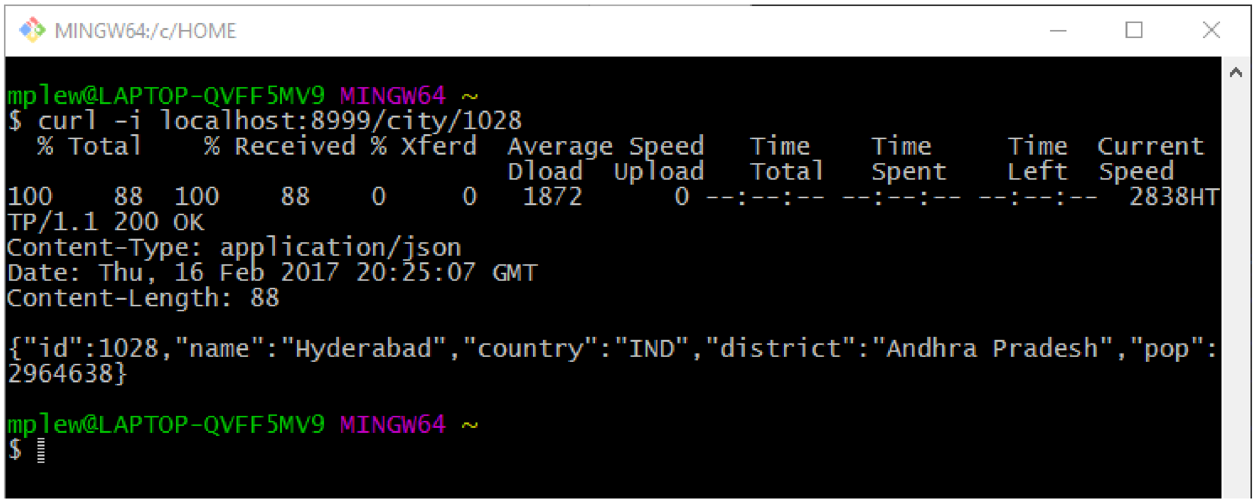
Figure 24: Requesting All Cities

Displaying a specific city

With the application still running in the first shell window, enter the following in the second shell window:

```
curl -i localhost:8999/city/1028
```

You should see details for the city with the ID of 1028—namely, Hyderabad in India. Enter some random city IDs and see which cities are referenced.



```
mplew@LAPTOP-QVFF5MV9 MINGW64 ~
$ curl -i localhost:8999/city/1028
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100    88  100    88    0     0   1872      0 --:--:-- --:--:-- --:--:--  2838HT
TP/1.1 200 OK
Content-Type: application/json
Date: Thu, 16 Feb 2017 20:25:07 GMT
Content-Length: 88

{"id":1028,"name":"Hyderabad","country":"IND","district":"Andhra Pradesh","pop":
2964638}

mplew@LAPTOP-QVFF5MV9 MINGW64 ~
$ 
```

Figure 25: Requesting a Specific City by ID

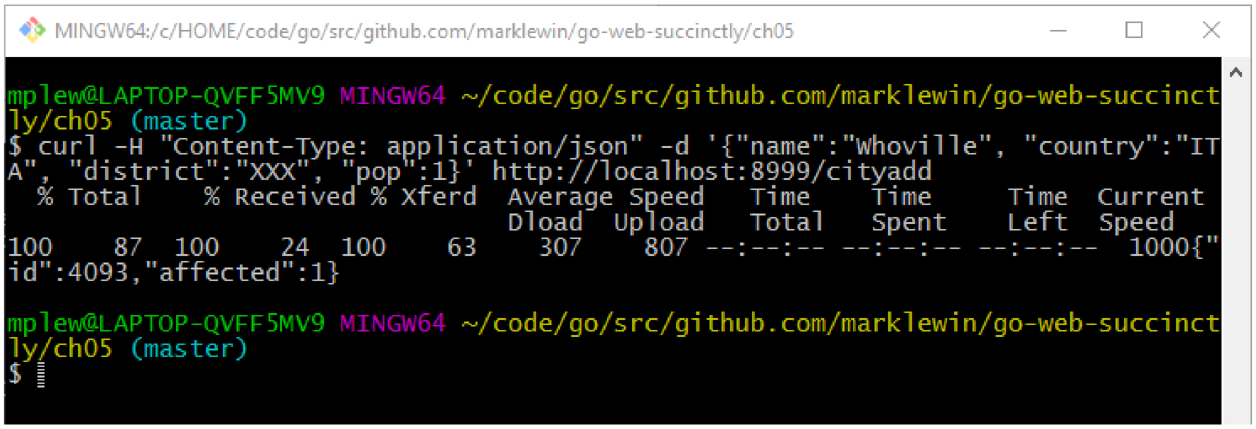
Adding a city

In the second shell window, enter the following:

```
curl -H "Content-Type: application/json" -d '{"name":"Whoville",
"country":"ITA", "district":"XXX", "pop":1}' http://localhost:8999/cityadd
```

Tip: Take care when entering the JSON object that represents the new city. JSON is simple, but it is not forgiving if you forget to close quotes or braces, or if you put quotes (that denote the string data type) around a value that is expected to be an integer.

The record should be added to the database and you should receive the new record ID in the response, as in Figure 27.



