Learn to Create Real World Web Applications using Go

by: Jonathan Calhoun

Web Development with Go

Learn to Create Real World Web Applications using Go

Jonathan Calhoun

Contents

About the author

Jon Calhoun is a software developer and educator. He is also a co-founder of EasyPost (easypost.com), an API that helps companies integrate with shipping APIs, where he also attended Y Combinator, a startup incubator. Prior to that he worked as an engineer at Google and earned a B.S. in Computer Science from the University of Central Florida.

Copyright and license

All source code in the book is available under the MIT License. Put simply, this means you can copy any of the code samples in this book and use them on your own applications without owing me or anyone else money or anything else. My only request is that you don't use this source code to teach your own course or write your own book.

The full license is listed below.

The Book Cover

The book cover was created by my brother, Jordan Calhoun, and was inspired by the Go Gopher by Renee French, which is licensed under Creative Commons Attributions 3.0.

Chapter 1

Introduction

Welcome to Web Development with Go: Learn to Create Real World Web Applications using Go!

Web Development with Go is designed to teach you how to create real world web applications from the ground up using the increasingly popular programming language Go (aka Golang) created by the kind folks at Google. That means that this book will take you from zero knowledge of web development to a level that is sufficient enough to deploy your very first web application.

1.1 Who is this book for?

Web Development with Go is for anyone who ever had an idea and thought "I wish I could build that." The book is for anyone who has visited a website and wondered "How does this work?". It is **NOT** just for computer science students, but instead is intended for anyone who has ever wanted to build a web application and share it with the world.

While this book will provide a ton of value to veteran developers, it was designed to be accessible for beginners.

The only real requirement is that you are vaguely familiar with Go and are willing to learn. That's it.

1.2 How to use this book

While I have attempted to make this book accessible for beginners, there is a lot of material covered. Not only will we be writing a lot of Go code, but we will also be using HTML, CSS, SQL, Bootstrap, and the command line. That is a lot to take in all at once, and you likely won't remember it all after one reading.

My advice is to go through the book once stopping as little as possible. Your goal here isn't to understand everything in detail, but to just get a broad understanding of what all the pieces in a web application are and how they work together. You also want to code along with everything in the book so you can get familiar with writing Go code. This means no copy-pasting!

After your first pass, I would then recommend going through the book a second time. This time your goal is to try to gain a deeper understanding of everything, using the higher level understanding you gained in the first pass as a foundation to build on.

If you would like, you could also use this second pass to attempt to build a slightly different application than we build in the book while using the book as a guide. For example, you might try to create a simplistic Twitter clone where users can sign up, post tweets, and follow other users. This will force you to really challenge how well you understand the material.

1.3 Reference materials

While reading this book you are likely to come across some things you are unfamiliar with that you want to research further. To help aid you, I have created and continue to maintain a beginners guide with resources for diving deeper into Go, HTML, CSS, SQL, and the command line.

You can access the guide at: calhoun.io/beginners

If you are brand new to any of those technologies I would suggest first checking out the beginner guide and getting vaguely familiar with them. "Vaguely" is the keyword here. You *DO NOT* need to be an expert at any of the technologies I listed, but instead just need to be familiar enough that you can follow along as we use them in this course.

In addition to the beginners guide, you are also encouraged to join the Web Development with Go Slack to ask questions, create study groups, and learn with other students. This has proven to be a vital resource, especially amongst newcomers to programming.

1.4 What if I am already an expert developer?

Even if you are already familiar with programming, web development, or Go, this book is likely to be a great reference for years to come.

You may not benefit from reading it start to finish, but you will most certainly find sections that provide insights and ideas that you never considered before.

To assist in this, I have also provided access to the code used in the book after each section is completed. This means you can easily jump to the section you want to reference, get the code, and follow along without having to complete the entire book.

1.5 What are we building?

Web Development with Go takes an hands-on approach to teaching web development. You won't be reading about theoretical web applications. We won't be talking about imaginary situations. Instead, we will be building a photo gallery application (shown in Figure 1.1) that you will deploy to a production server at the end of the book.

