Learn to Create Real World Web Applications using Go

by: Jonathan Calhoun

Web Development with Go

Learn to build real, production-grade web applications from scratch.

Jonathan Calhoun

Contents

Course Resources **v**

0.1 Ebooks, Slack, Source Code, and more v

1 Getting Started **1**

1.1 A Basic Web Application 1

1.2 Troubleshooting and Slack 4

1.3 Packages and Imports 8

1.4 Editors and Automatic Imports 10

1.5 The “Hello, world” Part of our Code 11

1.6 Web Requests 12

1.7 HTTP Methods 14

1.8 Our Handler Function 15

1.9 Registering our Handler Function and Starting the Web Server 18

1.10 Go Modules 22

1.10.1 Semantic Versioning (SemVer) 22

1.10.2 Initialize a Go Module 24

2 Adding New Pages **29**

2.1 Dynamic Reloading 29

2.2 Setting Header Values 33

2.3 Creating a Contact Page 35

2.4 Examining the http.Request Type 37

2.5 Custom Routing 40

2.6 URL Path vs RawPath 41

2.7 Not Found Page 42

2.8 The http.Handler Type 46

2.9 The http.HandlerFunc Type 50

2.10 Exploring Handler Conversions 54

2.11 FAQ Exercise 58

2.11.1 Ex1 - Add an FAQ page 58

3 Routers and 3rd Party Libraries **61**

3.1 Router Requirements 61

3.2 Using Git 64

3.3 Installing Chi 65

3.4 Using Chi 68

3.5 Chi Exercises 71

3.5.1 Ex1 - Add a URL Parameter 71
3.5.2 Ex2 - Experiment with Chi's builtin middleware . . . 72

4 Templates **73**

4.1 What are Templates? 73
4.2 Why Do We Use Server Side Rendering? 76
4.2.1 JavaScript Frontends Add Complexity 78
4.2.2 Server-side Rendering Works Well 79
4.2.3 Our App Doesn't Need a to be a SPA 80
4.3 Creating Our First Template 81
4.4 Cross Site Scripting (XSS) 89
4.5 Alternative Template Libraries 94
4.6 Contextual Encoding 95
4.7 Home Page via Template 98
4.8 Contact Page via Template 104
4.9 FAQ Page via Template 107
4.10 Template Exercises 109
4.10.1 Ex1 - Use template variables 109
4.10.2 Ex2 - Experiment with different data types 110
4.10.3 Ex3 - Learn how to use nested data 110
4.10.4 Ex4 - Create an if/else statement in your template . . . 111

5 Code Organization **113**

5.1 Code Organization 113

5.1.1 Flat structure 114

5.1.2 Separation of concerns 115

5.1.3 Dependency based structure 116

5.1.4 Plus many more... 118

5.2 MVC Overview 121

5.3 Walking Through a Web Request with MVC 123

5.4 MVC Exercises 128

5.4.1 Ex1 - What does MVC stand for? 129

5.4.2 Ex2 - What is each layer of MVC responsible for? . . 129

5.4.3 Ex3 - What are some benefits and disadvantages to using MVC? 129

5.4.4 Ex4 - Read about other ways to structure code 129

6 Starting to Apply MVC **131**

6.1 Creating the Views Package 131

6.2 fmt.Errorf 137

6.3 Validating Templates at Startup 140

6.4 Must Functions 146

6.5 Exercises 150

6.5.1 Ex1 - Add a new static page to your app 151

6.5.2 Ex2 - Experiment a bit with errors 151

7 Enhancing our Views **153**

7.1 Embedding Template Files 153

7.2 Variadic Parameters 160

7.3 Named Templates 164

7.4 Dynamic FAQ Page 168

7.5 Reusable Layouts 176

7.5.1 Approach #1: Header and Footer Templates 176

7.5.2 Approach #2: Named Page Template 179

7.6 Tailwind CSS 183

7.7 Utility-first CSS 188

7.7.1 Utility vs Component CSS 196

7.8 Adding a Navigation Bar 197

7.9 Exercises 204

7.9.1 Ex1 - Experiment with Tailwind 204

7.9.2 Ex1 - Create a Footer 204

7.9.3 Ex3 - Explore Go's embed package 204

8 The Signup Page **205**

8.1 Creating the Signup Page 205

8.2 Styling the Signup Page 210

8.3 Intro to REST 216
8.4 Users Controller 220
8.5 Decouple with Interfaces 223
8.6 Parsing the Signup Form 229
8.7 URL Query Parameters 233
8.8 Exercises 239
8.8.1 Ex1 - Experiment With Form Fields 239
8.8.2 Ex2 - Adjust Your Paths 239

9 Databases and PostgreSQL 241
9.1 Intro to Databases 241
9.2 Installing Postgres 246
9.3 Connecting to Postgres 255
9.4 Update: Docker Container Names 259
9.5 Creating SQL Tables 260
9.6 Postgres Data Types 263
9.7 Postgres Constraints 264
9.8 Creating a Users Table 265
9.9 Inserting Records 269
9.10 Querying Records 272
9.11 Filtering Queries 274

9.12 Updating Records 275
9.13 Deleting Records 276
9.14 Additional SQL Resources 277
9.14.1 Learning Resources 277
9.14.2 SQL Tools 278

10 Using Postgres with Go **279**
10.1 Connecting to Postgres with Go 279
10.2 Imports with Side Effects 284
10.3 Postgres Config Type 287
10.4 Executing SQL with Go 289
10.5 Inserting Records with Go 294
10.6 SQL Injection 298
10.7 Acquire a new Record's ID 302
10.8 Querying a Single Record 306
10.9 Creating Sample Orders 310
10.10 Querying Multiple Records 312
10.11 ORMs vs SQL 319
10.12 Exercises 324
10.12.1 Ex1 - Create new SQL tables and Go code to work with them 324
10.13 Syncing the Book and Screencasts Source Code 325

11 Securing Passwords **327**

11.1 Steps for Securing Passwords 327

11.2 Third Party Authentication Options 329

11.3 What is a Hash Function? 330

11.4 Store Password Hashes, Not Encrypted or Plaintext Values . . 333

11.5 Salt Passwords 336

12 Adding Users to our App **347**

12.1 Defining the User Model 347

12.2 Creating the UserService 351

12.3 Create User Method 354

12.4 Postgres Config for the Models Package 362

12.5 UserService in the Users Controller 366

12.6 Create Users on Signup 368

12.7 Sign In View 371

12.8 Authenticate Users 376

12.9 Process Sign In Attempts 378

13 Remembering Users with Cookies **381**

13.1 Stateless Servers 381

13.2 Creating Cookies 385

13.3 Viewing Cookies with Chrome 389

13.4 Viewing Cookies with Go 396
13.5 Securing Cookies from XSS 398
13.6 Cookie Theft 400
13.7 CSRF Attacks 402
13.8 CSRF Middleware 406
13.8.1 What is Middleware? 409
13.8.2 Coding the CSRF Middleware 413
13.8.3 Seeing the Changes 415
13.9 Providing CSRF to Templates via Data 416
13.10 Custom Template Functions 418
13.11 Adding the HTTP Request to Execute 423
13.12 Request Specific CSRF Template Function 428
13.13 Template Function Errors 433
13.14 Securing Cookies from Tampering 437
13.14.1 Digitally Signing Data 437
13.14.2 Obfuscation 441
13.15 Cookie Exercises 442
13.15.1 Ex1 - Set and View Cookies in your Browser 442
13.15.2 Ex2 - Explore the Cookie Docs 442
13.15.3 Ex3 - Redirect When a Cookie is Not Found 443
13.15.4 Ex4 - Attempt to Create Middleware 443

14 Sessions **445**

14.1 Random Strings with crypto/rand 445

14.2 Exploring math/rand 451

14.3 Wrapping the crypto/rand Package 454

14.4 Why Do We Use 32 Bytes for Session Tokens? 456

14.5 Defining the Sessions Table 459

14.6 Defining the Sessions Table 459

14.7 Stubbing the SessionService 463

14.8 Sessions in the Users Controller 468

14.9 Cookie Helper Functions 474

14.10 Create Session Tokens 478

14.11 Refactor the rand Package 481

14.12 Hash Session Tokens 486

14.13 Insert Sessions into the Database 491

14.14 Updating Existing Sessions 494

14.15 Querying Users via Session Token 497

14.16 Deleting Sessions 500

14.17 Sign Out Handler 501

14.18 Sign Out Link 504

14.19 Session Exercises 507

14.19.1 Ex1 - Move the token management code to its own type 507

14.19.2 Ex2 - Add a template for the current user page 507
14.19.3 Ex3 - Try using a JOIN in SQL 507

15 Improved SQL **509**

15.1 SQL Relationships 509
15.2 Foreign Keys 513
15.2.1 Existing table 514
15.2.2 Updating our code 514
15.3 On Delete Cascade 515
15.4 Inner Join 516
15.4.1 Prep the DB 516
15.4.2 Create records to query 517
15.4.3 Using JOIN 517
15.5 Left, Right, and Full Outer Join 519
15.6 Using Join in the SessionService 520
15.7 SQL Indexes 523
15.8 Creating PostgreSQL Indexes 526
15.9 On Conflict 527
15.10 Improved SQL Exercises 529
15.10.1 Ex1 - Design SQL database tables and relationships for an app 529
15.10.2 Ex2 - What are use cases for each type of JOIN? . . . 529

16 Schema Migrations **531**

16.1 What Are Schema Migrations? 531

16.2 How Schema Migration Tools Work 533

16.2.1 Tracking versions 533

16.2.2 Rolling back 534

16.3 Installing pressly/goose 535

16.4 Converting to Schema Migrations 538

16.5 Prepwork 538

16.6 Convert 538

16.7 Schema Versioning Problem 540

16.8 Running Goose with Go 545

16.9 Embedding Migrations 548

16.10 Go Migration Files 551

16.11 Removing Old SQL Files 553

17 Current User via Context **555**

17.1 Using Context to Store Values 555

17.2 Improved Context Keys 558

17.3 Context Values with Types 561

17.4 Storing Users as Context Values 562

17.5 Reading Request Context Values 564

17.6 Set the User via Middleware 566
17.7 Requiring a User via Middleware 572
17.8 Accessing the Current User in Templates 575
17.9 Request-Scoped Values 579

Course Resources

0.1 Ebooks, Slack, Source Code, and more

If you haven't already, **I highly recommend joining the course slack. You can request access to Slack here.**

The source code is currently stored as a private repository on GitHub. **You can request access to GitHub here.** After requesting access you will typically receive an email, but if you do not, log into your GitHub account and check for an invite there. In some cases users have disable GitHub notifications, which will cause the invite email to not get sent. I cannot do anything about this, and you need to log into your GitHub account and look for the invite.

As you progress through the course, you will typically find links the final source code for each lesson as well as a link showing what code changed in the lesson. This is often called a diff. If a lesson is missing these links (like this lesson), it is likely because we didn't make any changes to our code. This can happen when installing a developer tool like `git`, or when we are learning about something prior to making any changes in our code.

If you do see a source code link and it results in a 404 error, this is almost always due to either not being logged into your GitHub account, or not requesting access to the repo. Without these two things it shows up as a broken link.

The ebooks are listed below (), but they are lagging behind slightly, as I only

update them every week or so, while the lessons on the website get updated as soon as I release new changed. For the most up-to-date written version of each lesson, view the course from the courses website.

Finally, **if you run into an error while following a video** consider looking at the written version for a fix. In many cases if a tool is updated, or if a user runs into a unique situation that causes them an issue, the written lessons will get updated to provide advice, while the advice doesn't always make sense to add to the videos.

An example of this is when installing Postgres via Docker. In some cases users who already have Postgres installed may run into issues and the written lessons have advice on how to proceed, but most people won't need this advice, so it isn't included in the videos at this time.

Chapter 1

Getting Started

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 to make sure our 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. For now, follow along and make sure things are working.

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 be discouraging. It is very normal to write code, then later realize

how it could be improved. Very few people write perfect code on 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. Below is an example of doing this using the terminal, and the directory created is named **lenslocked**.

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

Aside 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. I also use a separate terminal program called **iTerm2** on Mac, but you can use the terminal built into VS Code or whatever else you are 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. If unsure of what editor to use, VS Code is a nice free option to start out with.

Next we 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 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.

```
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)
  fmt.Println("Starting the server on :3000...")
  http.ListenAndServe(":3000", nil)
}
```

Aside 1.2. Help, My Imports Keep Disappearing!

If you have your code editor setup with a Go extension, unused imports will be removed when you save the go file. If you continue with the rest of the code it will stop removing the imports (and will likely add them back in for you) once the imported packages are being used. More on this later.

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

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

Source Code

- Source Code - see the final source code for this lesson.

1.2 Troubleshooting and Slack

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.

Aside 1.3. 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.

Aside 1.4. 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.

```
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. A good editor will provide 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 course 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.

These are not the only editors that work well with Go, but they are two that work well with minimal setup and are easy to setup. Both have extensions for other languages (like Markdown), both provide all sorts of functionality for navigating through Go code, and both can provide feedback showing where

our code has compilation errors. The primary difference between the two is that VS Code feels a bit more like a text editor, while GoLand feels more like an entire development environment (aka an IDE).

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 familiar with emacs, vim, or an IDE, you should go with the option that works best for you. There is no need to attempt to mimic my development environment when it might not be a good fit for yourself.

Regardless of which editor is being used, it is strongly recommended that you focus on one thing at a time. For example, I do not recommend trying to learn both vim and Go at the same time. Instead, it is better to install a text editor you can use quickly and easily, then learn vim at a later date. 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 and Goland are incredibly quick to pick up and get proficient with, which is why I recommend them both.

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.