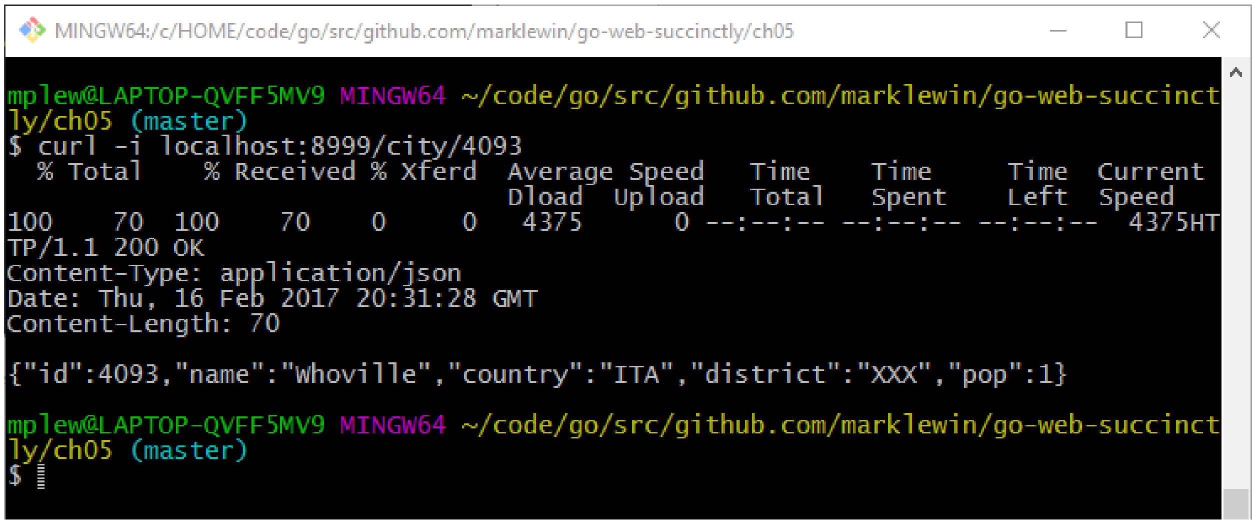
```
MINGW64:/c/HOME/code/go/src/github.com/marklewin/go-web-succinctly/ch05

mplew@LAPTOP-QVFF5MV9 MINGW64 ~/code/go/src/github.com/marklewin/go-web-succinct
ly/ch05 (master)
$ curl -H "Content-Type: application/json" -d '{"name":"Whoville", "country":"IT
A", "district":"XXX", "pop":1}' http://localhost:8999/cityadd
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100    87  100    24  100    63    307    807 --:--:-- --:--:-- --:--:--  1000{"
id":4093,"affected":1}

mplew@LAPTOP-QVFF5MV9 MINGW64 ~/code/go/src/github.com/marklewin/go-web-succinct
ly/ch05 (master)
$
```

Figure 26: Adding a New City

In the previous example, the ID of my recently added record is 4093. Yours is probably different. Rather than query the database again manually in order to see if the new record exists, you can simply use the following:

```
curl -i localhost:8999/city/4093
```

However, take care to replace 4093 with whatever ID was assigned when you added the city.



```
MINGW64:/c/HOME/code/go/src/github.com/marklewin/go-web-succinctly/ch05

mplew@LAPTOP-QVFF5MV9 MINGW64 ~/code/go/src/github.com/marklewin/go-web-succinct
ly/ch05 (master)
$ curl -i localhost:8999/city/4093
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100    70  100    70    0     0   4375      0 --:--:-- --:--:-- --:--:--  4375HT
TP/1.1 200 OK
Content-Type: application/json
Date: Thu, 16 Feb 2017 20:31:28 GMT
Content-Length: 70

{"id":4093,"name":"Whoville","country":"ITA","district":"XXX","pop":1}

mplew@LAPTOP-QVFF5MV9 MINGW64 ~/code/go/src/github.com/marklewin/go-web-succinct
ly/ch05 (master)
$
```

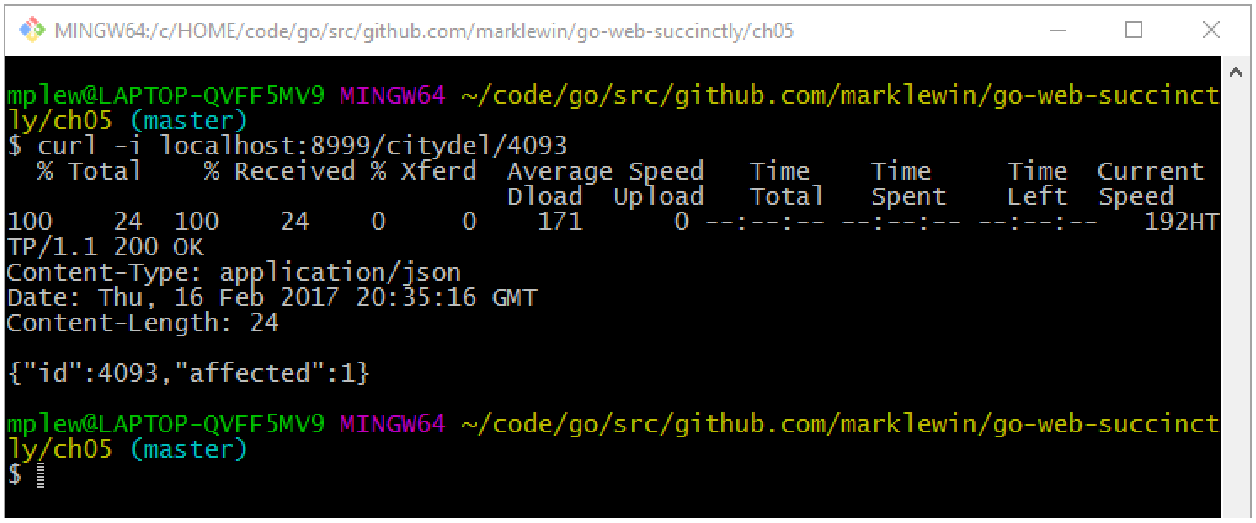
Figure 27: Verification—New City Added to the Database

Deleting a city

In order to delete a city, you use a syntax similar to your search for a city by ID. Try using the same city ID that you just added to the database. For example:

```
curl -i localhost:8999/citydel/4093
```

If everything proceeds according to plan, you should get a **`DBUpdate`** object encoded as JSON, which shows that a single row is affected, as in Figure 29.