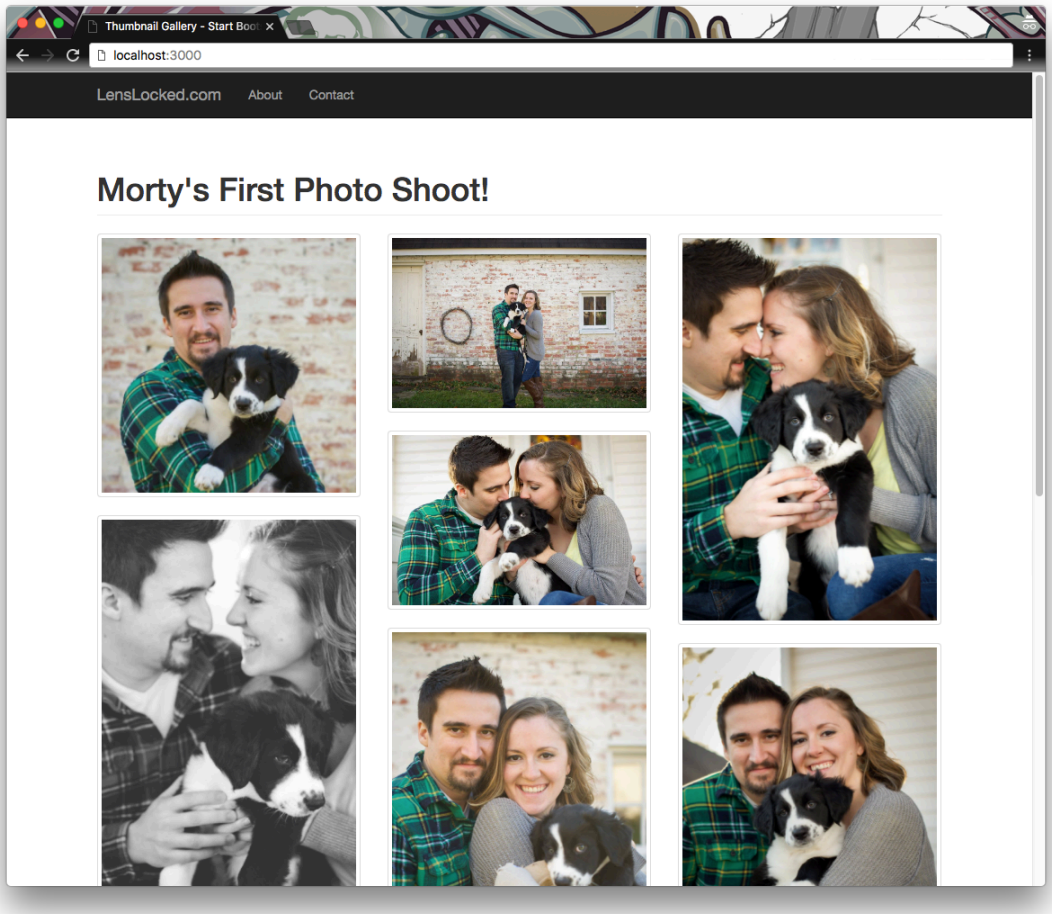

Figure 1.1: A picture of the web application we will build

In the application we build, users can sign up for an account, create galleries, and then upload images to include in each gallery. Then once a gallery has been created, they can send a link to their clients/friends to share the gallery.

Our application will start off incredibly simple; it will be a single "Hello, world" page. As we progress through the book we will slowly introduce new pages, improve our code, and tackle each new problem as it arises.

With this approach we will eventually arrive at a production-ready web app, but more importantly you will understand why we made each decision we made and will have a better idea of how to make those decisions on your own moving forward.

In short, we will be building our application as if you were learning on your own, stumbling through the documentation and making mistakes, but you won't have to blunder through them alone. I will be holding your hand and guiding you the entire time explaining everything along the way.

1.6 Conventions used in this book

Below are a few conventions used in this book that you should be aware of as your read it.

1.6.1 Command-line commands are prefixed with a `$`

Throughout this book I will provide you with some commands to run in the command line. For clarity, all of these commands will be prefixed with a Unix-style command line prompt, which is a dollar sign.

Listing 1.1: Unix-style prompt example

```
$ echo "testing"
testing
```

In Listing 1.1 the command you should type into your terminal is `echo "testing"` while the second line that reads "testing" represents the output from the terminal.

1.6.2 The `subl` or 'atom' command means "open with your text editor"

I will frequently use the command `atom` in my command line code listings to represent "opening a file with a text editor."

On my computer, running `atom hello.go` will open a file named "hello.go" in Atom, the editor that I use to write my Go code.

You are welcome to open files and create new ones however you wish, but I often depict them this way because it is the clearest way I know of representing it in a book.

1.6.3 Many code listings will be shortened for clarity

Throughout this book there will be many examples where we are working within a larger file but only need to make a slight change to a line or two. When this happens, showing a code listing of the entire file would be both confusing and a waste of space.

As a result, I will often only show relevant portions of the code and will use comments to represent unchanged code. These comments will often begin with `...`, and might look like Listing 1.2

Listing 1.2: A simple example of unchanged code

```
func main() {
  // ... everything before this remains unchanged
  var name string
```

```
  fmt.Println("What is your name?")
  fmt.Scanf("%s", &name)
}
```

It is important that you **do not** replace existing code with these comments. If you are ever uncertain, I suggest checking out the complete code for that section.

1.6.4 All code samples are limited to 60 columns when possible

In the screencasts I often write code that is wider than 60 columns, but in the ebook formats this doesnt' tend to work very well. As a result, I have attempted to limit all code samples to 60 characters or less per line.

You are welcome to alter your own code as you see fit. I personally think 60 characters per line isn't enough, but this was the simplest solution to the problem of writing for multiple book formats.

