# Learn to Create Real World Web Applications using Go

by: Jonathan Calhoun

Version 2

# Web Development with Go

Learn to create real, complex web applications

Jon Calhoun

# Contents

# Chapter 1

# Getting Started

We are going to get started with the course by building a basic web application, learning how it works, setting up your developer tools, getting advice on troubleshooting, and more.

## 1.1 A Basic Web Application

We are going to build a basic web application that is approximately 15 lines of code.

In future lessons we will explore what all of this code does, we will get familiar with the `net/http` package, and more, but for now our goal is just to make sure your tooling is working. To verify that we can build Go code, that our editor is properly set up, and that we are ready to proceed with the course. To achieve this goal, we are going to write some code without much explanation. Again - we will come back to what it does. Just follow along and make sure things are working for now.

It is also worth noting that almost all of this code will eventually be thrown away. The truth is, this is how development often works. We read docs, write experimental code, and try to understand our problem space a bit more. Once we have a better understanding we can step back and refine things - whether it by via design docs, refactoring code, or some other means.

Don't let this discourage you. It is very normal to write code, then later realize how it could be improved. Very few people write "perfect" code in their first pass, and more often than not getting something that works is far harder than refactoring it to be a bit cleaner.

Let's start by creating a directory for our code. I am going to use the terminal and name my directory `lenslocked`.

```
cd <path where you want your code>
mkdir lenslocked
cd lenslocked
```

**Box 1.1. Using the Terminal**

I will use the terminal quite a bit in this course, and you might see me refer to as the `console` from time to time. I use the terms interchangeably.

Throughout the course you will see me using a separate terminal program, but you should be able to use whatever tooling you are most comfortable with.

Now we need to open the directory in our editor. I am using VS Code, so for me this looks like:

```
# Open the entire directory I am in with VS Code
code .
```

This may be different if you are using Goland or some other editor.

Next I want to create and open a **`main.go`** file that we can add some code to. I will again do this via terminal, but you can do it however you want.

```
code main.go
```

Now we need to add code to our **`main.go`** file. Remember that we will explain this all later. For now we just want to make sure things are working. Still, I encourage you to write the code by hand (except for the imports) as this will help you retain what we are learning a bit better.

```go
package main

// May be provided automatically by your editor, so they may be removed if you
// hit save right after typing this bit. If your editor does support automatic
// imports you do NOT need to type this in.
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)
  fmt.Println("Starting the server on :3000...")
  http.ListenAndServe(":3000", nil)
}
```

Now we want to try to run the code. I will be doing this via the console, but again you can run it with your IDE if you are using one and that is easier for you.

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

Now open your internet browser and head to <localhost:3000>. If all went according to plan, you should see a web page with “Welcome to my awesome site!” in a somewhat large font.

You can stop the server by pressing `ctrl+c` in the terminal where you ran the code.

## 1.2 Troubleshooting and Slack

*If you are familiar with how to debug your code, you can likely skip this lesson.*

Hopefully your code is working, but at some point you may run into an issue. When that happens, you need to know the correct way to troubleshoot, request help, and more.

The issue could be a typo in your code, a breaking change in a library, or maybe Go isn't set up correctly. Regardless of the issue, my first suggestion is to try to resolve the issue on your own. You may need to practice your Google-fu, or you may need to get better at reading error messages, but in the end developing these skills will help you improve as a developer.

Another benefit to trying to debug on your own is that it will improve your chances of getting help from someone else. I have seen countless examples of new programmers asking for help without ever taking any time to try to debug on their own, and it can frustrate the developers who are helping you because it feels like you aren't putting in any effort.

Lastly, by debugging you will likely get better help; being able to explain what you have tried will often help others understand what you are trying to achieve, which in turn makes it easier for them to help.

Let's look at a quick example and I'll walk you through how I would try to debug it. Below is an example error message.

```
./main.go:11:6: no new variables on left side of :=
```

The first thing you should look for are the parts that tell you where in your specific code the issue is occurring. In this case it is saying the error is in the file `main.go` on line `11`. The `6` has a meaning too - I believe it refers to the byte on line `11` where the issue occurs - but I don't personally find that to be very helpful 99.9% of the time.

Once you know what file and line the error is on, you should start by examining the code there and seeing if the rest of the error message makes sense. Here is the line in our code:

```
err := doStuff()
```

At this point the error might be clear to you. When you use `:=` in Go it expects a new variable on the left side. If you are using a variable that already was declared, you will see this error message. In short, `err` was declared earlier in the program with code like `var err error`, `err := ...`, or something similar.

Now let's imagine you didn't know that. How could you proceed?

The next thing I would do is try copy/pasting the exact error message into a search engine, removing the parts specific to your code, and putting quotes around the rest. In this case that would give us:

```
"no new variables on left side of :="
```

When Googling this error you might find something helpful online. In this particular case that search query will likely lead you to ErrorsInGo.com, a website I happened to created at one point in the past. On the site there is an explanation of what causes this error as well as advice on how to fix it.

**Compare your code with the course code**

In this particular course you also the original source code to compare to. If you are ever stuck, consider grabbing the code in the course repo and comparing your code. If it isn't exactly the same, try to look at every difference and figure out what difference is causing the issue.

If both your code and the course repo look the same, try running the course repo source code to see if it gives you the same error. If it doesn't, then that means something is different and you haven't spotted it yet. If the code in the course repo doesn't work, that is a good sign that something outside of the code is misconfigured.

**Box 1.2. Accessing the Course Code**

You may need to request access to the code used in this course. Long term there will be a way to do this on the courses dashboard (courses.calhoun.io), but short term you may need to ask in Slack until this gets updated.

Almost all debugging is about ruling out potential sources for the problem. The steps may change from project to project, but the basic idea is the same.

**Slack and Community Help**

Another option for help troubleshooting is community help. This course has a private Slack where you can join and ask questions of myself and other students who have taken the same course.

**Box 1.3. Accessing Slack**

You can request access to Slack on the courses website at https://courses.calhoun.io/slack/invite

To get the most out of Slack you should:

- Post questions to a public channel (eg #webdevwithgo) so others can help and learn from the question. There is a very good chance that if you have a question, someone else will benefit from hearing the answer.

- Don't @tag me; I know many people want to do this to get my attention, but I promise I read all of the messages in Slack. I discourage students from tagging me because it can make a question seem less inviting to others, and the goal of the Slack is to build a community where students can also help and learn from one another. That doesn't happen if questions aren't open for everyone to discuss.

- Try to share the smallest piece of code that replicates your issue. Go playground links are highly encouraged because they force you to get rid of everything that isn't relevant to the issue or question at hand. This takes more effort on your part, but vastly simplifies things for anyone trying to help you and thus makes it more likely you will receive help.

- If you can't share a Go playground link, you can share your source code, but try to make it easier for others to help. Make the repository public so anyone can access it without requesting an invite. Gists that don't include all the code are hard to debug, and private repos are tricky for other students to help with because each student has to request access to see the code.

- Try to help others! Helping will reinforce what you are learning, and it can keep you motivated when you recognize how much you are learning.