```
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 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 only 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.

Aside 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:

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

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

```
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
```

Aside 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

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.

Aside 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 the URL below.

```
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
```

`lenslocked` is the name I'm giving to our web application, and **`joncalhoun`** is my GitHub username. If you want, you can change these two but it might be easier if you don't your first pass through the course.

After running `go mod init`, you should find that it created a `go.mod` file that is 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.

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.

Aside 1.8. Binaries May Differ in Windows

If you are using Windows it is possible your binaries will be `.exe` files. This is unique to Windows, so you will need to keep that in mind as you proceed with the course.

There might come a time when you want to clean up your modules. For in-

stance, 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.

Source Code

- Source Code - see the final source code for this lesson.
- Diff - see the changes made in this lesson.

Chapter 2

Adding New Pages

2.1 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. Below are a couple:

- modd
- air

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

Aside 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. As of this writing, I believe the easiest way to install is using Go.

```
go install github.com/cortesi/modd/cmd/modd@latest
```

If that does not work, I suggest heading over to the website to review the most up-to-date installation instructions. 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.

```
**/*.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
}
```

Aside 2.2. Windows Users May Need to Tweak Things

I believe Windows users need to add `.exe` to the "daemon +sigterm" line to make it work, but I don't personally use Windows for development so I can't say for sure at this time.

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.

Source Code

- Source Code - see the final source code for this lesson.
- Diff - see the changes made in this lesson.

2.2 Setting Header Values

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.

```
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

Change your code to set the content type to **`text/html`** so that our page works again. Let's also add a charset to the header so it is set explicitly. This will tell the browser how characters are encoded, which is UTF-8 in our case.

```
w.Header().Set("Content-Type", "text/html; charset=utf-8")
```

Headers and their values are not unique to Go. If you wanted to read more about the Content-Type header, you could do so on the MDN Web Docs and everything there would apply regardless of the programming language you are using. 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.

Source Code

- Source Code - see the final source code for this lesson.
- Diff - see the changes made in this lesson.

2.3 Creating a Contact Page

We are going to add a new page to our application - the contact page. This is a page that will give users information for reaching out if they have questions or run into issues.

It will be a static HTML page and the code will be minimal, but the purpose of this new page isn't the content. It is meant to help us learn how to create a new handler function and set our application up to render different pages depending on the URL path. The new page will also be useful in the future when we learn more 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; charset=utf-8")
  fmt.Fprint(w, "<h1>Contact Page</h1><p>To get in touch, email me at <a href=\"mailto:jon@calh
}
```

Aside 2.3. 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:

```
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 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.

```
// Rename the function here
func homeHandler(w http.ResponseWriter, r *http.Request) {
  // unchanged
}

// 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)
}
```

Source Code

- Source Code - see the final source code for this lesson.
- Diff - see the changes made in this lesson.

2.4 Examining the http.Request Type

When a user visits the home page of our application, our code calls the `homeHandler` 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 commonly called routing.

While the standard library provides some basic 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. We will even create a very basic router as a learning exercise.

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 former implies a user is looking to load a signup form, while the latter suggests they are submitting their information to sign up, and these two actions need to be handled differently.

The URL should help us take care of the second part of routing - the path. If a user navigates to **`/contact`** or **`/faq`** we need to render different pages.

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 we will print 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/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.

Source Code

- Source Code - see the final source code for this lesson.
- Diff - see the changes made in this lesson.

2.5 Custom Routing

Now that we know how to programmatically determine the path of any incoming web request, we are going to use that information to build a basic router. We will start out using an if/else statement. Open **main.go** and update the **pathHandler** function to reflect the code below.

```
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 different paths. Visit

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 exit the if/else statement and since there isn't anymore code in the **pathHandler**, nothing is written to the response body and we see an empty page in the browser.

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.

```
switch r.URL.Path {
case "/":
  homeHandler(w, r)
case "/contact":
  contactHandler(w, r)
default:
  // TODO: handle the page not found error
}
```

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.

Source Code

- Source Code - see the final source code for this lesson.
- Diff - see the changes made in this lesson.

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 detects 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.

Aside 2.4. 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)
```

Source Code

- Source Code - see the final source code for this lesson.
- Diff - see the changes made in this lesson.

2.8 The http.Handler Type

When we call `http.HandleFunc` to register a handler, the `DefaultServeMux` is used behind the scenes. The code below is from the `net/http` package and shows this happening.

```
// 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`.

```
// 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 DefaultServeMux is used, otherwise the handler we pass in will be used. That means if we could somehow pass our **pathHandler** function, or something like it, into **ListenAndServe**, it would be used directly.

Unfortunately, our **pathHandler** function doesn't implement the **http.Handler** interface, so if we try to pass it directly into ListenAndServe we will get compilation error.

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

Aside 2.5. Warning: Tons of Information Incoming

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.

Source Code

- Source Code - see the final source code for this lesson.
- Diff - see the changes made in this lesson.

2.9 The 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 function like our `pathHandler` function.

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)
}
```

Aside 2.6. First Class Functions

First class functions refers to the ability to use functions in the same way as you would any other data type. For instance, a programming language with first class

functions will allow developers to assign functions to variables and pass them around as arguments to other function calls.

Go allows developers to do all of this and even allows developers to create new types using functions. There are languages where this type of behavior isn't allowed, and if you are coming from one of those this lesson may seem a bit odd at first.

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.

```
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.

```
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 and is perfectly valid, we can reduce this to a single line of code if we want. This is shown below.

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

http.HandlerFunc looks like a function call at first glance, but it isn't; it is a type conversion similar to converting a float into an integer:

```
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)
}
```

When we write **int(r)** or **float64(a)** we aren't making function calls. 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.

```
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.

Source Code

- Source Code - see the final source code for this lesson.
- Diff - see the changes made in this lesson.

2.10 Exploring Handler Conversions

In the previous two lessons we saw how the **`http.Handler`** and **`http.HandleFunc`** types both differ, and are still very similar. In this lesson we are going to explore how the Go standard library's **`http.ServeMux`** type takes advantage of this and converts all handler functions into an **`http.Handler`** so that it can avoid duplicating similar code behind the scenes while still allowing end users to use either type.

Let's start by reverting our code back to what we were initially using to register our routes. Update your **`main()`** function to match the following code:

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

In this code we don't convert functions like **`homeHandler`** and **`contactHandler`** into an **`http.HandlerFunc`**, nor do 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.

```
mux.Handle(pattern, HandlerFunc(handler))
//                        ^
//                  The handler conversion
```

On this line of code, the function we passed in is being converted into the **`HandlerFunc`** type, which if you recall implements the **`http.Handler`** interface. That new variable is then passed into the **`ServeMux.Handle`** method as an **`http.Handler`**.

Aside 2.7. Missing http Prefix

You don't see the **`http`** prefix here because this code is inside the **`net/http`** package, but I will often include the prefix when writing about the types to make it clear that they are part of the **`net/http`** package.

If you recall the code we started with, we had the following in our **`main()`** function:

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

In this code we were doing the same thing. We are converting our **`pathHandler`** function into the **`http.HandlerFunc`** type, and then it is an acceptable argument for a function that accepts an **`http.Handler`** interface.

By writing the **ServeMux** type this way, the Go team was able to keep pretty much all of the 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 **mux.HandleFunc(...)** method 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 will likely use the **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)
```

Others like to pick one and stick to it in their code. For instance, the following is valid code and in which we only use **HandleFunc**, even when we have a type that implements the **http.Handler** interface.

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

This code works because **ServeHTTP** is a function that matches what **HandleFunc** expects as an argument and it will eventually be converted back into an **http.Handler** like we saw before.

We could also do the opposite - always use **http.Handle**:

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

Moving forward, the key takeaways:

- **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.

Finally, let's get our code back into a good state for the rest of the course. I am going to use the **Router** type and update **main()** with the following code:

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

Source Code

- Source Code - see the final source code for this lesson.
- Diff - see the changes made in this lesson.

2.11 FAQ Exercise

We will frequently be using exercises to reinforce what we are learning. For this section there will only be one exercise, but you should try to complete it in a variety of ways to reinforce everything from this section.

2.11.1 Ex1 - Add an FAQ page

Try to create an FAQ page to your application and set it up so that when we visit <localhost:3000/faq> in our app the page is rendered.

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

Below is one solution to the exercise.

```
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 repeat the `w.Header().Set(...)` bit, but we will handle that later in the course.

Next, add a case to our router's `ServeHTTP` method.

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

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

Source Code

- Source Code - see the final source code for this lesson.
- Diff - see the changes made in this lesson.

Chapter 3

Routers and 3rd Party Libraries

3.1 Router Requirements

Earlier in the course we explored routing using the `http.HandleFunc` function. This required us to pass in both a pattern and a handler function, similar to the code below.

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

We also looked at how to route with if/switch statements and a custom router type. An example of this is shown below.

```
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)
  }
}
```

Lastly, we looked at how the `http.ServeMux` type is used behind the scenes to do the actual routing when we use functions like `http.HandleFunc`.

We could proceed using the `http.ServeMux` type for our application, but instead we are going to use a third party library. The goal of this lesson is to discuss why that is the right decision for our web app.

In all of the routing code we have written up until this point, our needs have been pretty minimal. Our paths are simple, we don't have dynamic variables in the paths, and we haven't had a need to route based on HTTP methods.

All of that is going to change as our application becomes more complex. Eventually we will need support for:

1. Routing based on both the path and the HTTP method being used.
2. Support for dynamic parameters in the routing path.

For instance, a web request using the `GET` HTTP method and the `/galleries` path will be used to retrieve a list of galleries, while a web request with the `POST` HTTP method to the same `/galleries` path will be used to create a new gallery. The HTTP method will helps us distinguish what action is being taken, and each will need handled differently.

Aside 3.1. Do we need to use HTTP methods?

While it is possible to choose different paths for every action and eliminate the need to route based on the HTTP method, it would be a bad idea to do this. For starters, this could result in `GET` requests creating and deleting resources which could lead to a variety of bugs and issues, but even if that weren't the case you are likely going to eventually work on an application that routes using HTTP methods, so understanding how it is done is beneficial long term.

An example of dynamic parameters would be a path like `/galleries/:id` where a portion of the path is a variable. In this case the ID of the gallery we want to interact with is the variable. With a dynamic ID variable we can then write a single handler function to show a gallery, and then that function only needs to look up the ID of the gallery being requested from the URL, and it can handle requests regardless of which specific gallery was requested. In other words, we want web requests to `/galleries/123` and `/galleries/24` to both be handled by the same code on our server, each using the correct gallery ID.

Over time, the paths we need to support will continue to get more complex. For instance, the following routes will all eventually be added to our application:

- `GET /galleries/:id` - View a gallery
- `GET /galleries/:id/edit` - A form to edit a gallery
- `PUT /galleries/:id` - Update a gallery
- `DELETE /galleries/:id` - Delete a gallery
- `GET /galleries` - View a list of galleries
- `GET /galleries/new` - A form to create a gallery
- `POST /galleries` - Create a gallery

While it is possible to not use dynamic parameters in your path, it is a very common practice used across the web, so it is worth learning and understanding.

With all of that in mind, we will be using chi for our routing needs. Chi is a third party library that is well-tested, widely used, and makes it easy to do all of the things we need from a router.

Aside 3.2. gorilla/mux is a good alternative

Another popular routing option is gorilla/mux. I have used it in many projects in the past and it is an equally valid choice for any web app.

While it is possible to make the standard library's `http.ServeMux` work, I don't believe that is the right choice for most web apps, including the one we build in this course. The standard library is great when your needs are simple, but the extra code needed to support the more advanced routing we will be using adds more complexity and risk to our codebase than using a well-tested third party router.

3.2 Using Git

In the course videos I use `git` to check out a branch between each lesson. You don't need to know or use git to go through the course, but **you should install git** to avoid any tooling issues. You can find setup information on the git website.

Git is useful development tool that makes it possible to work on separate features, bug fixes, and more at the same time. This is achieved by creating a branch for each task, then eventually merging those changes into a main branch.

For instance, I could work on a new UI feature in one branch, submit it to a peer to review, and then go work on a complicated bug in another branch without any code from the UI feature being present while fixing the bug.

In this course I use git to create a branch for each lesson where there are code changes. This makes it possible to get the code for any individual lesson, as well as seeing how it differs from the lesson before it.

As you read the book you don't really need to think about git or branches. I provide links to the source code after every lesson where our app code changes, but git is a very common development tool worth learning long term.

3.3 Installing Chi

We are going to be using chi for routing our application. Let's start by installing it and making sure everything is working as intended. We can also take this opportunity to explore what Go modules are doing behind the scenes.

Aside 3.3. The course uses Chi v5

I am using Chi `v5` in this course. At the time of this writing, it is the newest version, but it is possible that at some point a newer version will release. Even if this happens, **I highly recommending sticking with the same version I am using as you progress through the course.** This will make it easier to follow along with the course and avoid any confusion or bugs.

If a newer version of Chi releases there will likely be breaking changes. As a result, using it while going through the course will force you to spend more time reading the docs and figuring out how the new version works. It will also be harder to compare your code to mine to find bugs since there will always be differences in our routing code.

Rather than using a newer version upfront, I suggest you use the same version as the course does and then upgrade the library once you have completed the course. Not only will this give you practice with the very real process of upgrading a dependency, it will also make the process easier as any breaking changes will likely be made obvious by the compiler.

As always, you are welcome to join the course Slack to discuss upgrading any dependencies or any other related questions.

Before installing a library we should first verify we are in the correct directory.