1.7 Accessing the code

The code for this book is provided as a git repository, with a branch for each chapter. You can download the code from each branch without any knowledge of git, but you will need an account at gitlab.com (*NOTE: This is different than github.com*)

Upon purchasing this course you should have received an email asking you to create an account at members.usegolang.com

When you log into your account there, you will find instructions for requesting accessing the code using your Gitlab account.

Once you have access to the repo, you will be able to follow links provided at the end of each section with the completed code for that section.

NOTE: You can also view changed source code for almost every section by following the diff link at the end of the section. Typically any new code will be on a green line and any removed lines will be shown with a red background. Sometimes when we change a line of code it will show up as both red (removing the old version) and then again as green (adding the updated version). This is how most code diff tools work, so it is useful to become accustomed to.

1.7.1 Why git?

I often get asked why I used git for the code, and the short version is that in Go import paths ARE NOT relative to your code, but instead are relative to your GOPATH.

This means that for me to provide a single zip file with all of the code from the course, I would need to change import paths for each section of the book and it wouldn't end up matching the code you are writing.

Rather than introduce this confusion, I instead opted to use git branches with links to each branch at the end of each section.

Any suggestions or feedback on how to improve this are welcome :)

1.8 Disclaimer: Not everything is one size fits all

Throughout this book I am going to be showing you just one way that you could structure and organize code for a web application, and that one way I show you prioritizes simplicity and ease of understanding over almost everything else.

Over time, your personal applications will likely become more robust and com-

plex, and you might find yourself questioning and changing the structure we use here. That is to be expected.

The truth is, there is no single way to design code that fits everyone. What works for one team, in one specific situation won't always work for everyone else.

Rather than attempting to show you the *one design to rule them all*, I am going to show you what I feel is an easy to understand and modify design. This means we will often write more code in order to avoid more complicated design patterns, but the intent with this is to make sure you understand the code well enough that you can experiment and try new patterns on your own once you have finished the book.

1.9 Commenting your exported types

If you use `golint`, or if you use an editor which automatically runs it on your code you may see warnings throughout the book. These typically read something like "exported function XXX should have comment or be unexported." What this warning means is that you are exporting a function (making it available outside of the package), but you haven't provided proper comments explaining what the package does for other developers.

Production grade code should indeed have comments for all exported functions, but throughout this book we will occasionally write code that doesn't have proper comments. Instead I have opted to explain what each piece of code is doing in the book, and I will also provide a final version of the code with all of the proper comments. This allows me to avoid writing duplicate explanations of what each exported type or function is used for upfront, but you still have access to a properly commented version of the source code used in the book.

Chapter 2

A basic web application

To get started, we are going to build a very basic web application. The app is incredibly simple, and it is only 15 lines of code, but with it we can start going over the basics of how web applications work before diving too deeply into anything.

A lot of this code will get thrown away, and a lot of it won't make sense at first. That is okay, and you shouldn't get frustrated if this happens. Part of learning to program is trying different things and seeing what they do, how they change things, and discovering which approach you like best. The goal of this approach is to try to emulate that process.

2.1 Building the server

Now that you know what to expect, let's get started with the code. First, we need to navigate to our Go `src` folder and create a folder for our application.

Listing 2.1: Creating the application directory

```
$ cd $GOPATH/src
$ mkdir lenslocked.com
```

```
$ cd lenslocked.com
```

Now that we have a proper directory to work from we can create our first go file, `main.go`. Create the file and then open it up in your favorite editor. I will be using Atom[1], so I will represent creating files with a command like the one in Figure **??**.

Listing 2.2: Creating `main.go`

```
$ atom main.go
```

Once you have opened up `main.go`, write the code in Listing 2.3 into it. Don't worry if you don't understand everything just yet; we will go over it shortly.

Listing 2.3: Creating a simple web application

```
package main

import (
  "fmt"
  "net/http"
)

func handlerFunc(w http.ResponseWriter, r *http.Request) {
  fmt.Fprint(w, "<h1>Welcome to my awesome site!</h1>")
}

func main() {
  http.HandleFunc("/", handlerFunc)
  http.ListenAndServe(":3000", nil)
}
```

Finally, we run the app.

[1] https://atom.io/

Listing 2.4: Running `main.go`.

```
$ go run main.go
```

Voilà! Assuming you didn't have any errors, you now have a web app running on your computer. Open up your browser and check it out at the URL http://localhost:3000/. You should see something similar to Figure 2.1.