In addition to the course slack, I would also suggest you join the Gophers Slack - https://invite.slack.golangbridge.org/

## 1.3 Packages and Imports

Now that we have a working program and our tooling is working, we are going to work our way through the code we wrote and try to understand it all. Some parts will require additional background information, so the process of explaining the app is broken into a few lessons.

Our Go code starts with the `package` keyword. This defines the package our code is part of.

```
package main
```

Packages are used to group code that is related. For instance, we might group a set of functions and types used for processing images into an `image` package, or we might have a `zip` package for compressing and decompressing files.

Having too many packages with barely any code in them can be unproductive, so early on it isn't uncommon to put all of your code in a single package. We will be doing this as well, then as our codebase grows we will start to create additional packages to organize our code.

The `main` package is a special package that tells our program where to start. When our code is built, it will look for a `main()` function inside of the `main` package and build a binary that starts there. If you want to create a program that you can run, you will need a `main` package with a `main()` function.

If you explore other projects, you may find multiple `main` packages. These will often be nested inside of a `cmd` directory, similar to below.

```
some-app/
  cmd/
    server/
      main.go # package main
    demo/
      main.go # package main
  blah.go
  foo.go
```

Using the **cmd** directory to store various potential binaries is a common pattern in Go. Each directoy - **server** and **demo** in our example - has its own **main** package that is used to build an entirely unique Go program. In other words, the code above can generate at least two different programs - server and demo.

Every individual program only has one **main** package. What you are seeing in the example above is a project with several programs in it.

Getting back to our code, the package is followed by imports.

```go
package main

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

Imports tell our program what other packages we wil be using. In this case we want to use some code from the **fmt** and **net/http** packages. Both of these are included in the standard library.

The **fmt** package is useful to print things. If you have written a “Hello, world” program in Go, there is a good chance you used the **fmt** package.

The **net/http** package has utilities for setting up web servers and receiving web requests, as well as code for making your own web requests to other services.

## 1.4 Editors and Automatic Imports

While not a requirement for writing Go code, I highly recommend setting up an editor with a Go integration. This will give you things like:

- Intelligent auto-completion
- Support for custom auto-complete templates
- Automatic importing of packages you use in your code
- Error highlighting
- Looking up definitions
- And more

At some point in the course I may present you with code to add to your application and forget to mention the required imports. If you have automatic imports set up, this shouldn't be a problem. On the other hand, if you do not have automatic imports you may need to reference the source code for the coruse to verify that you aren't missing an import.

If you are in need of an editor suggestion, hear are two to check out:

1. VS Code with the Go extension. See here for more info.
2. GoLand by Jetbrains. See here for more info.

Both of these options provide tons of functionality, extensions/plugins, and more. The primary difference between the two is that VS Code is closer to just a text editor with some helpful intelligence, while GoLand feels more like an entire development environment (aka an IDE).

Another key difference is the extensions available. VS Code has Github Copilot, which can offer some incredible autocompletion suggestions.

I prefer using a lightweight text editor, and then to use separate applications for my terminal and other tools. As a result, you will see me using VS Code but not using the integrated terminal. This is mostly a byproduct of how I learned, and shouldn't be taken as the best way to do things. It is mostly a personal preference, and you should determine what you prefer on your own.

If you are a fan of emacs or vim there are also ways to set them up with Go, but if you are using either of those I am going to assume you can figure that out on your own.

**Box 1.4. Focus on One Thing at a Time**

I do not recommend trying to learn multiple things at a time. For instance, you shouldn't try to learn both vim and Go at the same time. You will have much better results if you focus on one thing at a time. If you are unfamiliar with all code editors, VS Code is incredibly quick to setup and learn well enough that it won't hinder you as you go through this course.

## 1.5 The “Hello, world” Part of our Code

If we take a minute to look at the rest of our web application, there is a good chance you have seen code similar to part of it. Specifically, if you have ever written a “Hello, World” program, it likely looked very similar to the following code from our web application.

```go
package main

import (
  "fmt"
)

func main() {
  fmt.Println("Starting the server on :3000...")
}
```

The `main` function is where our program starts when we run it. The `fmt.Println` line is printing a message out to our terminal, and the `ln` part at the end means to add a new line at the end. This will cause any new text written to the terminal to appear on the next line.

## 1.6 Web Requests

In order to grasp the rest of our code, we need to take a minute to learn about web requests.

Throughout the course you will see me interact with things like headers, paths, and more that are all related to a web request. At some point you may find yourself asking, "How did he even know what a header is, let alone where to look to set one?"

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

A web request can contain a wide variety of information. For instance, it can tell a server what type of data it wants (json, xml, or something else). A request can include headers with additional information about the user making the request, or the browser they are using.

In this lesson we will be focusing on three parts of a web request - the URL the request is being sent to, headers, and the 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 **`example.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 our server might recognize this as a request for a signup form. If the path was instead **`/login`** then we might render the form to sign into their account.

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 request body is used to store data related specifically to the request being made. For example, if you filled out a sign up form and hit the submit button, your browser would include the data you just typed into the form as part of the request body. When you upload files, they would also be attached as part of the request body.

Responses to a web request look very similar to a web request. They have no need for a URL, but both headers and a body are present in nearly all web request responses. Headers are similar to headers in a request - they store metadata, cookies your browser should set, similar data. The body will contain the data that was request, or it can be empty if no specific data was requested and instead an action - like deleting a resource - was requested.

In our Go code, the http.Request is what our browser sends to the sever when making a web request. It includes a URL, headers, and a body. When we want to write a response to the request, we use the http.ResponseWriter.

As you work with web applications more you will start to familiarize yourself with more details about web requests and responses. Over time you will also realize that Go does a pretty good job of aligning the types in the standard library with what a real web request and response look like.

## 1.7 HTTP Methods

In addition to URL, headers, and a body, web requests also have a method associated with them. These are often referred to as HTTP request methods, and are used to categorize what type of request is being made. For instance, if you go to your browser and type `google.com` in, your browser will make a web request with a `GET` method, which signifies that you just want to read data. In this case that data is the landing page for Google.

There are a number of HTTP methods, and as the internet evolves new ones are proposed. In this course we are going to focus on the four most common methods:

- `GET` - reading a resource
- `POST` - creating a resource
- `PUT` - updating a resource
- `DELETE` - deleting a resource

In the last lesson we discussed how our server might decide what to do based on the path of the request. For instance, if the path is `/login` vs `/signup` we know the two are requesting different things.

HTTP methods can also be combined with the path to help determine how we should respond to a request. Let's look at two path/method combinations to help clarify.

- **`GET /login`** - A web request with a GET method to the **`/login`** path suggests the user is trying to load the login form. We would typically want to return the HTML page that has the form.

- **`POST /login`** - The path is the same as before, but this time we have a POST method. What this signifies is that we are not reading the form, but instead are submitting it to create a new login session. Our server shouldn't return the login form HTML, and should instead try to process the login request.

As we progress through the course we will be learning more about HTTP methods, REST, and more. For now, the major takeaway should be that they are used to signify what type of request is being made, and our server will need to look at both the HTTP method and the path to determine how to respond.

## 1.8 Our Handler Function

When looking at how web requests work, we saw that there are two main components:

- A request

- A response

In Go, the way we handle web requests is going to reflect this reality. Functions to process web requests need to accept two arguments - a **`http.ResponseWriter`** and a **`*http.Request`**. The **`net/http`** package in the standard library even has a type for it - http.HandlerFunc.