```
# Verify we are in the correct directory. pwd stands for
# print working directory and will show you what directory
# you are in.
pwd
# Output (yours will vary slightly):
# /Users/jon/Dev/courses-code/lenslocked

# We can also use ls to see what files are in the directory.
# Here I am looking for a file named go.mod to make sure I
# am in the correct directory.
ls
# Output:
# README.md go.mod    main.go   modd.conf
```

Now that we are sure we are in the correct directory, we can use `go get` to install the Chi library.

```
go get -u github.com/go-chi/chi/v5
```

When we run `go get` the Go tooling handles downloading any necessary files and installs the Chi library. It will also update our `go.mod` file with the exact version it installed.

```
require github.com/go-chi/chi/v5 v5.0.7 // indirect
```

Your minor version might be slightly different, but as long as your major version is the same you should be good to go.

Aside 3.4. Potential Errors

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. See the previous lesson for more information.

Moving forward, anytime someone builds or runs our code, go modules will handle downloading this library during the build process. That is a benefit of Go modules being built right into the Go tooling.

Down the road if we find ourselves no longer using some libraries we previously installed, we can use `go mod tidy` to clean up our `go.mod` file and remove unused libraries. If we run it now it will remove `chi` because we don't use the library in our code just yet. You can try that out if you want, but if you do be sure to re-run the `go get` command to install chi again!

Source Code

- Source Code - see the final source code for this lesson.
- Diff - see the changes made in this lesson.

3.4 Using Chi

Chi is going to:

1. Make using URL parameters much easier.
2. Enable HTTP method-based routing.
3. Make mounting subrouters a bit easier if we need to do that.
4. Provide some nice middleware, which we will learn about later.

In addition to making it easier to achieve some of our goals, Chi is also well tested and very popular, making the likelihood of undiscovered bugs pretty minimal.

As we progress through the course you will learn how Chi works in more detail. For now, our goal is to replicate the functionality our application currently needs which should help us become familiar with the basics.

Aside 3.5. Additonal examples

In addition to what we cover in the course, I also suggest reading the examples in the docs and experimenting a bit on your own. Long term this is going to be how you typically learn a new library, so it is good idea to get some practice.

Open up `main.go` and add the chi v5 import.

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

  "github.com/go-chi/chi/v5"
)
```

Next, navigate to the `main()` function and create a new router using the `chi.NewRouter()` function.

```
r := chi.NewRouter()
```

Aside 3.6. Auto-imports and Go Modules

Auto-imports are incredibly nice at times, but unfortunately there are times where it doesn't quite import what you want. This is one of those cases. If you don't explicitly verify you are importing Chi v5, there is a chance that v1 will be imported and your code won't compile as expected.

To check for this, head to your imports and look for `github.com/go-chi/chi/v5`. If you see an import without the `v5` part at the end, you will need to manually add it in.

We can now use our Chi router to set up some routes. Chi provides methods like `Get`, `Put`, and `Post` to declare routes that are specific to a single HTTP method. We will be using `Get` rather than `HandleFunc` since our routes are only meant to work for GET requests.

```
r.Get("/", homeHandler)
r.Get("/contact", contactHandler)
r.Get("/faq", faqHandler)
```

Now that we have our routes set up, we need to pass our router into the **ListenAndServe** function call so it knows to use our router.

```
// The last argument needs to be r, our chi router.
http.ListenAndServe(":3000", r)
```

Finally, we are going to add a handler for when invalid paths are used. We did this previously using the **default** clause in our custom router's switch statement. Chi has a default handler we could use, but just in case we wanted to do something custom Chi also provides us with a handy **NotFound** function. To use it, we pass a handler function in that will be used anytime an invalid path is routed.

```
r.NotFound(func(w http.ResponseWriter, r *http.Request) {
  http.Error(w, "Page not found", http.StatusNotFound)
})
```

At this point we can remove the **Router** type, any methods we created for it, as well as any usage of it inside of the **main()** function. We can also remove the **pathHandler** function. When finished, our main function should match the code below.

```
func main() {
  r := chi.NewRouter()
  r.Get("/", homeHandler)
  r.Get("/contact", contactHandler)
  r.Get("/faq", faqHandler)
  r.NotFound(func(w http.ResponseWriter, r *http.Request) {
    http.Error(w, "Page not found", http.StatusNotFound)
  })
  fmt.Println("Starting the server on :3000...")
  http.ListenAndServe(":3000", r)
}
```

Before proceeding, be sure to run your application and test all of your pages to make sure everything is working correctly. It is also worth following the link to final source code for this lesson to verify you deleted the correct code.

Source Code

- Source Code - see the final source code for this lesson.
- Diff - see the changes made in this lesson.

3.5 Chi Exercises

This lesson consists of two optional exercises intended to help you learn a bit more about Chi.

3.5.1 Ex1 - Add a URL Parameter

Read the docs and see if you can add a URL parameter to one of your routes, retrieve it in your handler, and output it to the resulting HTML.

For example, imagine that we wanted to add support for a path used to show individual galleries. We might use the path `/galleries/<id-here>`. See if you can create a handle that retrieves the id from the URL using Chi, and setup the Chi router to direct requests to the correct handler.

Aside 3.7. Hint

See these docs if you need some guidance. You shouldn't need to use context, just the `URLParam` method.

3.5.2 Ex2 - Experiment with Chi's builtin middleware

Chi provides quite a few builtin middleware. One is the Logger middleware, which will track how long each request is taking. Try to add it to your application, then to only a single route.

What other types of uses could you imagine middleware serving in an application?

Chapter 4

Templates

4.1 What are Templates?

At this point our application only returns static HTML. Even if we were able to look up a list of FAQ questions, we don't have a way of updating our HTML to include the new questions. What we need is a way to render HTML with dynamic data. In this section we are going to learn about the `html/template` package in the standard library that will allow us to create dynamic HTML, but first let's talk about templates in general.

Templates are a bit easier to understand if you think of them like Mad Libs. Mad Libs are silly games where you start with a story and some blanks spots for words, then you ask your friends to give you works to fill in the blanks. We don't want to end up with gibberish at the end, so each blank, spot often has some guidance on what type of work should be used. Below is an example form the original Mad Libs book:

```
_____________! he said ________
 exclamation            adverb

as he jumped into his convertible
```

```
______ and drove off with his
 noun

___________ wife.
 adjective
```

If we were playing, we might have the following exchange:

```
Me: Give me an exclamation
You: Aghast
Me: Give me an adverb
You: Angrily
Me: A noun
You: Shoe
Me: An adjective
You: Grumpy
```

Then I would fill in the template to create the following story:

```
_Aghast______! he said _angrily_
 exclamation            adverb

as he jumped into his convertible

_shoe_ and drove off with his
 noun

_grumpy____ wife.
 adjective
```

In this case the story is pretty silly, which was a big part of the game, but nonetheless we filled in a template and created a dynamic result.

Pretty much every programming language that has a library for building http servers will also have a templating library that works in a similar fashion. Rather than asking for nouns and adjectives, developers create a template that asks for specific variables, and sometimes even performs basic logic based on the value of a variable.

For instance, if we wanted to welcome new users, we might create the following HTML template.

```
<h1>Hello, {{.Name}}</h1>
```

By itself this template isn't very useful, but if we provide it with an object that has a `Name` field, we can now start to render dynamic HTML. For instance, if we used the greeting template above and provided it with the following data:

```
User{
  Name: "Susan",
}
```

The final result would be the HTML below.

```
<h1>Hello, Susan</h1>
```

Unlike the Mad Libs we saw, templates can be more complex. We can use conditional rendering based on the data provided, like showing a sign in or sign out button depending on whether the user is signed in or not. We are able to iterate over a list of items rendering each with a bit of HTML. With Go's `html/template` package we can even add custom functions, like one that converts a user's name into all capital letters before it is finally written to the HTML.

As we progress through the course we will gradually learn how handle each case we need as it comes up, but in this section we are going to work on getting a good overview of how Go's `html/template` package works and then we will end things by updating the FAQ page to render using a template and data we pass in from our Go code.

4.2 Why Do We Use Server Side Rendering?

When dealing with web apps, there are two common ways to render data into the final HTML that user's browser then renders:

1. The server generates the HTML and returns it. This is commonly referred to as server-side rendering.

2. The server returns data, and then lets something else render it. This is commonly referred to as an API.

Imagine that we wanted to render a welcome page for a new user and had the following data:

```
// Using this data to render...
user := User{
  Name: "Bob",
  Email: "bob@calhoun.io",
  ...
}
```

Using the first technique, our web server might use a template like this:

```
<body>
  <!-- A link to the users account details -->
  <a href="/account">{{.Email}}</a>
  <!-- ... -->
  <h1>Hello, {{.Name}}!</h1>
</body>
```

The final result would be the HTML below.

```
<body>
  <!-- A link to the users account details -->
  <a href="/account">bob@calhoun.io</a>
  <!-- ... -->
  <h1>Hello, Bob!</h1>
</body>
```

Using the second approach, our server wouldn't return HTML. Instead, it would just return data. Perhaps JSON like below.

```
{
  "name": "Bob",
  "email": "bob@calhoun.io"
}
```

With the second approach we would need something to render the data, so we might create a React application that uses code roughly like the code below to render the final HTML.

```
import React, { useState } from 'react';

function Example() {
  const { name, email } = // data from our Go server

  return (
    <body>
      <h1>Hello, {name}!</h1>
      ...
      <!-- A link to the users account details -->
      <a href="/account">{email}</a>
    </body>
  );
}
```

In reality there is a lot more going on than what we see here, but the key detail I want to point out is that in both cases we eventually end up using templates to render the final HTML. The two templates have a different syntax, but at the end of the day data is rendered using templates. Whether those are Go

templates, React, or something else, it is extremely likely that some template engine was involved.

APIs are popular these days for a many reasons, but one of the big ones is the ability to fetch data and render it however you see fit. That means we could use the same server to theoretically power an iOS app, and Android app, and a website using JavaScript to interact with the API. The downside to this approach is that it adds more complexity.

In this course I opted to teach web development using server side rendering. That means we won't be using a JavaScript frontend. I did this for a few reasons:

1. Using a JavaScript frontend adds complexity, and too much complexity is bad for learning.
2. Server-side rendering works well. It isn't bad or outdated
3. Our application doesn't benefit from a single page application (SPA).

4.2.1 JavaScript Frontends Add Complexity

It is worth noting that I have nothing against React or other JS frontends. They are great tools and are useful in many applications. Unfortunately, all technology comes with a trade-off.

Go is known for being easy to read and maintain, but this often comes at the cost of being more verbose.

Using a JavaScript frontend allows us to create far more dynamic pages that feel like a native application, but one of the costs is added complexity. If we reexamine the React sample code from earlier in this lesson it looks similar to the complexity of server-side rendering, but the React sample is ignoring several things:

- Making API calls over the network to fetch data
- Handling network errors
- Managing user authentication properly
- Building and deploying with both a Go and JavaScript app
- And more!

In addition to needing to handle extra details, we would also need to learn JavaScript and React, both of which are deserving of their own dedicated course.

In short, using a JavaScript frontend means we would need to learn nearly twice as much, and all the evidence that I have seen suggests that this is a bad way to learn. A better approach is to focus on the fundamentals, then expand upon them in the future. This gives us a solid foundation to build from without being overwhelmed.

I know that can be disappointing when you want to learn it all at once, but trust me, you will have much more success if you gradually build each skill then move on to the next one vs floundering and trying to master ten skills at once.

4.2.2 Server-side Rendering Works Well

Despite what you may have heard, server-side rendering isn't bad or outdated.

While it is true that you won't read about server-side rendering as often on tech websites, this isn't because it is a dead technology. This news phenomenon is a by-product of server-side rendering being an established and well understood technology while JavaScript frontends are still relatively new. As a result, people are often writing about lessons learned, new technologies in the space, and more. This doesn't happen as much with server-side rendering because it is a well understood space.

SQL, a database query language, is similar in this sense. While you will often read about new databases on tech websites, the reality is SQL is still the most popular database query language and is used in new applications every day. We read about it less often not because it is outdated, but because SQL has been around long enough that there isn't much new to write about.

Anecdotally, I use server side rendering in quite a few projects, and all of them work incredibly well. Those projects are often easier to maintain,faster to iterate on, and it makes it easier for a developer to own a feature completely. In the past I worked on a project with a Ruby backend and a React frontend, and one of the limiting factors to development speed was frontend development because most of the team knew Ruby, but very few knew React well enough to implement complex features.

When working with very small teams, simplicity is something I value. This course is designed to teach developers how to build an entire application on their own, so I opted to build the app the way I felt was best based on my experience. If I was building the Lenslocked app as my own project, it would use server-side rendering until there was a need to use a JS frontend.

4.2.3 Our App Doesn't Need a to be a SPA

Chances are you have heard the term, "single page application." It is commonly shortened to SPA. A single page application is an app where the browser only loads one page, and after that point a JavaScript application takes over. When users clicks a link or submits a form, the JavaScript decides what to do and updates what you see on the screen. The browser is still used to render HTML, but that HTML is being updated by the JavaScript code and the browser never has to load a new page.