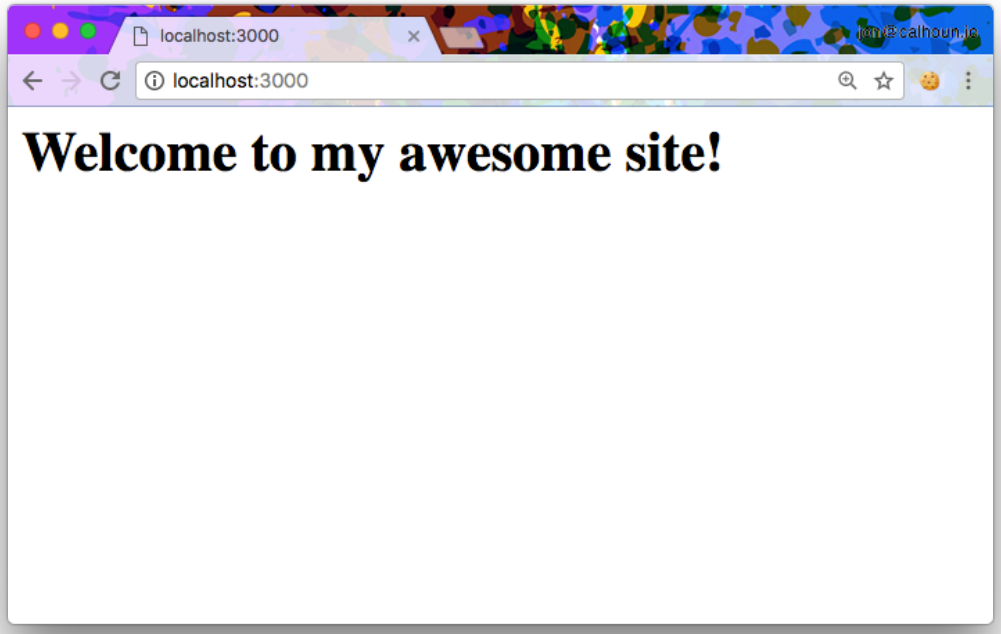

Figure 2.1: Our first web application!

After you have checked out your site you can shut down the server by hitting `ctrl+c` in the terminal where you ran `go run main.go`.

Box 2.1. Troubleshooting installation issues

At this point, some of you are going to experience an issue running your program. The most likely reason is that Go wasn't installed correctly.

My first suggestion is to practice your Google-fu and to try to resolve the issue on your own. I don't say this to be mean, but because part of becoming a developer is learning how to debug issues quickly.

If you can't figure it out from there, feel free to reach out via slack or email.

Source code

The link below has the completed source code for this section.

Completed - book.usegolang.com/2.1

2.2 Demystifying our app

Now you are probably asking yourself, what exactly is going on? To answer this question, we are going to go through our code one section at a time exploring what the code does. As we progress we will introduce new concepts and we will take a minute to explore each of them.

We won't go into quite this much detail for all of the code we write in this book, but for our first program I want to make sure you have a strong foundation to work from.

Starting with the first line...

```
package main
```

If you have written any code in Go, this should be pretty obvious. This simply declares what package this code is a part of. In our case, we are declaring it as part of the main package because we intend to have our application start by running the `main()` function in this file.

Next up are the imports:

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

`import` is used to tell the compiler which packages you intend to use in your code. As we expand on our application we will start to create our own packages so that we can test and reuse code more easily, but for now we are just using two packages from Go's standard library.

The first is the fmt package. Putting it simply, this is the package used to format strings and write them to different places. For example, you could use the `fmt.Println(...)` function to print "Hello, World!" to the terminal. We will cover how we used this in our code shortly.

The last package we import is net/http. This package is used to both create an application capable of responding to web requests, as well as making web requests to other servers. We end up using this library quite a bit in our simple server, and it will continue to be the foundation that our web application is built on top of.

Box 2.2. What are standard libraries?

Standard libraries are basically just sets of functions, variables, and structs written and officially maintained by the creators of Go. These generally tend to include code that is very commonly used by developers, such as code to print strings to the terminal, and are included with every installation of Go due to how frequently developers use them. You can think of them as code provided to make your life simpler.

That said, you don't have to use standard libraries to create a web server. If you really wanted to, you could go write your own **`http`** package and use it instead of the standard library for your web server, but that is a lot of work and you would be missing out on all of the testing and standardization provided by using the **`net/http`** package.

Next up we have the **`handlerFunc(...)`**.

```
func handlerFunc(w http.ResponseWriter, r *http.Request) {
  fmt.Fprint(w, "<h1>Welcome to my awesome site!</h1>")
}
```

This is the function that we use to process incoming web requests. Every time someone visits your website, the code in **`handlerFunc(...)`** gets run and determines what to return to the visitor. In the future we will have different handlers for when users visit different pages on our application, but for now we only have one.

All handlers take the same two arguments. The first is an **`http.ResponseWriter`**, declared as **`w`** in our current code, and a pointer to an **`http.Request`**, declared as **`r`** in our current code. While we don't currently use both of these in our code, it is worth going over what each is used now.

Box 2.3. A brief introduction to web requests

Whenever you click on a link or type a website into your browser, your browser will send a message to the web application asking for some specific page or set of data. This is called a web request. Once the server receives a web request, it will determine how to process it, and then send a response. The browser then decides how to show the resulting data - typically by rendering it as HTML.

Understanding the components of a web request and response aren't really required right now, but some of you will want to understand what is going on behind the scenes. So for all of you curious souls, a web request is a message to a web application specifying what type of data it wants in response. For our purposes we will be focusing on three parts of a web request - the URL the request is being sent to, headers, and a body.