```
// See https://pkg.go.dev/net/http#HandlerFunc
type HandlerFunc func(ResponseWriter, *Request)
```

We need to create a function that matches this pattern to handle incoming web requests. That leads to the following code in our application.

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

Our handler function could be named anything; I am choosing the name `handlerFunc` for now because it is our only handler function, but later on in the course when we have multiple handler functions we will use different names for each.

The `http.ResponseWriter` argument of our handler function is an interface that defines a set of methods we can use while creating a response to a web request. With it we can write a response body, write headers, set the HTTP status code, and more.

In our code we use the `ResponseWriter` along with `fmt.Fprint` to write the HTML response `<h1>Welcome to...</h1>`. `<h1>` is a header tag in HTML. Normally we might not write HTML directly like this, but to keep this example simple we are.

`fmt.Fprint` is similar to `fmt.Print` that you have likely seen in the past, except `Fprint` allows us to specify WHERE to write to. It is common to see `Fprint` used when writing to files, but we can also use it to write to an `ResponseWriter`. This works because `Fprint` accepts any implementation of the `io.Writer` interface as its first argument. If we look up the definition of this type, we see it is:

```
// See https://pkg.go.dev/io#Writer
type Writer interface {
  Write(p []byte) (n int, err error)
}
```

Any type with a `Write(p []byte) (n int, err error)` method implements the `io.Writer` interface, even if that type happens to be a larger interface like our `http.ResponseWriter`.

```
// See https://pkg.go.dev/net/http#ResponseWriter
type ResponseWriter interface {
  Header() Header
  // This method causes ResponseWriter to implement io.Writer
  Write([]byte) (int, error)
  WriteHeader(statusCode int)
}
```

If this feels a bit overwhelming, don't worry. Almost all of this takes time and practice before it really starts to sink in. For now, the main takeaway is that we use `fmt.Fprint` and the `ResponseWriter` to write an HTML response.

**Box 1.5. Interfaces**

If you would like to read more about interfaces, you can check out the following free resource: https://www.calhoun.io/crash-course-on-go-interfaces/

The second argument of our handler function - the `*http.Request` - is a pointer to an explicit struct type that defines an incoming web request.

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

With it we can read the request body, see the HTTP request method, view the path of the request, read headers provided by the request, access cookies included in the request, and more.

In our code we aren't reading anything from the request. We will use that later in the course, but for now we can leave it alone because we respond to all requests, regardless of what they are, with the same HTML.

## 1.9 Registering our Handler Function and Starting the Web Server

We have a function named `handlerFunc` that we want to use to handle incoming web requests, but we need to tell our Go program about it so it knows to use it. To do this we need to register the handler somehow.

In the future we will have multiple functions for handling different types of requests - like signing in, viewing a home page, or viewing a different page - and at that time we will need to signify which handler should be used in each situation using something called a router (or sometimes a mux). For now we are going to register a single handler for all incoming web requests. We do this with the following code:

```go
http.HandleFunc("/", handlerFunc)
```

If you look up http.HandleFunc in the standard library docs you will see it has the following definition:

```go
func HandleFunc(pattern string, handler func(ResponseWriter, *Request))
```

The first argument is a pattern that is used to match the incoming request path. If the pattern ends in a `/`, it will match anything that has that pattern as a prefix. In our case, we are using only a slash (`/`), which means this handler should be matched to any request that matches the empty prefix, which means it will match ALL request paths.

This information about pattern matching is available in the docs. If you go to http.HandleFunc in the docs it reads:

> … The documentation for ServeMux explains how patterns are matched.

If we follow this to the http.ServeMux docs we will find the following:

> Patterns name fixed, rooted paths, like "/favicon.ico", or rooted subtrees, like "/images/" (note the trailing slash). Longer patterns take precedence over shorter ones, so that if there are handlers registered for both "/images/" and "/images/thumbnails/", the latter handler will be called for paths beginning "/images/thumbnails/" and the former will receive requests for any other paths in the "/images/" subtree. Note that since a pattern ending in a slash names a rooted subtree, the pattern "/" matches all paths not matched by other registered patterns, not just the URL with Path == "/".

Getting back to our code:

```
http.HandleFunc("/", handlerFunc)
```

The second argument passed into `http.HandleFunc` is the function we want to handle requests that match our provided pattern. In our case we want the `handlerFunc` function that we created to handle the requests, so we pass it in here. It is important to note that we are passing the function itself in as an argument, and we are not calling the function using parenthesis `()`.

Behind the scenes, calling `http.HandleFunc` ends up isng the default `http.ServeMux`. This is mentioned in the docs, and we can see it if we examine the source code.

```
// HandleFunc registers the handler function for the given pattern
// in the DefaultServeMux.
// The documentation for ServeMux explains how patterns are matched.
func HandleFunc(pattern string, handler func(ResponseWriter, *Request)) {
  DefaultServeMux.HandleFunc(pattern, handler)
}
```

`DefaultServeMux` is a package global variable declared in the `net/http` package. (source code)

```
// DefaultServeMux is the default ServeMux used by Serve.
var DefaultServeMux = &defaultServeMux
```

**Box 1.6. Global Variables**

While global variables are often frowned upon, the `net/http` package provides the `DefaultServeMux` because it can drastically simplify basic web server setup. Without it, the basic web app we are learning from would need a few extra lines of code that would need further explaining. This is similar to how `fmt.Println` really uses `os.Stdout` behind the scenes - it is such a common use case that it makes sense for the standard library to simplify it. Later in the course when we learn more about routers and muxers we will move away from using the `DefaultServeMux` global variable.

Again, this is a lot of information, and truthfully all you really need to take away is that the line `http.HandleFunc` is registering our `handlerFunc` function to process all incoming web requests. The rest will come with time.

Going back to our original program, the next line in `main()` that we haven't discussed is the `http.ListenAndServe(...)` function call.