The upside to this approach is that an application can feel like something installed natively on a user's computer. Gmail is a good example of this. Users don't see page loads when clicking into each email. When a user hits reply, the

input box where they type appears without a page load. It feels very similar to an email client that would be installed on their computer.

While much of this is possible with server-side rendering and a little bit of JavaScript, applications become harder to manage as they become more complex. In cases like Gmail, a JavaScript frontend is a great option to explore.

We are building a photo gallery and our needs are fairly simple. I am a big believer in choosing the right technology for the job and React isn't right for this project.

It is very possible that the application might eventually evolve into something that is deserving of a more complex frontend, but the version we are building doesn't justify the use of React or any other JavaScript frontend. If that ever changes, a project can alway be refactored to support an API, and we will be structuring our Go code in a way that makes this fairly easy to do.

4.3 Creating Our First Template

Go's standard library has two template libraries:

- `text/template`
- `html/template`

Both libraries are used by developers in roughly the same way, but behind the scenes they each work differently.

The `html/template` package is designed for creating HTML, so it provides some additional functionality to assist in that. For instance, when rendering variables in HTML it helps us avoid cross site scripting (XSS). We will explore this in a future lesson.

The `text/template` package doesn't add HTML specific logic behind the scenes. Instead, it assumes the user wants templates to render variables exactly as they are provided.

As a result, it is possible that auto-imports in your editor might import the incorrect template package, and **it is worth ensuring you have the correct template package imported to avoid any security issues**.

Aside 4.1. Additional Reading

In this course we will focus on what we need to know about templates to proceed, but if you want to read more about them you can here at calhoun.io/intro-to-templates.

Throughout the rest of this lesson we are going to create a template and write some Go code to execute the template. The code won't be used in our final web application and is only used for learning purposes, so we will create a new directory to keep this code.

```
# Create the cmd directory
mkdir cmd
# Change into the cmd directory
cd cmd
# Create the exp directory inside the cmd directory
mkdir exp
# Change into the exp directory
cd exp
```

Alternatively, if you are NOT running on Windows, you can shorten this to the following:

```
mkdir -p cmd/exp
cd cmd/exp
```

We now have a **cmd** direcory inside of our codebase, and inside of it we have an **exp** directory. We will be working directly inside of this directory for the remainder of this lesson.

Aside 4.2. The cmd directory

A common pattern used in the Go community is to create a **cmd** directory that is used to store the various programs a project can build. For instance, Kubernetes has several binaries that can be built using the same project source code.

This pattern allows developers to create multiple **main** packages and to pick the correct one depending on what they want to build. For instance, we could have a **main** package for building a command line interface (CLI) at the **cmd/cli** directory, and another **main** package used to build our web server in the **cmd/server** directory.

We are going to follow a similar pattern with our experimental code, storing it in the **cmd/exp** directory. We will eventually move our web server to a directory inside the **cmd** folder.

Next we need to create a template. We will use the file extension **.gohtml** as most editors with a Go plugin will recognize this as a Go HTML template.

```
code hello.gohtml
```

Inside of **hello.gohtml** add the following.

```
<h1>Hello, {{.Name}}</h1>
```

With our first template we are essentially saying that we want to create an HTML file that outputs a heading (**<h1>**), and inside of that heading we want to insert the string **Hello, {{.Name}}** where the name changes based on the data passed into the template. This will become clearer with a complete example, so let's create a Go file to execute the template.

Create an new Go file in the **cmd/exp** directory.

```
code exp.go
```

Add the following code to **exp.go**. We will walk through what is happening in a moment.

```
package main

import (
        "html/template"
        "os"
)

type User struct {
        Name string
}

func main() {
        t, err := template.ParseFiles("hello.gohtml")
        if err != nil {
                panic(err)
        }

        user := User{
                Name: "John Smith",
        }

        err = t.Execute(os.Stdout, user)
        if err != nil {
                panic(err)
        }
}
```

Verify your code works by running it.

```
# You must be inside the cmd/exp directory when running the code
go run exp.go
```

Hopefully your output looked something like below.

```
<h1>Hello, John Smith</h1>
```

Now let's talk about what is happening in the code. The first thing we declare is the **`User`** type.

```
type User struct {
        Name string
}
```

This type has a **`Name`** field which is exactly the same as the field we reference in **`hello.gohtml`** via **`{{.Name}}`**. When passing data into a template, we need to make sure the field names of our structs are exported and match the names we use in our template.

Next up we have the **`main()`** function, and this starts off by calling template.ParseFiles.

```
t, err := template.ParseFiles("hello.gohtml")
if err != nil {
  panic(err)
}
```

`template.ParseFiles` is used to parse one or more template files. If everything goes well, you will receive a template.Template back and the error will be nil, otherwise the error will not be nil.

Given that this is experimental code, I am opting to panic if we get an error, but as we progress through the course we will start to see how to handle errors more gracefully.

If you are coding this up by hand and using auto-imports, it is a good idea to make sure the **`html/template`** *was imported and not* **`text/template`** *at this point.*

Aside 4.3. Paths are relative to where we run our code from

One important detail to note here is that the path to the template file is reletive to where we run our code from. **Filepaths are not relative to where the code is located.**

We are currently running our code from inside the **`cmd/exp`** directory, so we only need to provide **`hello.gohtml`** as the file path. On the other hand, if we were to navigate to our top level **`lenslocked`** directory and run our code via **`go run cmd/exp/exp.go`** we would see an error because there isn't a file named **`hello.gohtml`** in our **`lenslocked`** directory. If we wanted to run our code from here, we would need the filepath to be **`cmd/exp/hello.gohtml`**.

Later in the course we will look at how we can use a new feature in Go to embed these templates into our final binary, but for now we just need to keep this in mind.

At this point if our code hasn't panicked we know that **`t`** should have a valid template assigned to it, so we can proceed with creating data an executing the template.

```
user := User{
  Name: "John Smith",
}

err = t.Execute(os.Stdout, user)
```

```
if err != nil {
  panic(err)
}
```

When we call `t.Execute` we provide it with two arguments:

1. An `io.Writer` to write the output to.
2. An `interface{}` which is the data needed by the template.

If we were creating an http handler function, we might pass the `http.ResponseWriter` in as the `io.Writer`, but since this is a command line program, we are instead passing in `os.Stdout`. `os.Stdout` stands for standard output, and it essentially just means out to our terminal. When we run `fmt.Print` it is using `os.Stdout` behind the scenes to write to our terminal.

`t.Execute` accepts an `interface{}` as the second argument, which means we can pass anything in here including `nil`. If you are unfamiliar with the empty interface, I recommend reading How do interfaces work in Go.

Our template needs a data type with a `Name` field, so we pass the `user` variable in as the second argument.

While Go is a type language, data used inside of a template doesn't benefit from the same type checking. That is, if we updated our template with the following code and then ran our program, we would see an error at runtime, not compile time.

```
<h1>Hello, {{.Name}}</h1>
<p>You are {{.Age}} years old.</p>
```

Run the code...

```
go run exp.go
```

Give us the following error:

```
panic: template: hello.gohtml:2:13: executing "hello.gohtml" at <.Age>: can't evaluate field Ag
```

If we wanted this code to work, we would need to update our `User` type by adding the `Age` field.

```
type User struct {
        Name string
  Age int
}
```

We would also want to set an age in our code.

```
user := User{
  Name: "John Smith",
  Age: 123,
}
```

Running the updated code would then give us:

```
<h1>Hello, John Smith</h1>
<p>You are 123 years old.</p>
```

Source Code

- Source Code - see the final source code for this lesson.
- Diff - see the changes made in this lesson.

4.4 Cross Site Scripting (XSS)

Code injection is an exploit where attackers try to get their own malicious code to run somewhere it wasn't intended to run. One common way this can occur is when a website has an input field where users can provide data. If the input isn't properly sanitized, it might lead to code injection.

The popular example of this is creating a string that terminates an existing SQL query and starts a new one. This example even has an XKCD commic about it: https://xkcd.com/327/

The joke of the comic is that the child is named `Robert'); DROP TABLE students; --`, which happens to be a string that if not handled properly could terminate an existing SQL query and then run the query `DROP TABLE students;` which would delete a bunch of data in the database. We will explore SQL injection later in the course.

Another type of code injection is cross site scripting (XSS). Rather than having a server execute the code, XSS exploits result in code running in the browser of a website's user. Let's look at an example to help clarify.

Imagine our website let users enter information into a `bio` field. Our goal might be to allow them to provide a bit of information about themselves that can then be shared with other users. Rather than building all of this, we will fake it inside of our `homeHandler` by assigning a value to a variable.

```
func homeHandler(w http.ResponseWriter, r *http.Request) {
  bio := `I have been a developer for 256 years!`
        w.Header().Set("Content-Type", "text/html; charset=utf-8")
        fmt.Fprint(w, "<h1>Welcome to my awesome site!</h1><p>User's bio: " + bio + "</p>")
}
```

If we were to start up our web server and visit the home page, we would see roughly the following HTML being rendered on our screen:

```
<h1>Welcome to my awesome site!</h1>
<p>User's bio: I have been a developer for 256 years!</p>
```

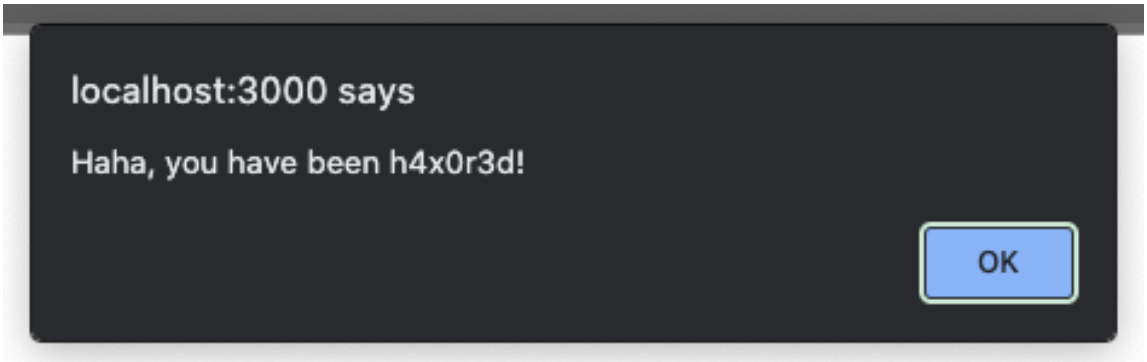
We are currently rendering the user's **bio** exactly as they provided it, which seems fine for now, but it could actually lead to XSS. Imagine if a user input the following as their bio information:

```
<script>
  alert("Haha, you have been h4x0r3d!");
</script>
```

We can easily test what happens by updating our **bio** variable with the new string. The code above is broken into several lines to make it easier to read, but an attacker could easily provide it a single line of text giving us the following:

```
func homeHandler(w http.ResponseWriter, r *http.Request) {
  bio := `<script>alert("Haha, you have been h4x0r3d!");</script>`
        w.Header().Set("Content-Type", "text/html; charset=utf-8")
        fmt.Fprint(w, "<h1>Welcome to my awesome site!</h1><p>User's bio: " + bio + "</p>")
}
```

Restart the server and visit the home page. We are now being greeted with a JavaScript alert!



When we use plain old strings to create our HTML output, we aren't taking into account the fact that variables might have script tags and other HTML inside of them As a reuslt, the final output returns the `<script>...</script>` as if we wrote it ourselves and the browser will process it like the rest of the HTML.

This is called cross-site scripting (XSS), and it is a form of code injection. While our example is only a silly alert message, real attacks might be more nefarious in nature and could lead to user information being leaked to the attacker, among other things.

Luckily, pretty much every modern HTML template library handles XSS prevention for us. This is done by converting characters with a special meaning in HTML into what are known as HTML character entities.

Going back to our script example, the encoded version would look like the following:

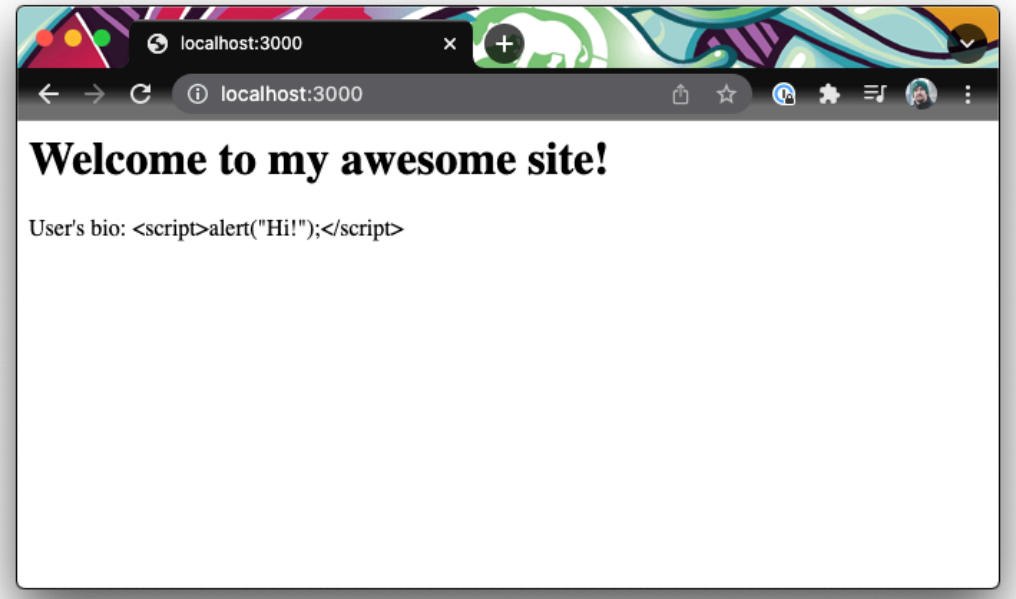
```
&lt;script&gt;alert(&quot;Hi!&quot;);&lt;/script&gt;
```

When a browser sees `<` it knows that it should render the `<` character on the screen. When it see `>` it knows to render the `>` character.