The URL is something most people are familiar with. It is composed of a few parts, but the one we will be focusing on most is the path. This is the part after the website name. For example, given the url `http://www.lenslocked.com/signup` the path would be the `/signup` portion. We focus on this part of a URL because this is how we determine what the user is trying to do. For example, if the path is `/signup` then we know to run our code that handles users trying to sign up. If instead the path is `/news` we know to run our code that handles displaying news articles.

Headers are used to store things like metadata, cookies, and other data that is generally useful for all web requests. For example, after logging into your account many web applications store this data in a cookie, and then when you visit various pages of the website your browser includes this cookie in the headers of your requests. This allows the website to determine both that you are logged in, and which user you are.

The body is used to store user submitted data. For example, if you filled out a sign up form and hit the submit button, the browser would include the data you just typed into the form as part of the body so that the web application can process it.

Likewise, a response from a server is also broken into two parts - headers and a body. The response doesn't need a URL because it is simply responding to your request. Similar to requests, the headers are used to store mostly metadata that is useful to the browser, and the body contains the data that was requested.

First up is `w http.ResponseWriter`. This is a structure that allows us to modify the response that we want to send to whoever visited our website. By default `w` implements the `Write()` method that allows us to write to the response body, hence the name `ResponseWriter`, but it also has methods that help us set headers when we need to.

Next is `r *http.Request`. This is a structure used to access data from the web request. For example, we might use this to get the users email address and password after they sign up for our web application.

Box 2.4. What are pointers?

Pointers are exactly what they sound like - a data type that doesn't actually contain the data itself, but instead points to a memory address in a computer where the data is stored.

Pointers are used for many different reasons, but the primary thing to remember is that when you pass a pointer to a function it can alter the data you provided. This means that when we alter the request object we received, the changes will still be present after our code is done running.

If you are unfamiliar with pointers, you should probably take a moment to familiarize yourself with them before proceeding beyond this chapter, as they are a pretty important component of programming in Go.

Now that we understand the arguments, let's look back over the one line of code in our handler.

```
fmt.Fprint(w, "<h1>Welcome to my awesome site!</h1>")
```

Earlier we discussed the `fmt` package, which is collection of functions useful for formatting and printing strings. On this line we use the `fmt.Fprint()` function in this package and use it to write the HTML string `"<h1>Welcome to my awesome site!</h1>"` to the `http.ResponseWriter`.

Box 2.5. Explaining `fmt.Fprint()` in more detail.

The more complex version of this is that `fmt.Fprint` takes in two arguments:

1. An `io.Writer` to write to.
2. Any number of `interface{}`s to print out. Typically these are strings, but they could be any data type.

An `io.Writer` is an interface that requires a struct to have implemented the `Write([]byte)` method, so fmt.Fprint helps handle converting all of the provided interfaces to a byte array, and then calls the Write method on the `io.Writer`.

Since we are writing a string, and strings can be treated as byte arrays, you could replace the line `fmt.Fprint(w, "<h1>Welcome to my awesome site!</h1>")` with `w.Write([]byte("<h1>Welcome to my awesome site!</h1>"))` and you would end up getting the same end result.

If anything here is still confusing, don't worry about it and continue to press on. As you get more experience with the Go and programming in general things like this will start to make more sense, but until you gain a higher level understandings of Go it will be very hard to comprehend details like this.

Box 2.6. What are interfaces?

Interfaces in Go are a way of describing a set of methods that an object needs to implement for it to be valid. For example, I might have a function that takes in a parameter, let's call it a `Book`, and when showing the book on the website we call `book.Price()` to show the price. But what happens when we want to list a toy on our website? Our function only accepts the `Book` type!

It turns out that our function really doesn't care if is passed a `Book`, a `Toy`, or even a `Computer`. All it really cares out is that whatever it is passed has a `Price()` method that it can use to display the price. This is what interfaces are good for - they allow us to say what type of methods we care about, and as long as the object we are passed has those methods our code works.

You don't need to worry about writing your own interfaces for now. We will go over those in more detail when they come up in the book.

If you are coming from another language, like Java, it is worth noting that interfaces in Go are quite different what you are used to. Primarily, in Java it is required to explicitly state that an object implements an interface, but that is not required in Go. If an object implements all of the methods of an interface, it is considered an implementation of the interface without explicitly stating that it is.

Finally we get to the main function.

```
func main() {
  http.HandleFunc("/", handlerFunc)
  http.ListenAndServe(":3000", nil)
}
```

First, we declare the function `main()`. This is the function that will be run to start up your application, so it needs to call any other code that you want to run.

Inside of the main function we do two things. First we set our `handlerFunc` as the function we want to use to handle web requests going to our server with the path `/`. This covers all paths that the user might try to visit on our website, so you could also visit http://localhost:3000/some-other-path and it would also be processed by `handlerFunc`.

Lastly, we call `http.ListenAndServe(":3000", nil)`, which starts up a web server listening on port 3000 using the default http handlers.