```
package main

// May be provided automatically by your editor, so they may be removed if you hit save right a
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)
  fmt.Println("Starting the server on :3000...")
  http.ListenAndServe(":3000", nil) // <----
}
```

`ListenAndServe` is how we start our web server. `:3000` is the port that our server will be started on, and it defaults to `localhost` - a network address for the computer we are on - if we don't provide anything else. What this means is our program will attempt to tell the operating system that any web requests coming in on port `3000` should be sent to the Go code we are running. That is why we need to type <localhost:3000> into our browser to see our page running - we have to explicitly tell our browser which port to use.

When we deploy we will make it so users don't need to provide a port, but in development it is common to use ports as you may have multiple web services running at the same time, and each needs its own port.

The last argument - `nil` in our case - is where you can pass in any `http.Handler` to handle web requests. If `nil` is passed in, it tells our code to use the `DefaultServeMux` that we mentioned earlier.

Again, if this doesn't make complete sense right now, don't get upset. I'm giving you a ton of information and some of it will take some time before it really starts to click. This is completely normal. As you build more web services in Go this will all start to make more sense, but for now it can feel overwhelming. Just keep pushing through the course and it will make more sense as you progress.

## 1.10 Go Modules

There are roughly two reasons to use Go modules:

1. Dependency management
2. Writing and building Go code outside of **`GOPATH`**

Dependency management is a way of making sure other developers who try to build our code also use the same libraries and versions that we used. Otherwise it is possible that a build will fail or have unexpected behavior because it isn't actually using the same code for every library.

We don't have any third party libraries yet, so this isn't a concern, but we will install some later so we will need Go modules for this.

Another benefit of Go modules is building outside of **`GOPATH`**. In the past all Go code needed to run from the same parent directory - the **`GOPATH`**. Unfortunately, this was somewhat confusing for some new developers. It was also limiting to others who didn't want to keep all of their code in the same directory for one reason or another.

For both of these reasons, we wil be using Go modules.

Go modules use Semantic Import Versioning (SIV). SIV is based on Semantic Versioning (SemVer), but it adds its own unique rules.

### 1.10.1 Semantic Versioning (SemVer)

Let's start by learning what SemVer is; SemVer is a way of versioning libraries and code. It is a version number composed of three parts:

- The major version
- The minor version
- The patch version

An example of a SemVer might be: `v1.23.45`. In this example, the major version is `1`, the minor is `23`, and the patch is `45`.

In theory, the way SemVer works is that every new release of your code results in a new version. The part of the version you increment depends on what type of change was made. If you made a minor bug fix that doesn't introduce breaking changes, you would likely increment the patch number. We might go from `v1.23.45` to `v1.23.46`.

If the change being made adds functionality in some backwards compatible way - that is anyone using the library now won't have to update their code for it to continue working as-is - then the minor version gets an update. We could go from `v1.23.45` to `v1.24.0`, resetting the patch number as the minor number increases.

Lastly we have major version updates. These are changes that introduce breaking changes. For instance, we might remove a function from the library, or we might change a function to accept a new argument. These are all breaking changes because anyone who was using the library before the change was made would need to update their code to use the new version. In this case a version like `v1.23.45` would become `v2.0.0`.

The benefit of SemVer is that we can upgrade libraries with a better understanding of when the updates will cause a breaking change. For instance, if we had a program using some imaginary `ffmpego` package to encode videos, and we were using `v1.11.4` we could upgrade to `v1.23.0` without worrying too much about our code breaking because only the minor version was updated. On the other hand, if `v2.0.0` released we would know that upgrading to this version might introduce breaking changes that we need to fix in our code.

I say "in theory" because it is up to developers to ensure each new release with the same major version number doesn't introduce a breaking change, and it isn't always clear what is or isn't a breaking change. As a result, it is possible for a library to make a mistake and introduce a breaking change in a minor version increase. Still, SemVer tends to work pretty well in practice.

**Box 1.7. Major Version Zero**

The major version `0` is a special version indicating that a library may introduce breaking changes. It is commonly used for libraries in a beta state, but it is also used by some libraries that don't care to deal with SIV.

Go modules helps us specify the major version of each library we use in our application so that future builds continue to work. There is more going on behind the scenes, but that is all you need to know at this point about dependencies.

*Related info: When building your own Go library, Go modules are useful for defining versions your own code, but given that we are building a web application that others won't be importing into their code this functionality isn't useful for us.*

## 1.10.2 Initialize a Go Module

Next we are going to set up our own Go module. After that, you won't need to think about modules too much aside from installing specific versions of libraries as the course continues.

To see go module commands, we run `go mod` in the terminal.

```
go mod
```

This gives us the following output.

```
Go mod provides access to operations on modules.

Note that support for modules is built into all the go commands,
not just 'go mod'. For example, day-to-day adding, removing, upgrading,
and downgrading of dependencies should be done using 'go get'.
See 'go help modules' for an overview of module functionality.

Usage:

  go mod <command> [arguments]

The commands are:

  download    download modules to local cache
  edit        edit go.mod from tools or scripts
  graph       print module requirement graph
  init        initialize new module in current directory
  tidy        add missing and remove unused modules
  vendor      make vendored copy of dependencies
  verify      verify dependencies have expected content
  why         explain why packages or modules are needed

Use "go help mod <command>" for more information about a command.
```

We want to initialize our module, so we will be using the `init` subcommand followed by the name of the module we are initializing. This is typically the import path, which is commonly something like `github.com/<username>/<pkg-name>`.

Make sure you are in the `lenslocked` directory with your code and run the following.

```
go mod init github.com/joncalhoun/lenslocked
```

You likely want to replace the `joncalhoun` part with your own username.

`lenslocked` is the name I'm giving to our web application. You can change it, but I'd suggest using the same name your first pass through the course to simplify things.

After running `go mod init`, you should find that it created both a `go.mod` file and a `go.sum` file. These are used to manage which versions of each imported third party library are used. From this point onwards if we run commands like **`go get`** or **`go build`** they will use or update the modules files as needed and we generally won't need to think too much about modules. If there is a time when you need to think about it, I will mention it.

I might also to encourage you to use a specific library version to ensure your code works as expected. It is highly recommended you use the same versions as you code along, otherwise there might be breaking changes. Typically, if a library is updated I'll add additional content at the end of the course explaining how to upgrade, and it is better to proceed through the course using the same version.

Now that our module is declared, we could run **`go install .`** and it would be installed with the name **`lenslocked`**. After that we could run our code by typing **`lenslocked`** into our terminal.

```
go install .
lenslocked # should start our server
```

Press **`ctrl+c`** to stop the server.

*If this does not work, there is a chance something isn't set with your environment variables.*

There might come a time when you want to clean up your modules. For instance, if you are no longer using a few libraries. You can run **`go mod tidy`** and this should handle tidying things up.

Lastly, in some editors like VS Code you need to open your code at the directory with the **`go.mod`** file for the Go tooling to work correctly. If you open another directory, autocompletion and other features may not work, or may use the incorrect version for some of your libraries.

# Chapter 2

# Adding New Pages

## 2.1 [Bonus] Dynamic reloading

This section isn't necessary to proceed, but dynamic reloading is a common request so I wanted to cover it before we proceed. Just remember that if you have an issue with the tooling here, you can always use `go run` and `go build` instead.

Dynamic reloading is basically the act of watching for changes in your code, and then automatically rebuilding and restarting your program once a change is detected. It can be helpful during development as it reduces the amount of steps you need to take between making a code change and seeing it live in your browser. Unfortunately, it can also lead to some issues.

The biggest issue is that breaking changes aren't always obvious. Depending on the tooling you use, some dynamic reloaders will keep the old version of your code running until a new build can be created. If your code has a compilation error in it, that means the old version will continue running and this can be confusing when you navigate to the browser and aren't seeing your changes.

Errors like this are typically displayed in the terminal, but developers can easily forget to check the terminal when using a dynamic reloading tool because they no longer need to head to the terminal to stop and restart their program.

I mention this to give one piece of advice - if you do opt to use dynamic reloading, always check the terminal to see if things have built correctly if anything seems off.

There are a few tooling options for dynamic reloading with Go:

- modd
- air

I will be using `modd` for the rest of this lesson.

**Box 2.1. modd and Go modules are not the same**

For anyone wondering, `modd` has nothing to do with Go modules. `modd` existed before Go modules and it is simply an unfortunate naming clash.

First we need to install modd. I suggest heading over to the website to review the most up-to-date installation instructions. As of this writing, the instructions say to head over to the modd releases page and to then download a binary for your operating system. After that you need to move it somewhere within your `$PATH`.

Given that this isn't a major part of the course, I am going to leave this part up to you. If you do get stuck, feel free to get on Slack and ask for help.

Once you have modd installed, verify it is working by checking your version.