MINGW64:/c/HOME/code/go/src/github.com/marklewin/go-web-succinctly/ch05

```
mplew@LAPTOP-QVFF5MV9 MINGW64 ~/code/go/src/github.com/marklewin/go-web-succinct
ly/ch05 (master)
$ curl -i localhost:8999/citydel/4093
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100    24  100    24    0     0    171      0 --:--:-- --:--:-- --:--:--   192HT
TP/1.1 200 OK
Content-Type: application/json
Date: Thu, 16 Feb 2017 20:35:16 GMT
Content-Length: 24

{"id":4093,"affected":1}

mplew@LAPTOP-QVFF5MV9 MINGW64 ~/code/go/src/github.com/marklewin/go-web-succinct
ly/ch05 (master)
$
```

Figure 28: Deleting a City

Congratulations! You have just written your first “real” web service using Go.

Of course, this is far from perfect, and if you were planning to put it into production, you would probably want to do a fair bit of refactoring, implement better error handling, and so on.

However, the point of all this is to emphasize that Go is an excellent language for working on this type of application. I’ve done similar work in other languages, including Node.js and Ruby, and I am much happier working in Go. Everything seems more tidy and better thought out, in my humble opinion.

Challenge step

If you’re up for a challenge, try to implement update functionality in the application—that is, to put the U back into CRUD!

For starters, allow the client to submit a JSON city object that will first delete the existing record (if there is a matching city ID), then add a new one and report success (or otherwise) in JSON format, too.

Next, allow the client to submit some partial JSON along with the ID, then simply update an existing record based on the fields that have changed. For example, the client might submit the following:

```
{"id":4088,"name":"Whereville”,pop":2}
```

This will only update the **`Name`** and **`Population`** fields in the City table.

Chapter 6 Cookies and Sessions

As you are doubtlessly aware, HTTP is a stateless protocol. Each request by a client to a web server is completely unrelated to any previous exchange between the two, and the communication mechanism consists solely of request/response pairs. The server is not required to retain any information regarding any previous requests.

The benefit of this approach is that the server doesn't need to assign memory to do so (because it doesn't "remember" any previous requests from the clients), and if the client connection dies, the server doesn't have to do any cleanup.

The downside of this approach is that, well, the server doesn't remember anything. This can make it tricky to build a rich, interactive web application because all too often we must send extra information with each request so that the server knows enough about the state of the client to provide a useful response.

The keys to making all this happen are cookies and sessions. Let's define both before we demonstrate how to use them in our Go web applications.

Introducing cookies and sessions

What is a cookie?

A cookie is simply a little text file that a browser puts on the user's computer. It stores information that helps to maintain the illusion of a persistent connection.

Cookies are often used for authentication, storing site-visitor preferences, maintaining shopping cart items, and identifying server sessions.

When the browser interacts with the web server, it passes the information in the cookie as part of the request. Note that cookies are domain-specific. If a browser creates a cookie for twitter.com, it cannot suddenly send that cookie to google.com.

Essentially, cookies are great for storing information about a user's interaction with a webpage as he or she moves from one page to the next.

What is a session?

Sessions allow you to store information about the client's interaction with a website just as cookies do, but the data gets stored on the server instead of on the client.

Sessions are a better alternative to stuffing lots of constantly changing information in cookies. Instead, the client stores only a unique identifier (the “Session ID”) and passes the ID to the web server with every request. The server uses the Session ID to look up information in its internal database and retrieve variables relating to the user's use of the application.

What is a session cookie? Or a persistent cookie?

Uh-oh. Just to confuse you even further, there are not only cookies and sessions, but there is also a session cookie!

With cookies, your application can set an expiry time. You might have seen this when the login page for a website gives you the “Keep me logged in?” option.

If you set an expiry time, the browser saves the cookie to the local file system. This is called a persistent cookie.

If you don’t set an expiry time, the browser usually keeps the cookie hanging around in memory, and this is called a session cookie.

So, session cookies and persistent cookies are simply cookies, but with different expiration times.

Working with cookies

Setting cookies

In order to write information to a cookie in Go, you use the **`net/http`**’s package’s **`SetCookie`** function, whose signature looks like this:

```
http.SetCookie(w ResponseWriter, cookie *Cookie)
```

The **`w`** is the response to the request and **`cookie`** is a struct:

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

The cookie can hold a lot of information, but the most important fields are:

- `Name`: A key for the cookie for referring to it in your code.
- `Value`: The cookie's data.
- `Expires`: A `Time` value that denotes when the browser can delete it.

Other fields that might be useful for controlling access to the cookie are:

- `Path`
- `Domain`
- `HttpOnly`

For now, let's keep it simple.

Here is an example of how you can set a cookie:

```
expiration := time.Now().Add(365 * 24 * time.Hour)
cookie := http.Cookie{Name: "username", Value: "jsmith", Expires: expiration}
http.SetCookie(w, &cookie)
```

Tip: Go's time functions are sophisticated but complex. For more information, see the documentation at https://golang.org/pkg/time/.

Fetching cookies

In order to retrieve a cookie from a request, you can do the following:

```
cookie, _ := r.Cookie("username")
fmt.Fprint(w, cookie.)
```

Or, if several cookies are associated with a request, you can iterate through them, as follows:

```
for _, cookie := range r.Cookies() {
    fmt.Fprint(w, cookie.Name)
}
```

Using cookies

Let's create a simple application that uses a cookie.