Box 2.7. Port 3000 and localhost.

If this is your first time seeing things like `localhost` and `:3000` they might seem confusing at first, but don't worry! They are actually pretty simple to understand.

In computer networking, the term `localhost` was created to mean "this computer". When we talk about visiting http://localhost:3000/ in your browser, what we are really telling the browser is "try to load a web page from this computer at port 3000".

The port comes from the last part of the URL; The `:3000` portion. When you type `www.google.com` into your browser you don't have to include a port because the browser will use a default port automatically, but you could type it explicitly if you wanted. Try going to http://www.google.com:80.

We don't have to use port 3000 locally. In fact, many developers use port 8080 for local development. I simply opted to use port 3000 because it is what I am used to.

Chapter 3

Adding new pages

A web application would be pretty boring with only one page, so in this section we are going to explore how to add new pages to a web application.

We will start off by adding a contact page where people can find information on how to get in touch with us. After that we will add a catch-all page (often called a 404 page) that we will show when someone goes to a page that we haven't specified.

Box 3.1. HTTP Status Codes.

When your web server responds to a request, it also returns a status code. For most web requests your server will return a 200 status code, which means that everything was successful. When something goes wrong that was a result of bad data or a mistake in the client, a 400-499 status code is returned. For example, if you try to visit a page that doesn't exist a server will return a 404 status code, which means that the page was not found.

A 404 page gets its name from its HTTP status code. When a user attempts to visit a page that doesn't exist your application should return a 404 status code, and

if the user was requesting an HTML page it should render a page telling the user that you couldn't find the page the user was looking for. For now we are going to use a basic 404 page, but remember that over time a lot of your users may see this page when they make a typo or any other mistake, and it is a great page to make an impression on them. You can see several great examples of 404 pages here.

3.1 Routing with if/else statements

In order to add new pages to our web application we need to first discuss routing. At a very high level, routing is just a mapping of what page the user is trying to visit and what code we want to handle that request. For example, if you visit lenslocked.com you will be directed to the homepage, but if you visit lenslocked.com/faq you will instead be shown the FAQ. Both of these requests end up going to the same web application, but each are handled by a different piece of code inside that web application and the router's job is to make that happen.

To get started, let's try to write our own very basic routing logic. Open up **`main.go`** and update the **`handlerFunc`** with the code in Listing 3.1.

Listing 3.1: Routing via the URL path.

```
func handlerFunc(w http.ResponseWriter, r *http.Request) {
  w.Header().Set("Content-Type", "text/html")
  if r.URL.Path == "/" {
    fmt.Fprint(w, "<h1>Welcome to my awesome site!</h1>")
  } else if r.URL.Path == "/contact" {
    fmt.Fprint(w, "To get in touch, please send an email "+
      "to <a href=\"mailto:support@lenslocked.com\">"+
      "support@lenslocked.com</a>.")
  }
}
```

If your server is already running, you will need to stop it by pressing **ctrl + c** before you see the changes. After you stop the server, run it again.

```
$ go run main.go
```

Box 3.2. Dynamic reloading

If you are interested in learning about dynamic reloading, check out the Go library **fresh** - github.com/pilu/fresh

Once your server has restarted, head on over to localhost:3000/contact and you will see your contact page. Then head to localhost:3000 where you will see your home page.

What happens if you go to another path? Try it out - navigate to a path we haven't defined, like localhost:3000/something. You should see a blank page. That seems odd.

When a user visits a page we haven't defined it is a best practice to return a 404 status code and let the user know that we couldn't find the page they were looking for. To do this, we are going to update our handler function by adding an additional else statement. The updated code is shown in Listing 3.2.

Listing 3.2: Creating a 404 page

```
func handlerFunc(w http.ResponseWriter, r *http.Request) {
  w.Header().Set("Content-Type", "text/html")
  if r.URL.Path == "/" {
    fmt.Fprint(w, "<h1>Welcome to my awesome site!</h1>")
  } else if r.URL.Path == "/contact" {
    fmt.Fprint(w, "To get in touch, please send an email "+
      "to <a href=\"mailto:support@lenslocked.com\">"+
      "support@lenslocked.com</a>")
  } else {
```

```
    w.WriteHeader(http.StatusNotFound)
    fmt.Fprint(w, "<h1>We could not find the page you "+
      "were looking for :(</h1>"+
      "<p>Please email us if you keep being sent to an "+
      "invalid page.</p>")
  }
}
```

Now if we restart the server and visit a page that doesn't exist we will get an error message like the one shown in Figure 3.1.