```
modd --version
# You should see output with a version.
# Mine is: 0.8
```

Next we need to create a configuration file for modd to use. This will tell it what files to watch for changes, as well as what to do when a file changes.

```
code modd.conf
```

Add the following to the `modd.conf` file.

```
# modd.conf
**/*.go {
  prep: go test @dirmods
}

# Exclude all test files of the form *_test.go
**/*.go !**/*_test.go {
  prep: go build -o lenslocked .
  daemon +sigterm: ./lenslocked
}
```

When we run modd, this will tell it to do two things:

1. To watch for changes to any `.go` files, including test files, and to run tests for any changed directories.
2. To watch for changes to any non-test `.go` file and to build and restart our app if a change is detected.

Once your `modd.conf` file is saved, run modd.

```
modd
```

You should see output showing that there are no tests, and you should see your app start up. Now we can make changes and the server will automatically rebuild and restart when modd detects those changes. Try changing a line in your `main.go` file and verifying this works.

We won't be writing tests in this course, as I personally think they should be learned after you get the web development basics down, but I did want to provide you with a `modd.conf` that will work for tests in case you add them in the future.

Lastly, the second clause in the `modd.conf` file will create a binary named `lenslocked` in our directory, so if you are using git you will want to create a `.gitignore` file and add `lenslocked` to it so the binary isn't included in source control.

```
# Create the .gitignore file
code .gitignore
```

Then add the following

```
# Ignore the lenslocked binary in source
# code commits
lenslocked
```

If you want to further expand the `.gitignore` file, there are a few generators out there with common setups. For instance I will be using this one from Toptal, and I have added `lenslocked` to it.

## 2.2 Content-Type Header

Earlier in the course we learned that both a web request and a response can have headers. In this lesson we are going to look at one specific header - the `Content-Type` header.

Headers are often used to provide additional information about a request or response. While they aren't directly rendered, the content type header can provide information about what data is provided, and thus how it should be rendered. For instance, our application is currently returning HTML, so we might set our content type to `text/html`.

There are a wide variety of content types available. Below are a few common ones:

- `text/html`
- `application/json`
- `image/png`
- `video/mp4`

In many cases, the content type can be determined by the data stored in the body. This is what is happening with our application right now, and that is why the HTML is rendered correctly in our browser. If we were to explicitly set the content type to something else - like `text/plain` - the browser would render the body a bit differently.

Open up your `main.go` source file and navigate to our handler function. Add the following line to our code to set the `Content-Type` header.

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

Restart your code and head to <localhost:3000>. You should now see your HTML being rendered as plain text.

Change your code to set the content type to **`text/html`** so that our page works again.

```
w.Header().Set("Content-Type", "text/html")
```

Headers and their values are not unique to Go. You can set headers in much the same way regardless of the programming language your web server is written in. What is unique to Go is the exact code needed to set the header. It will be slightly different in every programming language.

Let's take some time to explore how I figured out how to set a header using the **`http.ResponseWriter`**.

First, I headed over to the **`net/http`** docs. From there I looked for the ResponseWriter type.

Once I found the type I was working with, I looked at the **`ResponseWriter`** interface and found the following:

```
// Header returns the header map that will be sent by
// WriteHeader. The Header map also is the mechanism with which
// Handlers can set HTTP trailers.
//
// ... (some comments are missing for brevity)
Header() Header
```

From there I was able to click on the **`Header`** type in the docs, which brought me to the http.Header docs. From here it was a matter of reading over the available functions and deciding which made the most sense.

Both **`Add`** and **`Set`** would have worked for our needs. I went with **`Set`** because this allows me to set a value for any specific header key. **`Add`** sounded more appropriate if I planned to add multiple values for a single header key.

While I do want you to become comfortable using the docs, I will admit that most people won't learn web development by reading docs like this. Most developers I have met tend to learn the basics by reading books, tutorials, taking courses, and looking at example code. The docs are great for filling in gaps and clarifying details. Keep them in mind and consider looking up any new types or functions you are using to help improve your understanding of them.

## 2.3 Contact page

Next we are going to add a new page to our application - the contact page. This is going to be a static HTML page and the code will be minimal, but that will be sufficient to help us learn how to create a new handler function, register it, and eventually we will use it as we learn about routers.

Open up `main.go` and add the following function.

```
func contactHandler(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:jon@calhoun.io\">jon
}
```

**Box 2.2. Escaping Quotation Marks**

The backslashes in the string are used to escape the quotation mark (`"`) character. Without them, the compiler would think we were ending our string at the wrong place, so we add them before quotation marks that we want included in the string.

By the end of the course we won't be creating HTML in strings like this. It is a temporary measure for now.

Next we need to register our handler function. We want it mapped to the **`/contact`** path, so we use the following inside our **`main()`** function:

```go
http.HandleFunc("/contact", contactHandler)
```

Restart your application (or let **`modd`** do it) and verify the contact page is working by heading over to <localhost:3000/contact>.

At this point our the name **`handlerFunc`** doesn't make as much sense. It was fine when it was our only handler function, but now that we have two it probably makes sense to update it. Given that this function is currently used for our home page, let's rename it to **`homeHandler`** and update the code accordingly.

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

// Update main to reflect the change
func main() {
  http.HandleFunc("/", homeHandler)
  http.HandleFunc("/contact", contactHandler)
  fmt.Println("Starting the server on :3000...")
  http.ListenAndServe(":3000", nil)
}
```

## 2.4 Examining the http.Request

When a user visits the home page of our application, our code calls the **`handlerFunc`** function. When a user visits the **`/contact`** path, our code runs the **`contactHandler`** function. The act of determining which code to run based on the request path is called routing.

While the standard library provides some routing functionality, it is fairly limited in scope. As a result, we will be using a third-party routing library in the future. In the meantime, we are going to look at the `http.Request` type to see how we might write our own routing logic so you have a better understanding of what is happening behind the scenes.

Head to the docs for the http.Request type and try to look for anything that might help you write your own routing logic. Does anything stand out?

For me, there are two fields that look promising:

1. Method
2. URL

The method will be useful when we start routing based on HTTP methods. For instance, a `GET /signup` will be handled differently from a `POST /signup`.

The URL should help us take care of the second part - the path. Click on the `url.URL` type in the docs and look at its documentation.

```
// Comments are excluded for brevity
type URL struct {
  Scheme      string
  Opaque      string
  User        *Userinfo
  Host        string
  Path        string
  RawPath     string
  ForceQuery  bool
  RawQuery    string
  Fragment    string
  RawFragment string
}
```

The `url.URL` type has a `Path` field that looks promising. Let's try printing it out to see what it gives us.