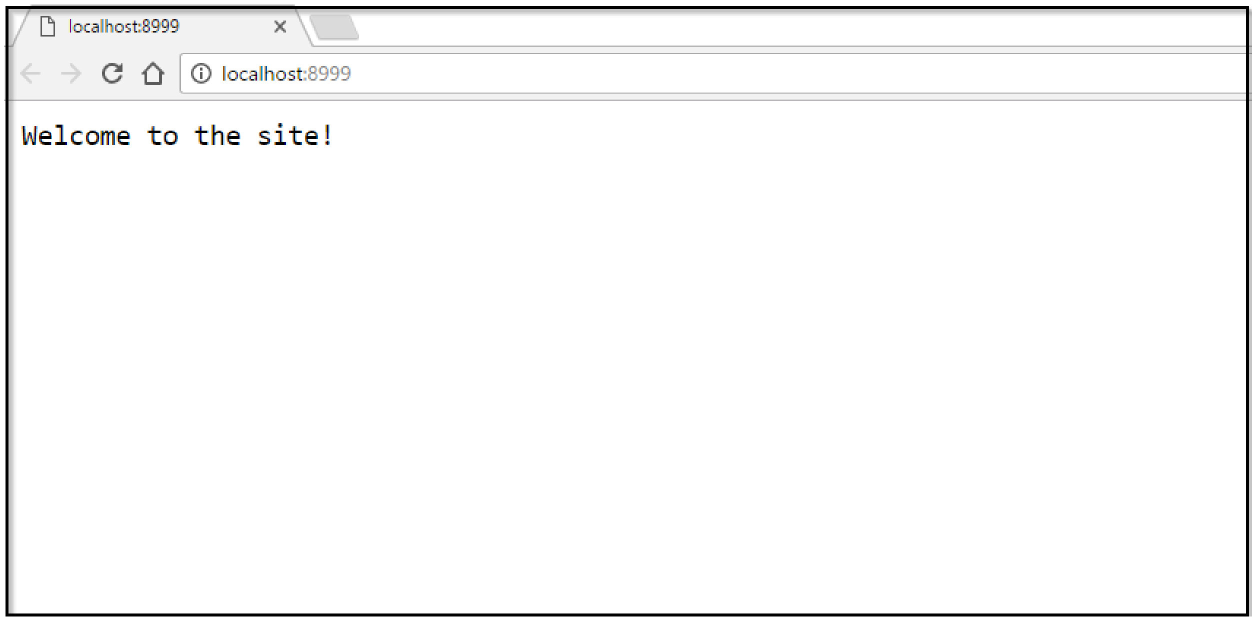
The following code checks to see if this is a visitor's first visit to our site. If it is, it displays a welcome message. If not, it displays the time of the last visit.

Code Listing 35: Using a Cookie to Check Site Visitor Status

```
package main

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

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

	c, err := r.Cookie("lastvisit") //

	expiry := time.Now().AddDate(0, 0, 1)

	cookie := &http.Cookie{
		Name:    "lastvisit",
		Expires: expiry,
		Value:   strconv.FormatInt(time.Now().Unix(), 10),
	}

	http.SetCookie(w, cookie)

	if err != nil {
		w.Write([]byte("Welcome to the site!"))
	} else {
		lasttime, _ := strconv.ParseInt(c.Value, 10, 0)
		html := "Welcome back! You last visited at: "
		html = html + time.Unix(lasttime, 0).Format("15:04:05")
		w.Write([]byte(html))
	}
}

func main() {
	http.HandleFunc("/", CheckLastVisit)
	http.ListenAndServe(":8999", nil)
}
```

The first line of code in the handler tries to drop the cookie to the user's browser. If it succeeds, it records the current time in the cookie's **`Value`** field and displays a "Welcome to the site!" message. If there is an error, this will occur because the cookie already exists, which means all it will do is update the time in the **`Value`** field and display it to users as the time they last visited.

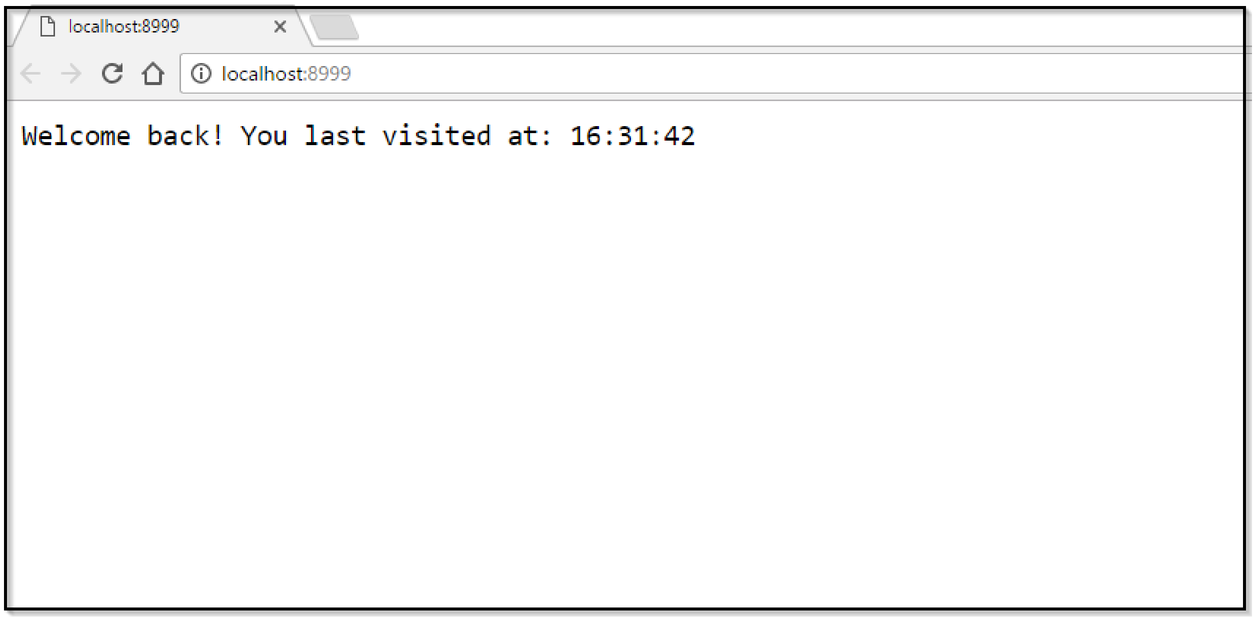
If we run the application and visit the root of the application, we'll see the message in Figure 30.



```
Welcome to the site!
```