If we take this string and insert it into our `bio` variable, we will see that the browser renders the `<script>...</script>` string rather than trying to process it like normal HTML.

```
func homeHandler(w http.ResponseWriter, r *http.Request) {
        bio := `&lt;script&gt;alert(&quot;Hi!&quot;);&lt;/script&gt;`
        w.Header().Set("Content-Type", "text/html; charset=utf-8")
        fmt.Fprint(w, "<h1>Welcome to my awesome site!</h1><p>User's bio: "+bio+"</p>")
}
```

Restart your server and navigate to the home page to see what is rendered now.



Let's take a look at the `html/template` package doing this for us. Open up your `cmd/exp/hello.gohtml` file and update it with the following code:

```
Bio: {{.Bio}}
```

Now open `cmd/exp/exp.go` and update the `User` type to have a `Bio` field. We will also need to update the `user` variable we declare to set the field.

```
type User struct {
  Bio string
}

// in main() update the user we create
user := User{
  Bio: `<script>alert("Haha, you have been h4x0r3d!");</script>`,
}
```

Now we are going to run this code and see what output we get. Be sure to **cd** into the **cmd/exp** directory so our relative paths work correctly when we run the code.

```
cd cmd/exp
go run exp.go
```

This will produce the following output:

```
Bio: &lt;script&gt;alert(&#34;Haha, you have been h4x0r3d!&#34;);&lt;/script&gt;
```

I mentioned earlier that it is important to verify that we have imported the correct template package. XSS prevention is one of the reasons why this is important, and we can actually demonstrate that now.

In **exp.go**, change the import to instead be **text/template**.

```
import (
  "os"
        "text/template"
)
```

When we run **exp.go** now we will see something very different in the output:

```
Bio: <script>alert("Haha, you have been h4x0r3d!");</script>
```

Change the import back to **html/template** before we proceed.

```
import (
        "html/template"
        "os"
)
```

The last thing we will discuss is how to provide data to the template that we do NOT want to be escaped. For instance, if our server processed Markdown and converted it into HTML that was then passed into a template, we would want this to be rendered as HTML and not converted into character entities.

The `html/template` package provides us with the `template.HTML` type, which is basically a string, but when the `html/template` is executing a template it knows to not encode any data with that type. We can test this by updating our `Bio` field to be of the type `template.HTML`.

```
type User struct {
        Bio template.HTML
}
```

Running our code now, even with the `html/template` package, results in the following output:

```
Bio: <script>alert("Haha, you have been h4x0r3d!");</script>
```

Source Code

- Source Code - see the final source code for this lesson.
- Diff - see the changes made in this lesson.

4.5 Alternative Template Libraries

Thus far in the course we have mostly created HTML by hand. An example of this is shown below.

```
func homeHandler(w http.ResponseWriter, r *http.Request) {
        bio := `&lt;script&gt;alert(&quot;Hi!&quot;);&lt;/script&gt;`
        w.Header().Set("Content-Type", "text/html; charset=utf-8")
        fmt.Fprint(w, "<h1>Welcome to my awesome site!</h1><p>Bio:"+bio+"</p>")
}
```

As we proceed through the course we are going to begin using the **html/template** package from the standard library for our templating needs. With that said, some people aren't big fans of this particular library. I personally don't think it is a bad library, but it takes some getting used to.

If you end up disliking the **html/template** package that we use, I suggest checking out plush. Passing data into plush templates is a little different than what we do in the course, but the HTML template files are similar to those used in Ruby on Rails and other some other languages.

While it is possible to create your own template library, I would not recommend it, as it is fairly easy to introduce a security issue without realizing it.

Regardless of what route you opt to go, I recommend following the course as-is the first time you go through the material. We will end up using a few advanced techniques - like adding custom functions to our templates - and you will have more success if you focus on learning the course material first, and then convert the templates to **plush** afterwards.

4.6 Contextual Encoding

In addition to encoding data to avoid XSS attacks, Go's **html/template** package will also encodes data based on the context, often making it easier to use Go data as JavaScript data when working inside of a script tag.

While we won't be taking advantage of this anytime soon, I want to demonstrate it so that you understand another reason why we are using the **html/template**

package. Open the `cmd/exp/hello.gohtml` template and update it with the following code.

```
<script>
  const user = {
    "name": {{.Name}},
    "bio": {{.Bio}}
  };
  console.log(user);
</script>
```

Next, update the `User` type inside `cmd/exp/exp.go` to have both a Name and Bio field, both of the type `string`. We also need to assign a value to the Name field.

```
type User struct {
        Name string
        Bio  string
}

// inside main()
user := User{
  Name: "Jon Calhoun",
  Bio:  `<script>alert("Haha, you have been h4x0r3d!");</script>`,
}
```

Now if we run our code we will see that the data is encoded differently.

```
<script>
  const user = {
    "name": "Jon Calhoun",
    "bio": "\u003cscript\u003ealert(\"Haha, you have been h4x0r3d!\");\u003c/script\u003e"
  };
  console.log(user);
</script>
```

Not only does our `Bio` field get encoded very differently, we also have quotation marks around both the `{{.Name}}` and `{{.Bio}}` fields even though we never added those to our template.

When executing a template, the **html/template** library uses both the data type of the original data, and the context of where it is being output in the template to determine the best format. If we were toa dd an **Age** field to our code, we would see this does not have quotation marks added.

Update **exp.go** with the following code.

```
type User struct {
        Name string
        Bio  string
  Age int
}

// in main()
user := User{
  Name: "Jon Calhoun",
  Bio:  `<script>alert("Haha, you have been h4x0r3d!");</script>`,
  Age:  123,
}
```

Next update the **hello.gohtml** template with the following:

```
<script>
  const user = {
    "name": {{.Name}},
    "bio": {{.Bio}},
    "age": {{.Age}}
  };
  console.log(user);
</script>
```

Running our program now gives us the following output:

```
<script>
  const user = {
    "name": "Jon Calhoun",
    "bio": "\u003cscript\u003ealert(\"Haha, you have been h4x0r3d!\");\u003c/script\u003e",
    "age":  123
  };
  console.log(user);
</script>
```

While the `<` encoding looks weird at first, if we were to open up our browser console and pasted the JavaScript we generated into it we would quickly see that the encoding used is appropriate for JavaScript. For instance, running the following:

```
const user = {
  "name": "Jon Calhoun",
  "bio": "\u003cscript\u003ealert(\"Haha, you have been h4x0r3d!\");\u003c/script\u003e",
  "age":  123
};
console.log(user.bio);
```

Will result in the following output in a JavaScript console.

```
<script>alert("Haha, you have been h4x0r3d!");</script>
```

Depending on where you insert the data from your template, the `html/template` package will try to encode it according to the current context.

Source Code

- Source Code - see the final source code for this lesson.
- Diff - see the changes made in this lesson.

4.7 Home Page via Template

Over the next few lessons we are going to replace all of the HTML our website creates with templates. We will do this by creating a template file for each page.

Create a directory for our HTML template files and create the a file for our home page template.

```
mkdir templates
code templates/home.gohtml
```

If you are on Windows, you may need to use a backslash (`\`) instead of the forward slash (`/`) in file paths. We will discuss handling this in our code later in the lesson.

Add some HTML to the home template.

```
<h1>Welcome to my awesome site!</h1>
```

We don't have any dynamic data in this template, but we want to get in the habit of creating our various pages this way so we can easily add in dynamic data as needed. We will also start using layouts and other cool features later and will benefit from all our pages already being templates.

Next we are going to update our home page handler to parse the template and execute it. Open **`main.go`** and update the **`homeHandler`** function.

```
func homeHandler(w http.ResponseWriter, r *http.Request) {
        w.Header().Set("Content-Type", "text/html; charset=utf-8")
  // NOTE: your path may be different in windows. More on this shortly.
        tpl, err := template.ParseFiles("templates/home.gohtml")
        if err != nil {
                panic(err) // TODO: Remove the panic
        }
        err = tpl.Execute(w, nil)
        if err != nil {
                panic(err) // TODO: Remove the panic
        }
}
```

*Double check you are importing the **`html/template`** package and not the **`text/template`** package.*

In most environments, paths are separated with a forward slash (`/`). This includes paths in URLs, Linux, and Mac OS. We will be deploying our code to

a Linux-based server, as this is pretty much what all web severs run. I also do my local development in Mac OS, so I will frequently hard-code paths using the forward slash.

If you happen to be using Windows, you will almost certainly run into issues if you use the forward slash when referencing a file path. Windows is a bit of an outlier and uses backslashes (`\`) for file paths. If you need your code to be operating system agnostic, you could instead use the following code to create your file paths:

```
tplPath := filepath.Join("templates", "home.gohtml")
```

This code will create a path like `templates/home.gohtml` or `templates.gohtml` depending on the operating system the Go code was built for.

If we update our `homeHandler` to use the `filepath` package we get the following code:

```
func homeHandler(w http.ResponseWriter, r *http.Request) {
        w.Header().Set("Content-Type", "text/html; charset=utf-8")
        tplPath := filepath.Join("templates", "home.gohtml")
        tpl, err := template.ParseFiles(tplPath)
        if err != nil {
                panic(err) // TODO: Remove the panic
        }
        err = tpl.Execute(w, nil)
        if err != nil {
                panic(err) // TODO: Remove the panic
        }
}
```

With our file path issue fixed, let's look at our error handling.

The first error occurs when our template fails to parse. This can happen for a variety of reasons, but they all relate to a template that isn't valid no matter what data we provide to it. For instance, if our template tried to use a nested

tempalte that doesn't exist, or if we tried to use an invalid function, we would see a error after calling `ParseFiles`.

We can test this in our code. Open `home.gohtml` change it to read:

```
<h1>Welcome to my awesome site!</h1>
{{ invalidFunction }}
```

Now rebuild and restart your web server and try to navigate to the home page at http://localhost:3000. There will be an error like `ERR_EMPTY_RESPONSE` in the browser, and if we look at the terminal we will see a panic message.

```
http: panic serving [::1]:59876: template: home.gohtml:2: function "invalidFunction" not define
... + more
```

Long term we will make sure all of our templates parse correctly before we even start our HTTP server, but for now let's handle the error a bit more gracefully.

```
tpl, err := template.ParseFiles(tplPath)
if err != nil {
  log.Printf("parsing template: %v", err)
  http.Error(w, "There was an error parsing the template.", http.StatusInternalServerError)
  return
}
```

The new code we added does three things:

1. It logs an error message out to the terminal.
2. It writes an error response to send to the user making the web request.
3. It returns, preventing any additional code in our `homeHandler` from executing.

When our code panicked, the end user wouldn't get a valid response. With our updated code they will see a message on the screen letting them know something went wrong. There will even be a `500` status code set in the response which was determined by the `http.StatusInternalServerError` constant we passed into `http.Error`.

The `return` is one of the most important lines in the new code we added. Without it, our code would continue trying to run the remainder of the `homeHandler` function which would result in another error because `t` is not a valid template to work with.

Restart your server and verify the code is working as intended. You should see an error response in your browser when you visit the home page, and a log message in theterminal. Once that is working, remove the `{{ invalidFunction }}` line from the `home.gohtml` file.

The next error we need to handle is the one that can occur when we attempt to execute our template. This type of error can occur if our template parses correctly, but execution fails for some reason. For instance, if the data we pass in is missing a field our template needs, we will see an error here.

Let's create this error in our code. Update `home.gohtml` with the following code.

```
<h1>Welcome to my awesome site!</h1>
{{ .Name }}
```

Next, update our code in `main.go` so it passes in data that doesn't have a `Name` field. A string should suffice.

```
// in homeHandler()
err = tpl.Execute(w, "a string")
```

Our code should panic when we visit the home page.

```
http: panic serving [::1]:60779: template: home.gohtml:2:3: executing "home.gohtml" at <.Name>:
... + more
```

We can handle this error more gracefully using code similar to what we used earlier in this lesson.

```
func homeHandler(w http.ResponseWriter, r *http.Request) {
        // ...
        err = tpl.Execute(w, "a string")
        if err != nil {
    log.Printf("executing template: %v", err)
                http.Error(w, "There was an error executing the template.", http.StatusInternal
                return
        }
}
```

Even though we are handling the **`tpl.Execute`** error, it is possible for a partial write to occur. In other words, we might see some of the HTML in our **`home.gohtml`** template written to our browser before our error message. This occurs because **`tpl.Execute`** will periodically write to the **`io.Writer`** passed in as it executes the template, meaning our **`http.ResponseWriter`** may have some of the template HTML written to it before the error occurs.

At this point we can remove the **`{{ .Name }}`** line in **`home.gohtml`**. We can also remove the **`"a string"`** bit being passed into **`tpl.Execute`** and revert it back to **`nil`**.

```
// in homeHandler()
err = tpl.Execute(w, nil)
```

Going back to our **`main.go`** source file, we can see that the updated **`homeHandler`** is significantly longer than it was before, and we haven't really added any new functionality. While we will look at ways to improve error handling as the course progresses, it is important to remember how important error handling

is. Without it, our code might silently fail leaving us questioning why it isn't working as expected.