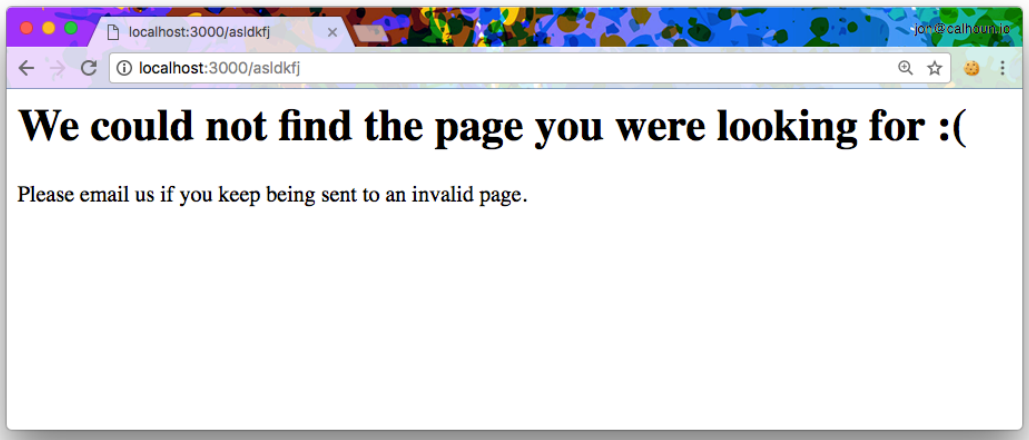

Figure 3.1: Our new 404 page

Before we add any new code, let's quickly review what is happening in our code.

First we get the URL from the `Request` object, which returns a `*url.URL`. This struct has a field on it named `Path` that returns the path of a URL. For example, if the URL was `http://lenslocked.com/docs/abc` then the Path would be `/docs/abc`. The only exception here is that an empty path is always set to `/`, or the root path.

Once we have the path we use it to determine what page to render. When the user is visiting the root path (`/`) we return our Welcome page. If the user is visiting our contact page (`/contact`) we return a page with information on how to contact.

If neither of these criteria are met, we write the 404 HTTP status code and then write an error message to the response writer. The StatusNotFound variably is really just a constant representing the HTTP status code 404, and the WriteHeader method is one way to write HTTP status codes in Go.

Box 3.3. Exporting common constants

As I stated above, `http.StatusNotFound` is really just a constant that represents the `404` status code. This isn't actually necessary, and you could replace it with `404` in your code, but constants like StatusNotFound are often exported by packages to make code easier to read and maintain.

Source code

The links below have the completed source code and a diff of all the code changed in this section.

Completed - book.usegolang.com/3.1

Changes - book.usegolang.com/3.1-diff

3.2 Popular routers

Now that we have some basic routing in place we are going to look into ways to improve and simplify our application. Specifically, we are going to explore open source routers and choose one of those to use with our project.

3.2.1 net/http.ServeMux

The first router we are going to look at is part of the standard library in Go. net/http.ServeMux is a pretty straightforward router, it is incredibly easy to use, and it works great for simple web applications.

Unfortunately this package is a little lacking in features, and many of these are features that we would ultimately need to write on our own. For example, the `http.ServeMux` doesn't provide an easy way to define URL parameters which is something we will utilize in later chapters.

While we could ultimately wrap this router in our own type and add all of the functionality we need, I didn't want to spend the entire book focusing on one specific part of a web application and instead opted to use another option.

3.2.2 github.com/julienschmidt/httprouter

github.com/julienschmidt/httprouter is a very popular router and one that I have seen used in several other tutorials and web frameworks. It is used in the popular web framework Gin, and I have even written a few blog posts myself using this router.

What I love about this router is its strong focus on being simple, fast, and efficient. If you check out some of the benchmarks on its Github page you will notice that it outperforms many other routers. On top of this focus on being

fast, it also supports named URL parameters and routing based on the HTTP request method used. We will learn more about this in a future chapter.

Ultimately I opted not to use `httprouter` for this book for two reasons. The first is that the library requires you to write http handler functions that take in a third argument if you want to access any path parameters. There isn't anything wrong with this approach, but I was concerned that it might lead to some confusion with new programmers and opted to instead use the router we are going to cover next - `gorilla/mux.Router`.

3.2.3 github.com/gorilla/mux.Router

Finally, we are going to check out github.com/gorilla/mux.Router. This router supports everything that we will need throughout this book (and much more), and it also allows us to work with regular old http handler functions.

On top of that, `gorilla/mux` is a part of the Gorilla Toolkit, a popular set of packages for building web applications. That in itself isn't a reason to use it, but since we will also be using a few other packages from the Gorilla Toolkit it is an added bonus that we are also using their router.

3.2.4 What about <some other router>

There are several other great routers out there that I don't mention here and don't use in this book. Many of them are great, and might even be better than gorilla/mux, but at the time of writing this gorilla/mux was the best choice available to me. Not only is it one of the most popular routers in Go, it has also been around for a very long time with a strong track record of stability.

3.3 Using the gorilla/mux router

The first thing you are going to need to do is install the package. Up until now we have only used standard packages, so as long as you have Go installed you have them installed. Stop your go application if you have it running (**ctrl + c**) and type the code in Listing 3.3 inside of your terminal.

Listing 3.3: Installing **gorilla/mux**.

```
$ go get -u github.com/gorilla/mux
```

This will download the package, and the **-u** option will tell Go that you want to get the latest version (in case you have an older version already installed).