Figure 29: First Visit to the Site

If we refresh the page, we'll see Figure 31.

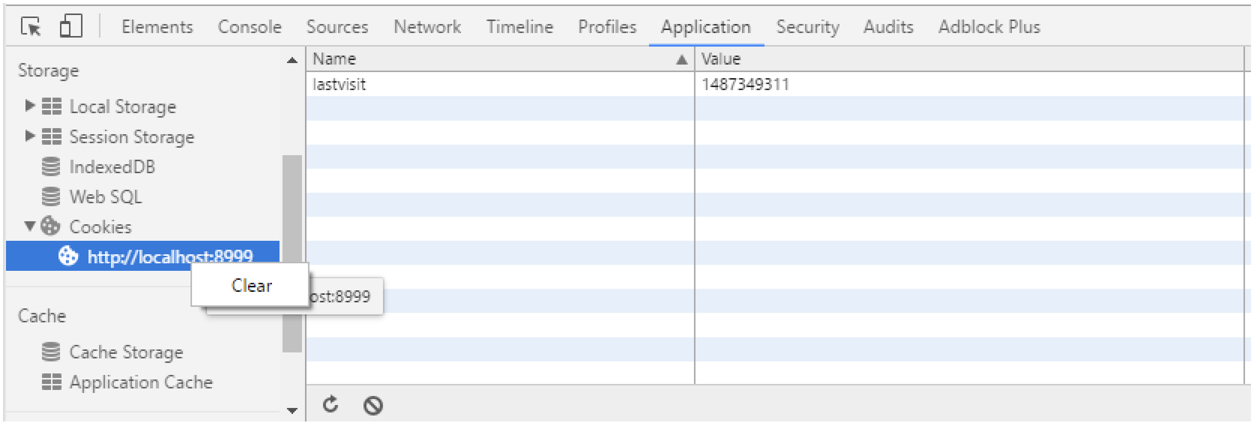


```
Welcome back! You last visited at: 16:31:42
```

Figure 30: Subsequent Visits to the Site

If you're using Chrome, as I am, you can delete the cookie this way:

1. Open Developer Tools (**Chrome menu** > **More Tools** > **Developer Tools**).
2. Select the **Network** tab.
3. Expand the Cookies section under Storage in the left pane.
4. Right-click the domain (http://localhost:8999) and click **Clear**.



Name	Value
lastvisit	1487349311

Figure 31: Deleting the Cookie Using Chrome's Web Developer Tools

If you’re using Firefox, you can delete the cookie this way:

1. From the Firefox button or the Tools menu, go to the Options > Privacy panel.
2. Select Firefox Will: Use Custom Settings for History.
3. Click **Show Cookies**.
4. Use the search box to enter the domain name of your site under development (http://localhost:8999) or drill down the folder lists to locate it.
5. Select the cookie in the list that you want to delete and click **Remove Cookie**.

If you delete the cookie and refresh the browser page, you’ll be presented with the “Welcome to the site!” message again.

Working with sessions

In order to work with sessions in Go, I’m going to suggest that we revisit the Gorilla Web Toolkit’s sessions module because, in my opinion, it’s a much cleaner implementation than the native Go approach.

Install it from GitHub as follows:

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

Basics

The following code demonstrates the basics of using the **`gorilla/sessions`** package to create and authenticate a session, retrieve the session, set some values, then save the session.

Code Listing 36: Using gorilla/sessions

```
package main

import (
	"net/http"

	"github.com/gorilla/mux"
	"github.com/gorilla/sessions"
)

var store = sessions.NewCookieStore(
				[]byte("keep-it-secret-keep-it-safe"))

func handler(w http.ResponseWriter, r *http.Request) {
	session, err := store.Get(r, "session-name")
	if err != nil {
		http.Error(w, err.Error(),
					http.StatusInternalServerError)
		return
	}

	// Set some session values.
	session.Values["abc"] = "cba"
	session.Values[111] = 222
	// Save the session values
	session.Save(r, w)
}

func main() {
	router := mux.NewRouter()
	http.Handle("/", router)
	router.HandleFunc("/", handler)
	http.ListenAndServe(":8999", nil)
}
```

This code first initializes the session store by calling **`NewCookieStore`** with a secret key used to authenticate the session.

Then, in our handler, we call the store's **`Get`** function to retrieve the session called "session-name." If it finds that, we'll have access to the session. If a session of that name does not exist, one will be created.

Tip: Session cookies created by Gorilla will last for a month by default. If this is too long for your requirements, you can either set the `MaxAge` property in each session's

`Options` or configure the session store so that all session cookies have the same `MaxAge` value.

When we have a session, we can assign values using its **`Values`** property. The **`Values`** property is a Go **`map`**, which is basically Go's implementation of a hash table that allows you to set properties using key/value pairs.

Next, we call the session's **`Save`** method to save the session in the response.

Tip: Always save the session values before returning the response. Otherwise, the response won't receive the session cookie.