Head over to your `main.go` source file and add the following function. This will be the start of our own router, but for now it simply prints out the request path.

```
func pathHandler(w http.ResponseWriter, r *http.Request) {
  fmt.Fprint(w, r.URL.Path)
}
```

Next, alter your `main()` function to only use this handler.

```
func main() {
  http.HandleFunc("/", pathHandler)
  fmt.Println("Starting the server on :3000...")
  http.ListenAndServe(":3000", nil)
}
```

Now restart your web application and try a few paths.

- <localhost:3000/>
- <localhost:3000/signup>
- <localhost:3000/galleries/123/images>

Verify that in each case you see the path being printed out via the `Fprint` line we added. Note that if no path is included, the `Path` field defaults to a slash (`/`).

Great, now that we know how to determine the path of an incoming request we can use that to create our own basic router.

## 2.5 Custom routing

Now that we know how to programmatically determine the path of any incoming web request, let's try to add some logic to our code that uses this information. Open `main.go` and change the `pathHandler` function to reflect the code below.

```go
func pathHandler(w http.ResponseWriter, r *http.Request) {
  if r.URL.Path == "/" {
    homeHandler(w, r)
  } else if r.URL.Path == "/contact" {
    contactHandler(w, r)
  }
}
```

Restart the server and try navigating to a few paths. Try going to <localhost:3000> without any path. Then try the `/contact` path. Now try other paths - like `/signup`. What results are you seeing?

Our code is setup to send all incoming web requests to the `pathHandler` function. From there, our logic kicks in and decides how to proceed.

If the path matches `/`, we use the homeHandler to render the home page. If the path matches `/contact` we call the contactHandler function. If neither of those cases occur we do not do anything which results in the empty page being rendered.

Let's update our code in preparation for handling the invalid path case. Specifically, let's use a switch statement so it is easier to add new cases.

```go
switch r.URL.Path {
case "/":
  homeHandler(w, r)
case "/contact":
  contactHandler(w, r)
default:
  // some error message handler!
}
```

In a future lesson we are going to add a handler for when an invalid path is provided. This is commonly referred to as a Not Found page, and it even has an HTTP status code associated with it. As an exercise, try to explore the docs to look for this status code and anything else that might help you render a not found page.

Start by determining what status code you should be using. Next, look in the **`net/http`** package for a constant that might have this information. Finally, look for a function in the **`net/http`** package that helps render errors.

## 2.6 URL Path vs RawPath

If you took some time to explore the **`url.URL`** type, you might notice that it has both a **`Path`** and a **`RawPath`** field.

```
type URL struct {
  // ...
  Path        string
  RawPath     string
  // ...
}
```

Seeing both of these fields, you might stop and ask, "Which should I be using?" In this lesson, we are going to dive into the difference between the two.

When dealing with a URL, some characters have special meanings. For instance, the question mark (**`?`**) signifies the start of URL query parameters. These are values in the URL which provide additional data, but they aren't really part of the path. These exist in part because a GET request doesn't have a request body, so the only way to provide information about what you are requesting is to add it as a URL query parameter.

If you wanted to create a path that included a question mark in it, you would need to encode the character. You can try this yourself by heading over to urlencoder.com and typing in various characters to be encoded. Most won't need any special encoding, but a few will. The question mark character encodes to **`%3F`**.

When looking at Go's **`url.URL`** type, the **`Path`** field stores the path with values decoded for you. 99% of the time this is what you will want, as it allows you to skip all the extra work of decoding values and there is rarely ambiguity.

There is one case where ambiguity still exists; When looking at the `Path` field it is impossible to tell the difference between a path of `/cat/dog` and `/cat%2Fdog` which will both be represented as `/cat/dog` in the `Path` field.

99% of the time you won't care about this distinction, but if you happen to be building an application where this matters you will need to use the `RawPath` field to determine the difference. Keep in mind that the `RawPath` is only ever set if it is needed, so it is only set if an encoded slash (`/`, which encodes as `%2F`) is in the path.

We won't need to use RawPath at all in this course, but I wanted to point it out in case you were wondering how I knew which to use, or if you happened to use the wrong one and didn't understand the difference.

## 2.7 Not found page

With any application, users will eventually try to navigate to invalid paths. It may happen due to an invalid link, or it might be the result of a user trying to manually type in a URL and getting it wrong. In either case, we want to show them an error page. In this lesson we are going to see how to handle this while also learning a bit about HTTP status codes.

When responding a web request, status codes are a way of indicating the results of the request in a more machine-friendly way. For instance, when a server responds to a web request normally, the default status code is `200` which stands for `OK`. When a user tries to visit a page that doesn't exist, the `404` status code can be used, which stands for `Not Found`.

Normal people using a web browser rarely care about status codes. Instead, they will read the information on the webpage to determine what happened. If the page says, "Sorry, this page could not be found." they will likely interpret this to mean that they typed in the URL incorrectly, or perhaps they followed a broken link. HTTP status codes aren't as important for regular users because they have the HTML on the page to tell them what is going on.

On the other hand, machines can't simply look at an HTML page and automatically determine what happened. Every website might represent an error differently, using different HTML, different text, and possibly even a different language. HTTP status codes exist to make it easier for tools like browsers and API clients to determine the result of a web request.

Status codes are numbers that range from 100 to 599. According to the MDN docs, these are typically broken into the following categories:

- Informational responses (`100–199`)
- Successful responses (`200–299`)
- Redirection messages (`300–399`)
- Client error responses (`400–499`)
- Server error responses (`500–599`)

Status code meanings are RFCs like rfc 2616. Not all of the numbers from 100 to 599 have a meaning assigned to them, and it is also common for some APIs to assign special meanings to some status codes and then describe these in their docs. For instance, Stripe's docs have some special status codes and if a user deteces one of these it can be a signal that they should be looking for an error response rather than whatever they originally expected.

In this course we will be using some of the common and universally accepted status codes. A few of these common ones are:

- `200` = `OK`, which means everything with the request went well.

- **`404`** = **`Not Found`**, which usually means a page or resource requested. doesn't exist. It can sometimes be returned when you don't have access to a resource (perhaps because you aren't logged in), and the web server doesn't want to acknowledge for sure whether the resource exists to those without access. Gitlab does this at times.

- **`500`** = **`Internal Server Error`**, which is a generic catch-all error that means something unexpected happened on the server, and it is likely due to an issue on the server.

There are many more HTTP status codes, and these are all available as constants in the net/http package as constants.

**Box 2.3. Status Code Easter Egg**

Some HTTP status codes are nonsensical or not commonly used. For instance, **`418`** means **`I'm a teapot`** and is mostly an easter egg. I've yet to encounter a scenario where my server magically became a teapot.

We will use a few more status codes throughout this course. When they come up, I will explain what they are and why we are using that particular status code.