When writing Go code, try to avoid ignoring `error` values, as this is an easy way to miss a bug in your code. If you absolutely must skip error handling until later, consider at least logging the error or using `panic` until it can be properly handled.

In an earlier iteration of this course I ignored the error returned by `tpl.Execute` with the plan to capture and handle it later in the course. Unfortunately, this lead to several confused students who couldn't figure out why their code was silently failing when they had bugs in their template.

Source Code

- Source Code - see the final source code for this lesson.
- Diff - see the changes made in this lesson.

4.8 Contact Page via Template

Moving our contact page to a template will be very similar to how we moved the home page to a template. We start by creating the template file.

```
code templates/contact.gohtml
```

Then we add some HTML to the template file.

```
<h1>Contact Page</h1>
<p>
  To get in touch, email me at
  <a href="mailto:jon@calhoun.io">jon@calhoun.io</a>.
</p>
```

In **main.go** our contact handler will end up looking nearly identical to the **homeHandler**. The only exception is which template file we use. As a result, you can copy/paste the **homeHandler** function and rename it **contactHandler** if you would like. Just be sure to update the **tplPath** to use the **contact.gohtml** file.

```
func contactHandler(w http.ResponseWriter, r *http.Request) {
        w.Header().Set("Content-Type", "text/html; charset=utf-8")
        tplPath := filepath.Join("templates", "contact.gohtml")
        tpl, err := template.ParseFiles(tplPath)
        if err != nil {
                log.Printf("parsing template: %v", err)
                http.Error(w, "There was an error parsing the template.", http.StatusInternalSe
                return
        }
        err = tpl.Execute(w, nil)
        if err != nil {
                log.Printf("executing template: %v", err)
                http.Error(w, "There was an error executing the template.", http.StatusInternal
                return
        }
}
```

At this point we are technically done. Our server now renders the contact page via a template, but as we noted before, the **contactHandler** and the **homeHandler** are nearly identical. Perhaps we can find a way to reduce the repetitive code a bit.

When learning to program, you are likely going to come across the acronym DRY. It stands for Don't Repeat Yourself, and the idea is that you probably don't want to repeat the same code over and over again a dozen times in your application. Code written this way can be a challenge to maintain as any changes need to be done in multiple places.

On the other hand, sometimes taking code and trying to make it reusable can result in such complex code that we have to stop and ask ourselves, "Is this really any easier to maintain?" In the words of Rob Pike, "A little copying is better than a little dependency." (source)

Repeating code on occasion isn't necessarily a bad thing. The copied code is

often easier to read and comprehend than an abstracted reusable version, and it is definitely easier to change if one of your use cases diverged from another.

Ultimately it ends up being a judgement call. Early on in your career you will likely make a few mistakes with these, but the goal is to get better at deciding when to break code up into something reusable and when it makes sense to copy/paste a bit.

Typically two copies of the same code isn't enough for me to warrant a reusable function. It is a sign that it might eventually be a candidate for a refactor, but it generally isn't enough information to know for sure.

In this particular case, even though we have only have two copies of the duplicate code, I believe we have enough information to justify a refactor. One of the big motivators here is that I know we can refactor this without making the code significantly more complex. Another factor is knowing that our application is likely to include additional static pages in the future.

Once we have decided we are going to refactor our code, the next step is figuring out how to break things apart. Should we have a helper for parsing templates, and another for executing? Or would it make more sense to create one single function to parse and execute a template?

In my mind it makes more sense to just have a single **executeTemplate** function that takes in the response writer, a template name, and handles everything, but if you are ever unsure you can try out both versions and see which seems like a better fit.

Inside **main.go** add the following function.

```
func executeTemplate(w http.ResponseWriter, filepath string) {
  w.Header().Set("Content-Type", "text/html; charset=utf-8")
        tpl, err := template.ParseFiles(filepath)
        if err != nil {
                log.Printf("processing template: %v", err)
                http.Error(w, "There was an error processing the template.", http.StatusInterna
                return
        }
        err = tpl.Execute(w, nil)
```

```
        if err != nil {
                log.Printf("executing template: %v", err)
                http.Error(w, "There was an error executing the template.", http.StatusInternal
                return
        }
}
```

Long term we might need to alter this to add in data, but for now we don't need it so I'll leave it out. One nice thing about a project like this is that we can let the code evolve rather than trying to predict needs ahead of time. This tends to lead to better software, as our predictions are usually way off.

Now we can use the **executeTemplate** function in both of our handler functions.

```
func homeHandler(w http.ResponseWriter, r *http.Request) {
        tplPath := filepath.Join("templates", "home.gohtml")
        executeTemplate(w, tplPath)
}

func contactHandler(w http.ResponseWriter, r *http.Request) {
        tplPath := filepath.Join("templates", "contact.gohtml")
        executeTemplate(w, tplPath)
}
```

Source Code

- Source Code - see the final source code for this lesson.
- Diff - see the changes made in this lesson.

4.9 FAQ Page via Template

The last page we need to convert to a template is the FAQ page. First, create the template file.

```
code templates/faq.gohtml
```

Add the HTML to the template.

```
<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>
```

And finally, update the FAQ handler function inside of **`main.go`**.

```
func faqHandler(w http.ResponseWriter, r *http.Request) {
  executeTemplate(w, filepath.Join("templates", "faq.gohtml"))
}
```

In this case we are opting to put the **`filepath.Join`** bit inline with the **`executeTemplate`** function call. This isn't any better or worse than using the **`tplPath`** variable like we did in the other handler functions; it just depends on which you find more readable, and in this particular case I think the inline version is more readable. I would typically update the other handler functions to match this style, but for now I will leave them be so you can see both working in the code.

Source Code

- Source Code - see the final source code for this lesson.

- Diff - see the changes made in this lesson.

4.10 Template Exercises

At this point you likely have a very rough idea of how templates work in Go. You aren't expected to understand them perfectly, but you should understand what they are used for at a high level.

If you are interested in further reading about templates, consider checking out the `html/template` docs and the following articles:

- An Introduction to Templates in Go - A free series I wrote about Go templates
- Writing Web Applications - A fairly short wiki provided by the Go team covering some of the basics for building web apps with Go. This includes using the `html/template` package.

If you want to read more about encoding you can do a search for "HTML character entities" or "HTML character codes". There is also a w3.org page that shows every character entity along with its code.

In this lesson we are going to do some exercises to solidify your knowledge. Some of the exercises require more challenging techniques, but don't be discouraged if you can't solve them all. Try your best and we will look at each technique later in the course as we need to use it.

4.10.1 Ex1 - Use template variables

Head over to `cmd/exp/` and experiment with using new variable names inside your template and update the Go code to work with the template.

You can approach this in two steps:

1. Adding a new field to the `User` type and updating the `user` variable in your Go code
2. Reference the new field inside of your `hello.gohtml` template

4.10.2 Ex2 - Experiment with different data types

Up until now we have mostly used strings in our HTML templates. Try experimenting with a few other data types like integers, floats, maps, and slices. Can you figure out how to iterate over a slice inside of your template and generate HTML for each element in it? What about a map - how do you access keys and values of a map?

We are using the `html/template` package, but some of the docs are better in the `text/template` package. For instance, the list of actions such as `{{range pipeline}}` are better in the `text/template` package. Luckily, all of these actions can be used the same way between the two packages, so you just need to read the `text/template` docs a bit.

4.10.3 Ex3 - Learn how to use nested data

Update your `User` type inside `cmd/exp/exp.go` to include nested data types, then experiment with using nested data like this in your template.

Next, try adding a `map` field to your `User` type. How do you access individual keys and values of that map? If you know the key for the value you want, how do you lookup a specific value in the map?

This may be a bit harder to figure out, especially if you have a limited background with templates and Go, but attempting to figure it out on your own will really help reinforce everything you are learning.

4.10.4 Ex4 - Create an if/else statement in your template

Check out the available actions in the template packages: https://golang.org/pkg/text/templat
Actions

Once you have an idea of what is available, update your template to use an `if` and an `else` statement inside of your template.

This will be similar to writing an if/else block inside of your Go code, but because we are writing in templates there will be some differences.

Chapter 5

Code Organization

5.1 Code Organization

We are going to take a quick detour to discuss code organization.

As you learn to program, you will eventually start to notice that some codebases are easy to navigate while others are confusing. At first it may be hard to tell what makes the difference, but over time you will quickly learn that a big factor is the code structure and how it is all organized.

If a developer needs to add a new feature or fix a bug in a codebase, it is much easier if they can fairly easily guess where to start digging and can easily trace things like a web request from start to finish. When that isn't possible, debugging can mean digging through significantly more source files to try to figure out what is or isn'‘t relevant.

A well structured project often makes it easier to determine where to start looking, even if you have never seen the code before.

There are countless ways to organize a Go codebase, but with every structure there is typically some organizational technique being applied with the goal of

making it easier for future developers to navigate the code. No single structure will be ideal for every team or project, so in this lesson we will explore a few code structures giving you a sample of some of the options. We will also talk about the one we are going to use early on in this course.

5.1.1 Flat structure

Some organization strategies are simple; so simple that you might not even consider it an organization strategy. A flat structure where all of our code is in a single package is one of these.

While a flat structure might seem like the exact opposite of an organizational strategy, it can actually work quite well for some smaller projects. The key here is to use file names to help organize your code. For instance, you might have the following:

- `user_store.go` - source code used to interact with users in a database or some other data store.
- `user_handler.go` - HTTP handlers related to users.

If we were imagining our photo gallery application with a flat structure, the directory might look something like this:

```
myapp/
  gallery_store.go
  gallery_handler.go
  gallery_templates.go
  user_store.go
  user_handler.go
  user_templates.go
  router.go
  ...
```

As we start to add various files to our codebase it becomes clear that, even though we don't have folders and packages, it isn't incredibly challenging to guess where various pieces of logic might be contained in our code. The organization comes from how we name our source files, and while it might not scale incredibly well, it could work reasonably well for a small to medium sized project.

5.1.2 Separation of concerns

Separation of concerns is an organization strategy where we organize our code based on its duties.

HTML and CSS is an traditional example of this. The HTML files focus on the overall structure of the data, while the CSS is focused on styling it. Each has a separate duty.

Aside 5.1. Separation isn't always good

HTML and CSS is a traditional example of the separation of concerns pattern, but ironically some developers have found that HTML and CSS aren't as decoupled as everyone initially thought. As a result, technologies like CSS-in-JS and libraries like Tailwind CSS have grown in popularity where the styling and the HTML structure aren't really separate.

Outside of HTML and CSS, Model-View-Controller (MVC) is the most common pattern that falls under separation of concerns. The basic idea with MVC is that code related to data is stored in a `models` directory, code related to rendering is stored in a `views` directory, and the code used to manage incoming requests and connect everything is stored in a `controllers` directory.

Below is an example of what our application might look like using MVC.

```
myapp/
  controllers/
    user_handler.go
    gallery_handler.go
    ...
  views/
    user_templates.go
    ...
  models/
    user_store.go
    ...
```

Buffalo uses what I consider to be a variation of MVC, but doesn't name directories exactly the same. Controllers are stored in an `actions` directory, and views are stored in a `templates` directory, and there are several other directories being used in addition to the three core MVC directories. It is important to remember that using MVC does not mean you are limited to only putting code in those three directories.

MVC is very common in frameworks for a variety of reasons, but one of those reasons is that it is predictable. It doesn't matter if you are building a banking app or time tracking app, any project using MVC will store the various types of files in the same directories making it easier for the framework tooling to guess where the controllers or views are in the source code. With that knowledge, frameworks can often do some magic behind the scenes to speed up initial development.

Examples of popular web frameworks that use MVC include Ruby on Rails (ruby) and Django (python).

5.1.3 Dependency based structure

Code can also be structured based on dependencies. This is a strategy taught by Ben Johnson on his site Go Beyond.

The idea with dependency based structure is to try to organize your code based

on what dependencies it has. For instance, if you used a PostgreSQL database, you might have a **`postgres`** package that has all of your postgres-related code. If you rely on the Stripe API, you might have a package named **`stripe`** with all of your code related to interacting with Stripe.

In order to put all of these dependencies together, code structured this way will often have a common set of interfaces and types that are dependency agnostic. Dependencies then use these interfaces for anything outside of their scope, and we can provide the actual implementations when setting up our application.

Let's look at an example. Imagine we had the following directories and source files.

Warning: *If some of the code here starts to go over your head don't worry about it. This organization pattern is a little more advanced and we will likely explore it again later in the course when you have a bit more experience.*

```
myapp/
  user.go
  user_store.go
  psql/
    user_store.go # implements the UserStore interface with a Postgresql specific implementatio
```

Inside **`user.go`** we might declare a **`User`** type. This type won't be specific to a database or anything else, it is just a User type that we can use anywhere in our application.

```
package app

type User struct {
  ID UserID
  Name string
  Email string
}
```

`user_store.go` might provide us with a **`UserStore`** interface that defines some common actions we take with users, but doesn't actually provide an implementation.

```
package app

type UserStore interface {
  Create(name, email, password string) (*User, error)
  // ...
}
```

At this point our code doesn't have any real implementation of the `UserStore` type, so we might provide a Postgres specific implementation inside `psql/user_store.go`

```
package psql

type UserStore struct {
  // ...
}

func (us *UserStore) Create(name, email, password string) (*app.User, error) {
  // Create a user in the Postgres database.
}
```

With this code structure nested code like that in `psql/user_store.go` will often reference types defined in parent packages (eg `app.User`), but the reverse is typically a code smell and can lead to cyclical dependency bugs.