That's Gorilla's basic implementation of sessions. However, gorilla/sessions also gives us extra, useful functionality that we will look at next.

Flash messages

Gorilla borrowed the idea of flash messages from Ruby. A flash message is simply a session value that exists until it is read.

We use flash messages to temporarily store data between requests, such as a success or error message during the form submission process, in order to avoid it being duplicated in error later.

We add a flash message using the session's **`AddFlash`** method and retrieve the flash messages by calling **`session.Flash`**.

We won't dwell too much on flash messages here—just be aware that they exist. Code Listing 37 demonstrates how to get and set flash messages.

Code Listing 37: Using Flash Messages in gorilla/sessions

```
package main

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

	"github.com/gorilla/mux"
	"github.com/gorilla/sessions"
)

var store = sessions.NewCookieStore(
					[]byte("keep-it-secret-keep-it-safe"))

func handler(w http.ResponseWriter, r *http.Request) {
	session, err := store.Get(r, "session-name")
	if err != nil {
```

```
            http.Error(w, err.Error(),
                                        http.StatusInternalServerError)
            return
      }

      // Get any previous flashes.
      if flashes := session.Flashes(); len(flashes) > 0 {
            // Do something with them
            for f := range flashes {
                  fmt.Println(flashes[f])
            }
      } else {
            // Set a new flash.
            session.AddFlash("Flash! Ah-ah, savior of the universe!"
                                                       + time.Now().String())
      }
      session.Save(r, w)
}

func main() {
      router := mux.NewRouter()
      http.Handle("/", router)
      router.HandleFunc("/", handler)
      http.ListenAndServe(":8999", nil)
}
```

Chapter 7 Development Techniques

If you've stuck around this long, you should now have a good grasp of the various bits and pieces that go toward creating a web application in Go. In this chapter, I want to cover a couple of techniques that will help you debug and test your Go web applications.

Logging

Any application that grows beyond the trivial will need good logging in order to enable its developers to locate and fix errors.

Basic logging

Go's **`log`** package lets you write to all the standard devices, custom files, or any destination that supports the **`io.Writer`** interface. So far, we've only logged to **`stdout`** (via **`fmt.Println`**, etc), but that's not a realistic option for production applications. So, let's consider how we might log output to files and be specific about which file gets which type of log message.

In the following code, we create an application that will log notices to **`notices.log`**, warnings to **`warnings.log`**, and errors to **`errors.log`**.

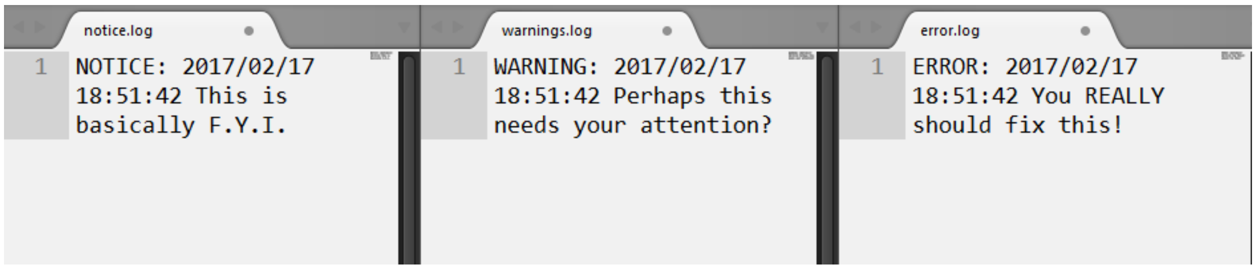
Code Listing 38: Logging to Different Files Based on the Type of Information

```
package main

import (
	"log"
	"os"
)

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

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

      Notice = log.New(noticeFile, "NOTICE: ", log.Ldate|log.Ltime)
      Notice.SetOutput(noticeFile)
      Notice.Println("This is basically F.Y.I.")

      Warning = log.New(warnFile, "WARNING: ", log.Ldate|log.Ltime)
      Warning.SetOutput(warnFile)
      Warning.Println("Perhaps this needs your attention?")

      Error = log.New(errorFile, "ERROR: ", log.Ldate|log.Ltime)
      Error.SetOutput(errorFile)
      Error.Println("You REALLY should fix this!")
}
```

Note that in the above code listing, **`0660`** is the file mode argument that enables reading and writing of the file to users and groups. For more information on file mode **`0660`** (and file modes in general), see http://www.filepermissions.com/file-permission/0660.

The output is as shown in Code Listing 39.

Code Listing 39: Log File Output



```
NOTICE: 2017/02/17 18:51:42 This is basically F.Y.I.
```

```
WARNING: 2017/02/17 18:51:42 Perhaps this needs your attention?
```

```
ERROR: 2017/02/17 18:51:42 You REALLY should fix this!
```

However, none of that is very useful for logging web requests, which any decent web server can do. So, let’s consider another approach for this.

Logging web requests

Let's revisit the RESTful web service that we created in Chapter 5. Remember how we split the routes from the router into **`routes.go`** and **`router.go`**, respectively? Let's look again at a number of code listing examples and separate the various concerns into these modules:

- **router.go**
- **routes.go**
- **logger.go**
- **logger.go containing Logger middleware**
- **Logger middleware added to routers**

Code Listing 40: router.go

```
package main

import "github.com/gorilla/mux"

func NewRouter() *mux.Router {

      router := mux.NewRouter().StrictSlash(true)
      for _, route := range routes {
            router.
                  Methods(route.Method).
                  Path(route.Pattern).
                  Name(route.Name).
                  Handler(route.HandlerFunc)
      }

      return router
}
```

Code Listing 41: routes.go

```
package main

import "net/http"

type Route struct {
      Name        string
      Method      string
      Pattern     string
      HandlerFunc http.HandlerFunc
}

type Routes []Route
```

```
var routes = Routes{
	Route{
		"HomePage",
		"GET",
		"/",
		HomePage,
	},
	Route{
		"CityList",
		"GET",
		"/city",
		CityList,
	},
	Route{
		"CityDisplay",
		"GET",
		"/city/{id}",
		CityDisplay,
	},
	Route{
		"CityAdd",
		"POST",
		"/cityadd",
		CityAdd,
	},
	Route{
		"CityDelete",
		"GET",
		"/citydel/{id}",
		CityDelete,
	},
}
```

Managing routes in this way makes decorating them with middleware very easy, as we discussed in Chapter 2. Consider this new addition to our web service application, shown in Code Listing 42.