Right not we want to utilize the **`404`** status code. This is the one referenced when a user tries to visit a page that does not exist, much like the case with our router in the **`default`** case. It is available in the **`net/http`** package as:

```
const (
  // ...
  StatusNotFound                    = 404 // RFC 7231, 6.5.4
  // ...
)
```

To set a status code, we need to look at the **`http.ResponseWriter`** type. Reading the docs there are two things to note:

1. If we call **`Write`** without setting a status code, the **`200 OK`** status code will be set by default.
2. If we want to set our own status code, we need to call **`WriteHeader`** before calling **`Write`**.

Head to your **`main.go`** source file and add the following to the **`default`** part of the **`pathHandler`** function's **`switch`**.

```
default:
  w.WriteHeader(http.StatusNotFound)
  fmt.Fprint(w, "Page not found")
}
```

Restart your server and try to navigate to a page that doesn't exist, like <localhost:3000/invalid/path>. You should see the "Page not found" string on the page, and if you open your developer tools in your browser you should be able to see the 404 status code. This is typically visible in Chrome's network tab. You may need to reload after opening up the developer tools.

The **`net/http`** package also provides us with an **`http.Error`** function to help render errors like this. While we might want to use a custom page long term, it is a nice little helper when getting started.

Head back to your **`pathHandler`** function and replace the code we added with the following code:

```
http.Error(w, "Page not found", http.StatusNotFound)
```

## 2.8 http.Handler type

When we call `http.HandleFunc` the `DefaultServeMux` is used behind the scenes.

```go
// Don't code this. It is from the net/http source code
func HandleFunc(pattern string, handler func(ResponseWriter, *Request)) {
  DefaultServeMux.HandleFunc(pattern, handler)
}
```

Using the DefaultServeMux like we are isn't going to be problematic, but let's take a moment to look at what we would need to change in our code to avoid using the DefaultServeMux.

If you look at the `http.ListenAndServe` function, you will notice that the second argument can be an `http.Handler`.

```go
// Don't code this. It is from the net/http source code.
func ListenAndServe(addr string, handler Handler) error
```

According to the docs, if we pass `nil` in as the second argument the Default-ServeMux is used, otherwise the handler we pass in will be used. That means we want to somehow pass our `pathHandler` function into ListenAndServe so it is used directly.

You might try something like the code below, but this code will error because our `pathHandler` function doesn't match the `http.Handler` type.

```go
// This will error, but you can try it in your main.go
http.ListenAndServe(":3000", pathHandler)
```

***Warning***: *Throughout the rest of this lesson, and in the next few lessons, we are going to do a deep dive on several types and functions in the* `net/http` *package. I will try to explain this all as clearly as I can, but it is a lot of information* *to take in at once. It can also be hard to understand until you have used some of these types a few times. I suggest following along with the next few sections and learning what you can for now, then returning to review the material at a later date once you have a bit more experience. If it doesn't all click right now, that is okay. Don't let that stop you from moving on with the course.*

Looking back at our code, why can't we pass the **`pathHandler`** function in as the second argument to ListenAndServe?

To answer this we first need to look at what the second argument to ListenAndServe is supposed to be - an **`http.Handler`**. According to the **`net/http`** docs, this is an interface with a **`ServeHTTP`** method.

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

While the **`ServeHTTP`** method is very similar to our **`pathHandler`** function, the Handler type is an interface and we can't simply pass in a function. Instead, we need to create a type that has the **`ServeHTTP`** method. Let's give that a try. Open up **`main.go`** and create a **`Router`** type with a **`ServeHTTP`** method.

```
type Router struct {}

func (router Router) ServeHTTP(w http.ResponseWriter, r *http.Request) {
  switch r.URL.Path {
  case "/":
    homeHandler(w, r)
  case "/contact":
    contactHandler(w, r)
  default:
    http.Error(w, "Page not found", http.StatusNotFound)
  }
}
```

The code inside of our **`ServeHTTP`** method is the exact same code we used in the **`pathHandler`** function. In fact, we could have called **`pathHandler`** inside of **`ServeHTTP`** if we wanted.

We can now use the Router type as an http.Handler when we call ListenAndServe. Head down to the `main()` function and change it to use the following code.

```
func main() {
  var router Router
  fmt.Println("Starting the server on :3000...")
  http.ListenAndServe(":3000", router)
}
```

A common question at this point is, "Why would I want a Router struct instead of using a handler function like our pathHandler?"

Using your own struct type can be incredibly useful because it allows us to set fields in the struct that can later be used when handling HTTP requests. For instance, we could create a Server type with a database connection that is available for each handler to use. A fake example of this is shown below.

```
// You do not need to code this. It is for illustrative purposes only.
type Server struct {
  // DB would normally be a *sql.DB or similar, but
  // using a string for demonstration purposes here.
  DB string
}

func (s Server) ServeHTTP(w http.ResponseWriter, r *http.Request) {
  // We can now access s.DB to make database queries!
  // In this particular example we are just writing out the value of
  // s.DB to demonstrate how we would access it.
  fmt.Fprint(w, "<h1>" + s.DB + "</h1>")
}
```

We could now use the `Server` type to spin up multiple servers, each with its own database connection. This could be useful for testing, allowing each test case to interact with its own isolated database.

We will see more benefits of using a struct type as an http.Handler later in this course. For now you will have to trust me when I say there are several benefits to using types that implement an interface.

## 2.9 http.HandlerFunc type

At this point our code is entirely self-reliant for routing. We aren't even using the ServeMux from the standard library. We achieved this by implementing the **`http.Handler`** interface with our **`Router`** type, and then passing an instance of that type into the **`ListenAndServe`** function.

In this lesson we are going to explore how we might achieve similar results with a handler function, like our **`pathHandler`** function, rather than a struct type.

If we head back over to the **`net/http`** docs, we can find another type there named **`http.HandlerFunc`**. While this might seem a bit odd at first, this is a type with a function as the underlying type.

```
type HandlerFunc func(ResponseWriter, *Request)
```

In Go you can declare a type that is backed by a struct, which is what most of us are familiar with. This is what our **`Router`** type was. In addition to creating struct types, you can also create a function type.

As weird as it sounds, a function type is treated the same as any other types. Most importantly, it can have methods just like a struct type. As a result, the **`http.HandlerFunc`** type is able to declare a **`ServeHTTP`** method where it calls itself.

```
func (f HandlerFunc) ServeHTTP(w ResponseWriter, r *Request) {
  f(w, r)
}
```

**Box 2.4. First Class Functions**

Languages that allow developers to use functions like any other variable or data type are said to support first class functions. Go allows developers to do all of this and even allows developers to create new types using functions, so it supports first class functions. There are languages where this type of behavior isn't allowed and if you try using one you will find that different techniques may need to be used at times.

Let's look at this with a real example, then we can see how it all works. Head to **`main.go`** and update the **`main()`** function with the following code.

```go
func main() {
  var router http.HandlerFunc
  router = pathHandler
  fmt.Println("Starting the server on :3000...")
  http.ListenAndServe(":3000", router)
}
```

In this code we first declare the **`router`** variable with the **`http.HandlerFunc`** type. We saw earlier that a **`HandlerFunc`** is really any function that takes in a **`ResponseWriter`** and a **`Request`**, so we can assign our **`pathHandler`** function to this variable. From there we can pass the **`router`** variable into **`http.ListenAndServe`** because the **`HandlerFunc`** type implements the **`ServeHTTP`** method.