While this code structure can work incredibly well when you get the hang of it, it is a bit more advanced and confusing, so it is generally not a great choice to initially learn with, and most developers have more luck adopting it after they have some experience with Go.

5.1.4 Plus many more...

There are countless other design structures that aren't explained here: domain driven design, onion architecture, and many more.

Each of these application structures can have similarities, and each might be a better fit depending on what you are building. Unfortunately, there isn't a

single best architecture. Instead, we as developers need to decide what fits our needs best from project to project.

When deciding what structure to use, it is important to remember that there isn't a one-size fits all structure. Some people may try to convince you that there is, but it simply isn't true. Each and every way to organize code has pros and cons.

I have seen small teams use a flat structure and iterate quickly, but I've also seen large projects that would be pretty awful as a flat structure.

Similarly, I've seen large teams use an organization pattern successfully, while I've seen smaller teams try to replicate that success with little luck. Whether it is team size, how well they understand their domain, needing to iterate faster or not needed the flexibility provided by a particular code structure, there are many reasons why one might not be a good fit for a project.

I have also found that experience has a big impact on what code structure works and doesn't work well for each individual. Developers with experience in a larger project will typically understand why global variables can be problematic, but developers who have mostly worked on smaller indy projects might not have the same opinion. When someone talks about whether or not a code is testable, typically only people who have written untestable code, discovered it wasn't testable, and finally refactored it into testable code will truly understand what is being said. The same is true for understanding and appreciating various code structures. Sometimes it takes experience to grasp the value.

When first starting with a project, I find that trying to decide on the perfect structure can lead to decision paralysis. Developers get so caught up in finding the perfect structure that they don't actually get to the building part. If you don't believe me, go read some of the history on Reddit. There are countless questions about code structure there, and in many cases developers let that hurdle prevent them from moving forward at all.

Rather than getting stuck on the structure, I believe it is better to start building and to remember that we can always refactor the code at a later date when we

understand the challenges we are facing a bit better.

In this course we are going to use MVC as we learn. It isn't perfect, but it...

- is predictable.
- will provide us with some clear rules to follow when organizing our code
- will prevent us from getting stuck trying to decide how to organize our code
- is very approachable regardless of experience level

Later, once we understand all of our application needs, we will be in a much better place to explore alternative design structures. At that time we will look into refactoring our codebase and using another code structure.

The key here is that we are going to do that refactoring after we understand everything a bit better; it is *really* hard to get code structure right when you don't have a deep understanding of what you intend to build, and chances are most students of this course don't have that understanding at this time. That doesn't mean we need to wait until a project is 100% complete to decide on the structure and refactor, but it helps to write some code and get a general sense of the project and its needs.

My goal here isn't to convince you to use MVC all the time, but to teach you something you can use and understand right now. I want to set you up for success, and give you a foundation to build from.

Now let's learn more about MVC so we can proceed with the course!

Source Code

- Source Code - see the final source code for this lesson.
- Diff - see the changes made in this lesson.

5.2 MVC Overview

Model-View-Controller, commonly referred to as MVC, is an architectural pattern for organizing code. With MVC the organization is sorted based on what the code is responsible for. This is oftentimes referred to as separation of concerns.

When using MVC, developers are ultimately responsible for putting code into the right location and making sure the MVC pattern is followed. As a result, differing opinions or mistakes can lead to two codebases that both use MVC, but end up with a very different organization of the code.

There are three distinct roles in MVC are **models**, **views**, and **controllers**. In our application we will start by breaking our code into three packages with these names.

Models are about data, including logic and rules for data. In practice this typically means models encompass code used to interact with the database, but it isn't limited to database code. Models could also entail code for interacting with files on the hard drive or data from a third party API.

Code classified as a model is not limited to fetching data. Models can also include code for validating and normalizing data. For example, our web application is going to have user accounts, and when we store those in our database we will want to make sure that an email address in all capital letters is not treated differently from one in lowercase. More specifically, the following two email addresses should be considered the same for account purposes:

```
jon@calhoun.io
JON@CALHOUN.io
```

We will solve this problem by converting all email addresses to lowercase before querying or inserting them into our database, and that code will be part of our models.

Views are all about rendering data. In our case we want to render HTML for people using our application, but we could also return JSON data via an API and store that code in a `views` package.

A big piece of advice about views is to keep the logic here as minimal as possible. One of the biggest mistakes beginners tend to make with MVC is placing unnecessary logic in their views code, and this can be very hard to test and maintain. That doesn't mean views have no logic in them at all, but it should be limited.

For example, if we are rendering a paginated list of images, our views might contain logic like, "If a next exists, show the next page link." It would be incredibly hard to create useful HTML pages without logic like this in our views. On the other hand, if your HTML page needs a bunch of data points for a graph it is rendering, that data should probably be calculated elsewhere (perhaps as part of the models code) while the views focus strictly on rendering the data after it has been calculated.

If we follow this pattern, verifying data calculations and similar tasks become easier, making our application easier to maintain. In the case of our graph example, if our views handled all of the data calculation we would be forced to verify that both the data calculation is accurate and the graph renders correctly at the same time. If one of these two steps breaks, we will need to check both to see where the bug is. On the other hand, if we broke these steps up and put the data calculation in our models code, we could verify that separately from the data rendering step in our views code.

Controllers contain the logic that connects everything together. In the case of a web request, controller code won't directly render HTML and it won't directly interact with a database or an API, but it will use code from both the models and views to handle incoming web requests.

Controllers are a bit like air traffic controllers; they don't fly planes, but they do coordinate with pilots to tell them which runway to land on, when to land, when it is clear to takeoff, etc. An air traffic controller is essential for a smooth

running airport because someone needs to be in charge of coordinating everything.

In practice, our HTTP handlers will be our controllers and they won't have much complex logic in them. Instead, our handlers will mostly parse incoming data then provide it to models and views code as needed. Our controllers will still need some logic to decide how to handle errors, what page to render, etc, but this will generally just be coordination logic.

As we progress through the course we will start by organizing our code with the MVC pattern, but over time as our needs change we might start introducing other packages and ways to organize our code.

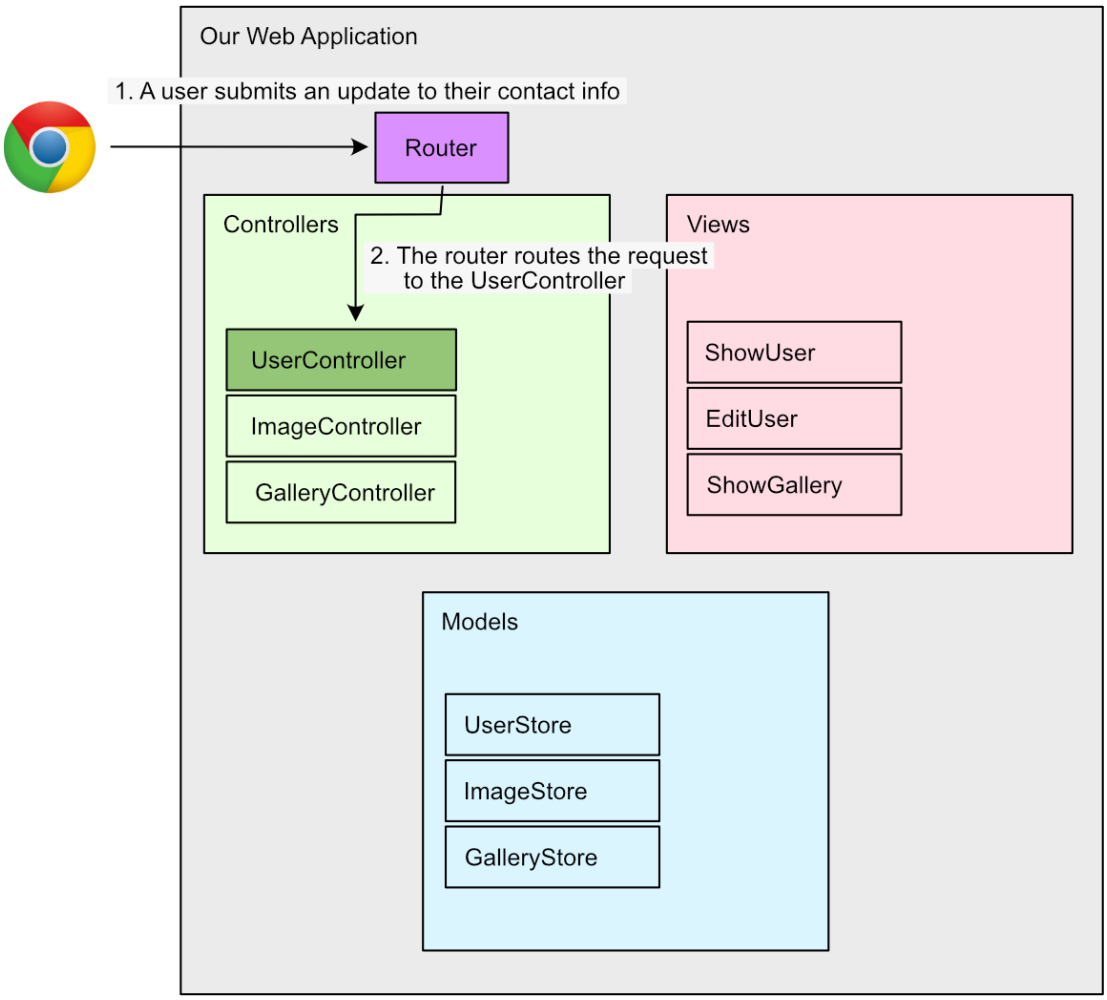
Source Code

- Source Code - see the final source code for this lesson.
- Diff - see the changes made in this lesson.

5.3 Walking Through a Web Request with MVC

MVC may be easier to understand if we take a normal web request and examine how each part of the request would be handled in a theoretical application. We haven't written all the pieces of code that we are going to talk about just yet, but seeing how data will flow through our application once we build it should help as we progress with building our application.

We will walk through what happens when a user submits an update to their contact information. For instance, they might be updating their address after moving.



1. A user submits an update to their contact information

The first step is pretty straightforward. The user submits their updated contact information to our server by submitting a form. Their browser then sends a web request to our server and our router is pretty much the first thing to interact with the request. Technically other code may run before the router, and we will see this later in the course, but for now we can just assume that all incoming requests are directed to the router as their first step.

There isn't really a specific place that routing needs to go in an MVC architec-

ture. It could technically fit into the controllers, but many applications separate routing from the controllers, and I have opted to do the same here.

2. The router routes the request to the UserController

When the router receives the web request, it looks at several pieces of information to decide how to proceed. In this case, it likely sees something like a PUT request to the `/user` endpoint and opts to send the request to some code in the `UserController` source code. This might be a method `Update` on a `UserController` type, or perhaps just an `UpdateUser` function.

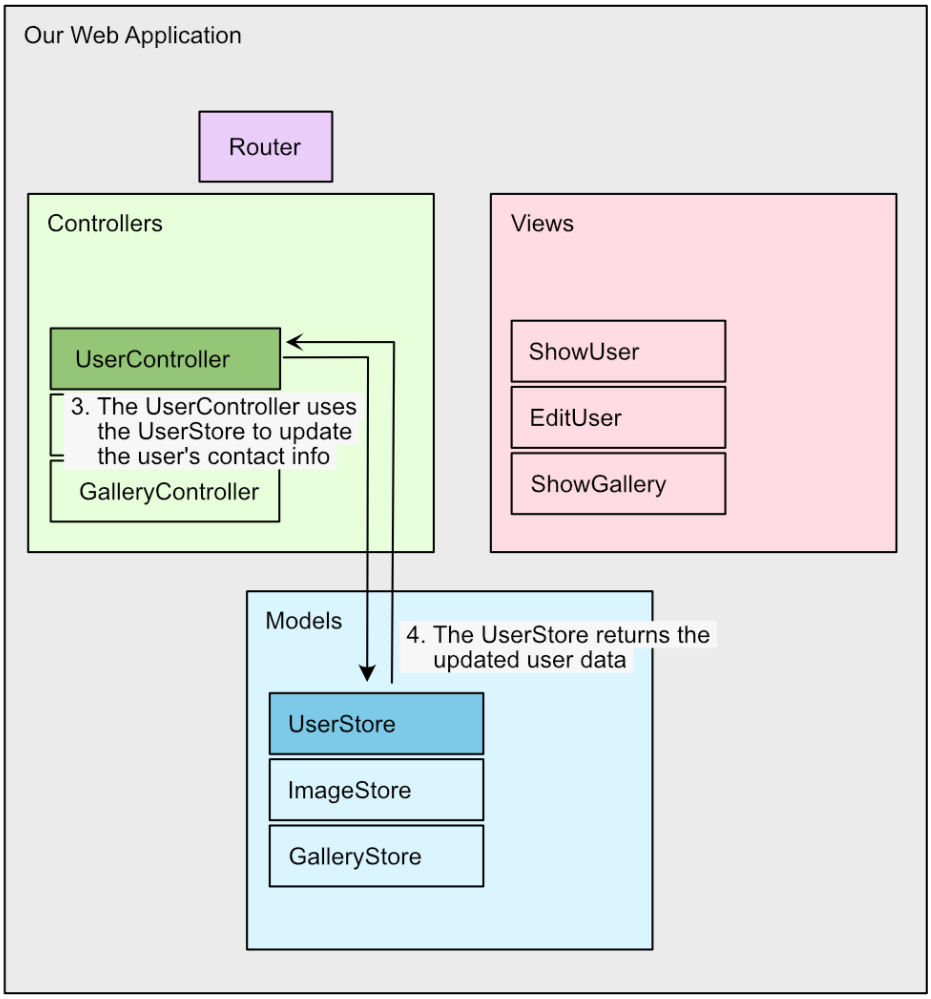
```
// Theoretical code. Don't add it to your project
package controllers

// Option A: UserController type with an Update method
type UserController struct {
  // ...
}

func (uc *UserController) Update(w http.ResponseWriter, r *http.Request) {
  // ...
}

// Option B: UpdateUser function
func UpdateUser(w http.ResponseWriter, r *http.Request) {
  // ...
}
```

This code is entirely theoretical at this point, but if I were working on an MVC web app I would be expecting something like those two options. Both options have merit, and it isn't vitally important that every MVC app uses the exact same naming rules.