Code Listing 42: logger.go, Containing Logger Middleware

```
package main

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

func Logger(inner http.Handler, name string) http.Handler {
      return http.HandlerFunc(func(w http.ResponseWriter,
                                              r *http.Request) {
            startTime := time.Now()

            inner.ServeHTTP(w, r)

            log.Printf(
                  "%s\t%s\t%s\t%s",
                  r.Method,
                  r.RequestURI,
                  name,
                  time.Since(startTime),
            )
      })
}
```

The new file **logger.go** contains a function called **Logger**. We can pass our handlers to **Logger**, which will then automatically provide logging and timing information.

Middleware is pretty cool, right?

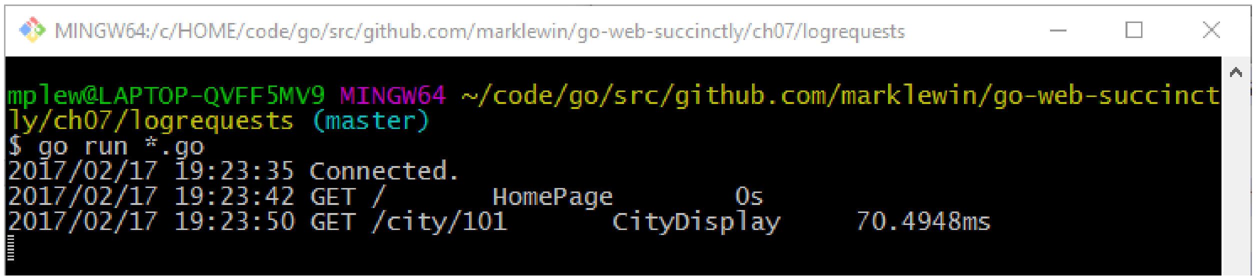
We can apply this middleware to our handlers by modifying the **NewRouter** function in **router.go**.

Code Listing 43: Adding Logger Middleware to Our Routers

```
package main

import (
      "net/http"

      "github.com/gorilla/mux"
)

func NewRouter() *mux.Router {

      router := mux.NewRouter().StrictSlash(true)
      for _, route := range routes {

            var handler http.Handler
            handler = route.HandlerFunc
            handler = Logger(handler, route.Name)

            router.
                  Methods(route.Method).
                  Path(route.Pattern).
                  Name(route.Name).
                  Handler(handler)
```

```
	}

	return router
}
```

Next, when we launch the application and create a request, our console displays log messages as in Figure 33.



MINGW64:/c/HOME/code/go/src/github.com/marklewin/go-web-succinctly/ch07/logrequests

```
mplew@LAPTOP-QVFF5MV9 MINGW64 ~/code/go/src/github.com/marklewin/go-web-succinct
ly/ch07/logrequests (master)
$ go run *.go
2017/02/17 19:23:35 Connected.
2017/02/17 19:23:42 GET /        HomePage        0s
2017/02/17 19:23:50 GET /city/101        CityDisplay     70.4948ms
```

Figure 32: Testing the Logger Middleware

Testing

Testing is baked into the Go language, with the **testing** package in the standard library. Testing offers the ability to run tests by executing the **go test** command.

And better still, Go testing is quite simple and intuitive. In order to write a test, you need only to import the **testing** package and write a test function.

All test functions in Go begin with the word "Test" and receive a single parameter of type ***testing.T**:

```
package foo

import "testing"

func TestSomething(t *testing.T) {
	// do your testing here...
}
```

When you have written a test and saved it in a file with a name ending in **_test.go**, you next exercise it by running **go test** in the same directory.

Let's look at an example of how it all works.

We'll create a very simple function that calculates the average of a slice of numbers in a function called **Average**, in a package called **math**, in a file called **average.go**.

Code Listing 44: The Average Function in the Math Package

```go
package math

func Average(nums []float64) float64 {
	total := float64(0)
	for _, x := range nums {
		total += x
	}
	return total / float64(len(nums))
}
```

Code Listing 45 shows the **`main`** function in **`main.go`** that we would normally use to invoke our **`Average`** function.