Inside the **`ServeHTTP`** method, the **`HandlerFunc`** is called.

```go
func (f HandlerFunc) ServeHTTP(w ResponseWriter, r *Request) {
  f(w, r)
}
```

What this ultimately ends up doing is calling the **`pathHandler`** function that we assigned as the HandlerFunc.

In our code we currently declare a variable, assign **`pathHandler`** to it, and then finally use that variable in the **`ListenAndServe`** function call. While this works, we could actually reduce this to a single line.

```go
func main() {
  fmt.Println("Starting the server on :3000...")
  http.ListenAndServe(":3000", http.HandlerFunc(pathHandler))
}
```

While `http.HandlerFunc` looks like a function call at first glance, it is actually a type conversion. It is similar to converting a float into an integer:

```go
func abs(a int) int {
  // math.Abs expects a float64 input and returns a float64
  // when we do float64(a) we convert the integer to a float64
  r := math.Abs(float64(a))
  // when we use int(r) we convert the float64 to an integer.
  return int(r)
}
```

We aren't making function calls here, we are converting a type to another type. `http.HandlerFunc` is a type, so when we write the following code we are converting `pathHandler` to the `http.HandlerFunc` type.

```go
http.HandlerFunc(pathHandler)
```

In summary:

1. The `HandlerFunc` type implements the `Handler` interface
2. We can convert our handler functions into `HandlerFunc`s
3. When a function is converted into a `HandlerFunc`, it has a `ServeHTTP` method which implements the `Handler` interface.

This is all made even more confusing due to the similarity between `http.HandleFunc` and `http.HandlerFunc`! The only difference is a single `r` character!

Here are the big things to remember:

There are two main types in the `net/http` package that we are looking at right now:

- `http.Handler` - an interface with a ServeHTTP method
- `http.HandlerFunc` - a function that accepts the same args as a ServeHTTP method

There are also two functions in the `http` package that look similar, and are used to register their respectively similar types for a web server:

- `http.Handle` - a function that accepts a pattern and an `http.Handler` as its arguments.
- `http.HandleFunc` - a function that accepts a pattern and a function that looks like a `http.HandlerFunc` as its arguments.

Go provide both functions to make life easier for developers like you and I. They both ultimately end up using the same code behind the scenes. That is, `http.HandleFunc` ultimately converts its arguments into an `http.Handler` behind the scenes using the `http.HandlerFunc` type. We will explore this in the next lesson.

## 2.10 Bonus: Exploring the Handler Conversion

In this lesson we will explore how everything is eventually converted into an `http.Handler`.

Let's start by looking at the code we initially had to setup our routes:

```
func main() {
  http.HandleFunc("/", homeHandler)
  http.HandleFunc("/contact", contactHandler)
  fmt.Println("Starting the server on :3000...")
  http.ListenAndServe(":3000", nil)
}
```

In this code we didn't convert any functions to an `http.HandlerFunc`, nor did we use the `http.Handler` interface, so how did everything end up working?

To answer that question we first need to look at the `http.HandleFunc` source code.

```
// From the net/http package
func HandleFunc(pattern string, handler func(ResponseWriter, *Request)) {
  DefaultServeMux.HandleFunc(pattern, handler)
}
```

This leads us to the `DefaultServeMux` variable, which has the type `ServeMux`. If we look up the `ServeMux.HandleFunc` source code we find the following.

```
func (mux *ServeMux) HandleFunc(pattern string, handler func(ResponseWriter, *Request)) {
  if handler == nil {
    panic("http: nil handler")
  }
  mux.Handle(pattern, HandlerFunc(handler))
}
```

It took a few steps, but we can now see where the type conversion is happening. The handler function being converted into the `HandlerFunc` type and then passed into the `ServeMux.Handle` method which expects an `http.Handler` as its second argument.

**Box 2.5. Missing http Prefix**

You don't see the **`http`** prefix here because this code is inside the **`net/http`** package.

This conversion is the same as the one we wrote with the following code.

```
http.ListenAndServe(":3000", http.HandlerFunc(router))
```

By writing the **`ServeMux`** type this way, the Go team was able to keep all of the real logic inside the **`ServeMux.Handle`** method.

```
func (mux *ServeMux) Handle(pattern string, handler Handler) {
  mux.mu.Lock()
  defer mux.mu.Unlock()

  if pattern == "" {
    panic("http: invalid pattern")
  }
  if handler == nil {
    panic("http: nil handler")
  }
  if _, exist := mux.m[pattern]; exist {
    panic("http: multiple registrations for " + pattern)
  }
  // ...
}
```

It doesn't matter if we call **`http.HandleFunc(...)`**, **`http.Handle(...)`**, or if we create a **`ServeMux`** and use the methods on it. All of these eventually end up using the **`ServeMux.Handle`** method.

When writing code, I tend to use the function that is most convenient to me at the time. If I am trying to register a route and I have a type that implements the **`http.Handler`** interface like our **`Router`** does, then I use **`Handle`** function.

```
var router Router
http.Handle("/", router)
```

On the other hand, if I have a function like our **`pathHandler`** and I want to register it, I will frequently use the **`HandleFunc`** function.

```
http.HandleFunc("/", homeHandle)
```

If I really wanted to, I could always use one of the two. For instance, the following is valid code and in which we only use **`HandleFunc`**. This code works because **`ServeHTTP`** is a function that matches what **`HandleFunc`** expects.

```
var router Router
// I am passing in the ServeHTTP function here, not the router
http.HandleFunc("/", router.ServeHTTP)
http.HandleFunc("/contact", contactHandler)
```

It likely doesn't make sense to do this, but I simply wanted to point out that it does work.

Moving forward, the general guidelines to remember are:

- **`Handle`** is used to register anything that implements the **`http.Handler`** inteface.
- **`HandleFunc`** is used to register functions that look like an **`http.HandlerFunc`**.
- You can convert back and forth using the **`net/http`** package as needed.

## 2.11 Exercises

Nothing helps reinforce something better than exercises. Only one for this section.

### 2.11.1 Ex1 - Add an FAQ page

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.

#### Solution

First, let's add a handler for the FAQ page.

```
func faqHandler(w http.ResponseWriter, r *http.Request) {
  w.Header().Set("Content-Type", "text/html; charset=utf-8")
  fmt.Fprint(w, `<h1>FAQ Page</h1>
<ul>
  <li>
    <b>Is there a free version?</b>
    Yes! We offer a free trial for 30 days on any paid plans.
  </li>
  <li>
    <b>What are your support hours?</b>
    We have support staff answering emails 24/7, though response
    times may be a bit slower on weekends.
  </li>
  <li>
    <b>How do I contact support?</b>
    Email us - <a href="mailto:support@lenslocked.com">support@lenslocked.com</a>
  </li>
</ul>
`)
}
```

This introduces a multi-line string in Go using the backtick. This is handy when we want to write a string across several lines in our source code.

Long term, it would be nice to not have to repeat the `w.Header().Set(...)` bit, but we will handle that later in the course.

Next, add a case to our router.

```
case "/faq":
  faqHandler(w, r)
```

Finally, visit <localhost:3000/faq> to see it in action.