Box 3.4. If you see an error...

At this point you might run into an error because you do not have git installed. If that happens, I recommend installing git and restarting to see if that resolves the issue.

https://git-scm.com/book/en/v2/Getting-Started-Installing-Git

Next we need to get our code ready. While we could continue handling multiple pages with a single function like we did in Section 3.1, this would be very hard to maintain.

Instead, we are going to break each of our pages into its own function so that we only need to tell our router which function to call depending on which page the user visits. The code for this is shown in Listing 3.4 and should be added to your **main.go** file.

Listing 3.4: Breaking up our handlers into functions.

```
func home(w http.ResponseWriter, r *http.Request) {
  w.Header().Set("Content-Type", "text/html")
  fmt.Fprint(w, "<h1>Welcome to my awesome site!</h1>")
}

func contact(w http.ResponseWriter, r *http.Request) {
  w.Header().Set("Content-Type", "text/html")
  fmt.Fprint(w, "To get in touch, please send an email "+
    "to <a href=\"mailto:support@lenslocked.com\">"+
    "support@lenslocked.com</a>.")
}
```

The code we wrote in Listing 3.4 should look very familiar because it is very similar to the code we wrote in our original **`handlerFunc`** function, minus the 404 page logic. Now that we have that code broken into two functions, we can delete the **`handlerFunc`** function from **`main.go`**.

That leaves us ready to start using **`gorilla/mux`** in our code. First we need to add an import at the top of our source code.

Listing 3.5: Importing **`gorilla/mux`**.

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

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

All we did here was add the line telling our code that we will be using the gorilla/mux package. When importing third party libraries you will be importing them using a url like the one we just used. This is typically a link to a source control service, such as github, but you will later see us using the lenslocked.com domain for private packages that we create.

Now that you have the **`mux`** package imported we can use it in the main function. The code in Listing 3.6 will replace the code currently in your main

function.

Listing 3.6: Using `gorilla/mux` for the first time.

```
func main() {
  r := mux.NewRouter()
  r.HandleFunc("/", home)
  r.HandleFunc("/contact", contact)
  http.ListenAndServe(":3000", r)
}
```

Now restart your application (`ctrl + c` followed by a `go run main.go`) and we should have a working webpage. Awesome!

If you visit a page that doesn't match any of your defined routes you might also notice that we have a 404 page that we didn't define. gorilla/mux provides a simple (albeit ugly) 404 page out of the box, as this is the behavior most developers expect from a router.

The new code inside of our main function can basically be broken into three stages. First we create a new router, then we start assigning functions to handle different paths, and finally we start up our server.

The last time we called the ListenAndServe function we passed `nil` in as the last argument. This time we are passing in our router as the default handler for web requests. This tells the ListenAndServe function that we want to use our own custom router.

Our router will in turn handle requests long enough to figure out which function was assigned to that path, and then it will call that function.

Source code

The links below have the completed source code and a diff of all the code changed in this section.

Completed - book.usegolang.com/3.3

Changes - book.usegolang.com/3.3-diff

3.4 Exercises

Congrats, you have built your first web app with multiple pages and you have even used a third party library!

Now let's practice what you learned by completing the following exercises.

Box 3.5. Our first practice problems.

Going forward I will start adding a few challenges after every chapter that will require you to use most of the knowledge you learned in that chapter.

You aren't required to complete these, but I definitely recommend giving each a shot before moving on to test your understanding of the content in each chapter.

3.4.1 Ex1 - Add an FAQ page

This one is pretty straight forward. Try to create an FAQ page to your application under the path `/faq`.

You can fill the page with whatever HTML content you prefer, but you should make it different from the other pages you are certain your code is working as you intended.

3.4.2 Ex2 - Custom 404 page

I mentioned earlier that gorilla/mux has a 404 page by default for paths we don't define with our router. You can actually customize this page by setting the NotFoundHandler attribute on the gorilla/mux.Router.

If you are new to Go this exercise is likely going to prove to be challenging because you will need to create an implementation of the http.Handler interface and then assign that to the NotFoundHandler, and this is a little different from what we have done so far.

To help with this, I have provided an example of how to convert the `home` function that we wrote earlier in this chapter into the http.Handler type.

```
var h http.Handler = http.HandlerFunc(home)
r := mux.NewRouter()
// This will assign the home page to the
// NotFoundHandler
r.NotFoundHandler = h
```

You will want to do something similar, but using your own unique 404 page function.

3.4.3 Ex3 - [HARD] Try out another router

This exercises is labeled *hard* because it is a little open ended.

Check out another router, like github.com/julienschmidt/httprouter, and try to replicate the program we have written so far using this router instead of gorilla/mux.

This is a great way to both (a) ensure you understood what we were doing, and (b) practice reading docs and using other libraries you are unfamiliar with.

Chapter 4

Plus many more chapters...

This is just a sample, so the rest of the book has been omitted. You can find out more about this book at the website - usegolang.com - along with a detailed outline of all 17 chapters included in the book.