Code Listing 45: main.go, in Package Main

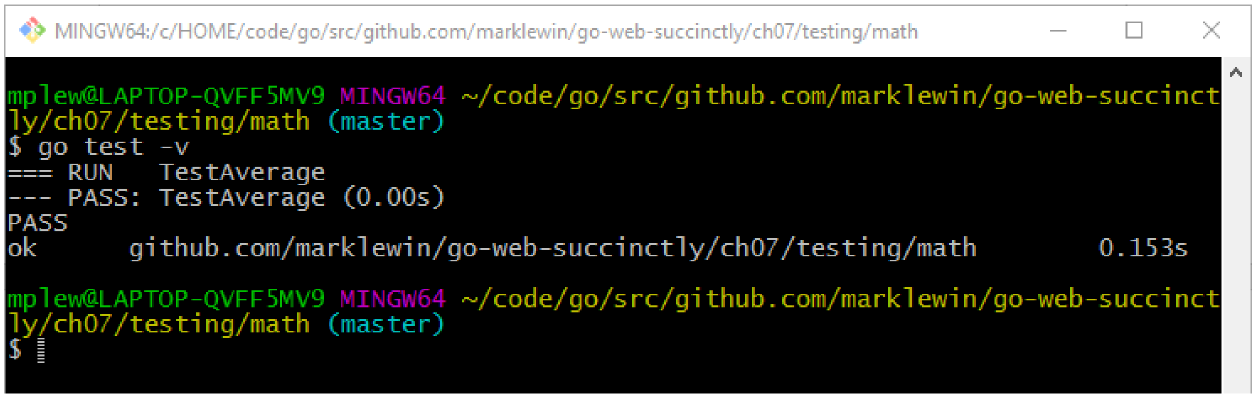
```go
package main

import (
	"fmt"

	"github.com/marklewin/go-web-succinctly/ch07/testing/math"
)

func main() {
	nums := []float64{1, 2, 3, 4}
	avg := math.Average(nums)
	fmt.Println(avg)
}
```

Next, we can create a function called **`TestAverage`** in a file called **`average_test.go`**, also in **`math`**, in order to test the function.

Code Listing 46: The Code to Test the Average Function in the Math Package

```go
package math

import "testing"

func TestAverage(t *testing.T) {
	var v float64
	v = Average([]float64{1, 2, 3, 4, 5})
	if v != 3.0 {
		t.Error("Expected 3.0, got ", v)
	}
}
```

Now, run the test from the same directory in which the files reside. Use the **`-v`** flag in **`go test`** for verbose output.


MINGW64:/c/HOME/code/go/src/github.com/marklewin/go-web-succinctly/ch07/testing/math

```
mplew@LAPTOP-QVFF5MV9 MINGW64 ~/code/go/src/github.com/marklewin/go-web-succinct
ly/ch07/testing/math (master)
$ go test -v
=== RUN   TestAverage
--- PASS: TestAverage (0.00s)
PASS
ok      github.com/marklewin/go-web-succinctly/ch07/testing/math        0.153s

mplew@LAPTOP-QVFF5MV9 MINGW64 ~/code/go/src/github.com/marklewin/go-web-succinct
ly/ch07/testing/math (master)
$
```

Figure 33: Executing the Test Successfully

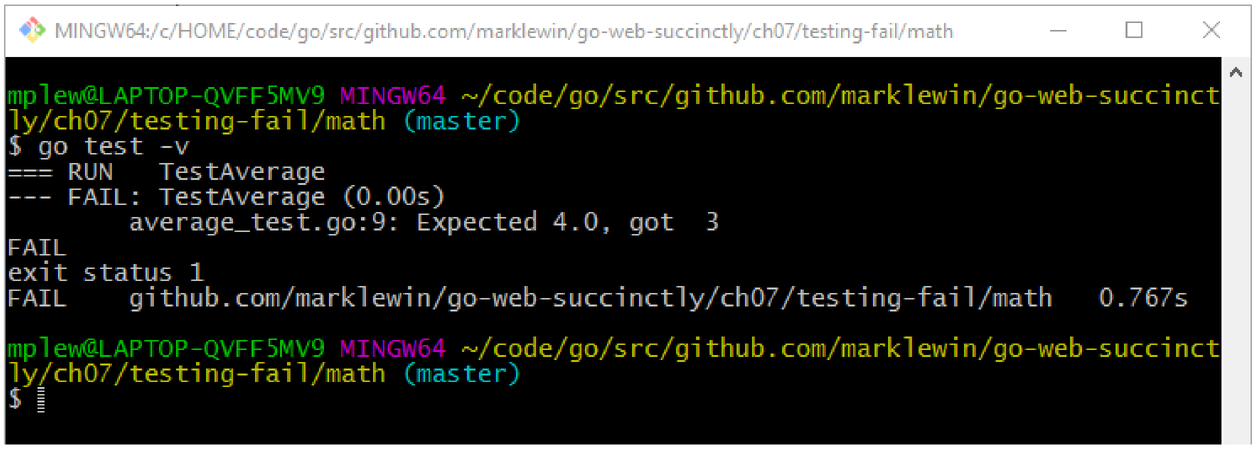
In this instance, all the tests in the **`math`** package passed (although currently there is only the one).

What if our test fails? Let’s simulate that. In this instance, it is our test, rather than the code we are testing, that’s at fault—it’s expecting a different value from the correct one.

Code Listing 47: Sabotaging the TestAverage Function

```go
package math

import "testing"

func TestAverage(t *testing.T) {
	var v float64
	v = Average([]float64{1, 2, 3, 4, 5})
	if v != 4.0 {
		t.Error("Expected 4.0, got ", v)
	}
}
```

Figure 35 demonstrates what a failed test looks like.


MINGW64:/c/HOME/code/go/src/github.com/marklewin/go-web-succinctly/ch07/testing-fail/math

```
mplew@LAPTOP-QVFF5MV9 MINGW64 ~/code/go/src/github.com/marklewin/go-web-succinct
ly/ch07/testing-fail/math (master)
$ go test -v
=== RUN   TestAverage
--- FAIL: TestAverage (0.00s)
        average_test.go:9: Expected 4.0, got  3
FAIL
exit status 1
FAIL    github.com/marklewin/go-web-succinctly/ch07/testing-fail/math   0.767s

mplew@LAPTOP-QVFF5MV9 MINGW64 ~/code/go/src/github.com/marklewin/go-web-succinct
ly/ch07/testing-fail/math (master)
$
```

Figure 34: A Failed Test

Hopefully, you won’t experience too many failing tests during your Go web development adventures, but better to know in development rather than in production!

Whatever happens, I wish you a very happy and productive time as a Go web developer!