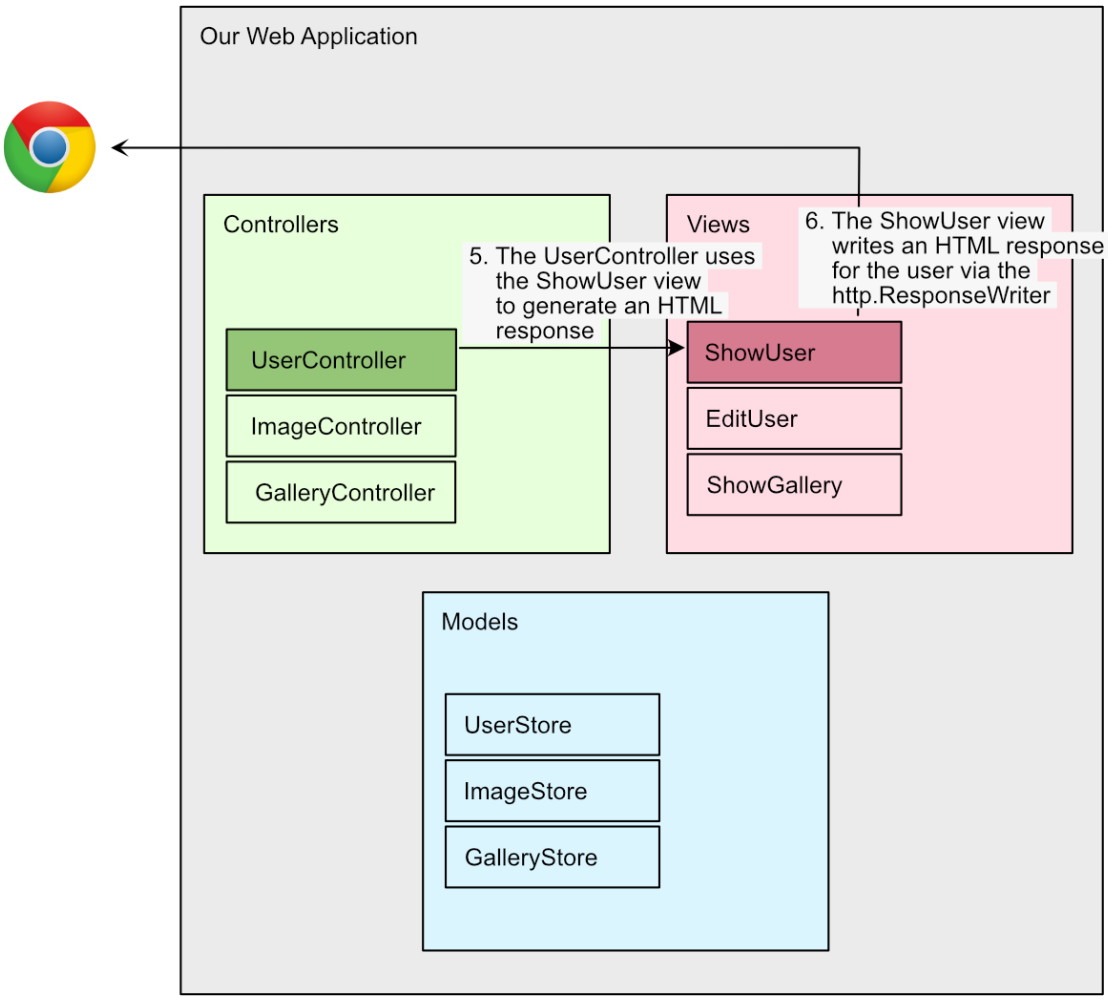
3. The UserController uses the UserStore to update the user's contact info

When the incoming web request gets passed along to the user controller code, it will need to do a few things. First, it needs to parse the incoming data. Second, it needs to update the user in the database with the new information provided.

Controllers don't interact directly with the database, so the user controller uses the `UserStore` provided by the `models` package to update the user's contact information. This allows us to isolate all of our database specific code from our controller code, making both easier to manage.

4. The UserStore returns the updated data

After updating the user in the database, the UserStore will return the updated user object to the controller. This sounds complex, but really this is just a function inside the `models` package returning data after it is called.



5. The UserController uses the ShowUser view to generate HTML

Once the user has been updated the last thing our controller needs to do is render a response to the user. Controllers don't create HTML directly, so our code

would use something like a `ShowUser` view to generate an HTML response that shows the newly updated contact information.

Notice that in all of these steps our controller is basically just calling code from the `views` and `models` packages. It rarely does any hard work on its own and instead acts more like a coordinator.

6. The ShowUser view writes and HTML response to the user

After the user controller uses the `ShowUser` view, HTML will be returned in a response to the web request. This will likely be something like their contact information page with the updated information and perhaps a message saying something like, "Your information has been updated!"

As we progress through the course we will be writing the code that allows us to use the same structure discussed here in our code. We will gradually start introducing packages, organizing our HTTP handlers together, isolating our database code together, etc. If any of this feels a bit unclear at this point, it will start to come together as we code everything.

Source Code

- Source Code - see the final source code for this lesson.
- Diff - see the changes made in this lesson.

5.4 MVC Exercises

This section hasn't contained any coding, and instead we have been learning about code structures and MVC. As a result, it would be hard to do any coding exercises. Instead, we are going to review a few questions related to MVC.

5.4.1 Ex1 - What does MVC stand for?

What does each letter in the MVC acronym stand for?

5.4.2 Ex2 - What is each layer of MVC responsible for?

Try to jot down what each layer of MVC is responsible for and review the chapter to see if you are correct.

5.4.3 Ex3 - What are some benefits and disadvantages to using MVC?

Take a moment to think about why MVC might help keep your code organized better than an ad-hoc structure.

Why might it be easier for others to help you work on your code?

Why is it easier to use when you aren't 100% certain what your final application will look like?

Can you think of any reasons why MVC might not be a great fit for some projects?

5.4.4 Ex4 - Read about other ways to structure code

Another exercise worth doing is to read up on other code structures.

Ben Johnsons gobeyond.dev has some interesting articles worth reading as well, but some of them may use code you don't quite understand at this point, so don't feel like you need to grasp it all to proceed.

Kat Zien also has a talk on this from a past Gophercon at https://www.youtube.com/watch?v=

Source Code

- Source Code - see the final source code for this lesson.
- Diff - see the changes made in this lesson.

Chapter 6

Starting to Apply MVC

6.1 Creating the Views Package

Our application currently has a some logic in the `main` package that relates to rendering HTML templates. Our first step in moving towards an MVC architecture is going to be moving that into a `views` package that handles all of our HTML rendering logic.

When organizing code, we don't need to get it perfect on the first try. If we were working on a codebase and were unsure what code should go where, we can always make a judgement call and refactor in the future if we feel we got it wrong.

Inside `main.go` we have the `executeTemplate` function. When we wrote the code, it made sense to create a single function. At the time we weren't parsing a template until we executed it, and it made sense to merge those two steps. As we saw with our code, the downside to this approach is that we don't find out that our template failed to parse until we attempt to render a page using it.

In an ideal world we would be able to parse out templates as our application

starts up, then start our web server which will use those templates to generate HTML. To do this, we are going to break our template logic into two steps:

1. Parsing a template
2. Executing a template

We are going to create our own `Template` type to handle all of this logic. This will be different from the `template.Template` type provided by `html/template`, as it our `views.Template` type will contain our custom logic for templates.

Let's start by creating the `views` directory and creating a Go file for this code.

```
mkdir views
code views/template.go
```

Inside `template.go` we first create declare the package and the `Template` type.

```
package views

import "html/template"

type Template struct {
        htmlTpl *template.Template
}
```

Inside of our `Template` type we added an `htmlTpl` field. This has the type `*template.Template` which comes from the `html/template` package. Adding this field will allow us to use the `html/template.Template` type in the custom logic we will be adding to our `Template` type.

While this may seem weird at first, it would be the same as if we created the following type:

```
// Don't code this
type View struct {
        htmlTpl *template.Template
}
```

We are opting to use the name **Template** because writing code later like **views.Template** will seem far more natural than **views.View**. The latter seems repetitive, and doesn't make it as clear what the type is used for.

Next we are going to add an **Execute** method to our new **Template** type. The code for this will be very similar to the code in our **executeTemplate** function, but we are going to use the **htmlTpl** field on our **Template** type and assume that the template is already parsed when **Execute** is called. We will also accept a **data interface{}** argument, which is the data used when rendering a template.

```
func (t Template) Execute(w http.ResponseWriter, data interface{}) {
        w.Header().Set("Content-Type", "text/html; charset=utf-8")
        err := t.htmlTpl.Execute(w, data)
        if err != nil {
                log.Printf("executing template: %v", err)
                http.Error(w, "There was an error executing the template.", http.StatusInternal
                return
        }
}
```

In our original **executeTemplate** code we parsed a template using a file path and then proceeded to use the template. We need to add code to our **views** package to do something similar. Again, some of this code will come directly from the **executeTemplate** function we already wrote, but we will need to make some tweaks to suit our needs.

```
func Parse(filepath string) (Template, error) {
        htmlTpl, err := template.ParseFiles(filepath)
        if err != nil {
    // Sidenote: We will discuss this `fmt.Errorf` in a
    // future lesson.
```

```
                return Template{}, fmt.Errorf("parsing template: %w", err)
        }
        return Template{
                htmlTpl: htmlTpl,
        }, nil
}
```

In this code we first start by declaring a function named `Parse`. This takes in a filepath and returns our custom `Template` type, and an error.

Inside the `Parse` function we first use the `html/template` package's `ParseFiles` function. This is identical to how we were parsing the HTML template in our original code, but what needs tweaked is how we handle errors. The purpose of this code is to parse a template at any time, not just while handling a web request. As a result, we don't have access to an `http.ResponseWriter` to write an error to, nor do we want to. Instead, we want to return an error if one occurs so that whoever calls the `Parse` function can decide how to proceed. We do this by returning a zero-value template (`Template{}`), and we use `fmt.Errorf` to wrap our error with a bit more context.

Aside 6.1. fmt.Errorf

We will discuss `fmt.Errorf` in a future lesson. For now, just know that it is a way of wrapping an error with additional information which can make debugging errors easier.

If our call to `template.ParseFiles` is successful, we instead return a `Template` with the `htmlTpl` field set to the `template.Template` that we parsed. Since this is a success case, we also return a `nil` error making it clear that nothing went wrong.

Putting this all together, we get the following code inside `template.go`:

```
package views

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

func Parse(filepath string) (Template, error) {
        htmlTpl, err := template.ParseFiles(filepath)
        if err != nil {
                return Template{}, fmt.Errorf("parsing template: %w", err)
        }
        return Template{
                htmlTpl: htmlTpl,
        }, nil
}

type Template struct {
        htmlTpl *template.Template
}

func (t Template) Execute(w http.ResponseWriter, data interface{}) {
        w.Header().Set("Content-Type", "text/html; charset=utf-8")
        err := t.htmlTpl.Execute(w, data)
        if err != nil {
                log.Printf("executing template: %v", err)
                http.Error(w, "There was an error executing the template.", http.StatusInternal
                return
        }
}
```

Long term we are going to parse templates before our web server starts up, but short term we want to verify that this code is working. The easiest way to do that is to head to the **`executeTemplate`** function inside **`main.go`** and refactor it to use this new code.

```
package main

import (
        // We want to add a new import similar to this one:
        "github.com/joncalhoun/lenslocked/views"
)

func executeTemplate(w http.ResponseWriter, filepath string) {
```

```
        tpl, err := views.Parse(filepath)
        if err != nil {
                log.Printf("parsing template: %v", err)
                http.Error(w, "There was an error parsing the template.", http.StatusInternalSe
                return
        }
        tpl.Execute(w, nil)
}
```

This is the first time we are importing code that we wrote, so check to make sure everything is working as expected. If you initialized your Go module as something other than **`github.com/joncalhoun/lenslocked`**, your import path will have that prefix instead. Ideally your editor will handle the import for you and avoid any troubles, but if you are having an issue open your **`go.mod`** file and look for the first line in it:

```
module github.com/joncalhoun/lenslocked
```

The module listed here is going to be the prefix for all of our custom package imports when we import other packages from within this codebase. It doesn't matter if your github username isn't **`joncalhoun`**, if you are working on a project with that module name then this will prefix your local imports.

If your imports are in order, start your application up and verify that each page is working. If you would like, you can add **`fmt.Println`** statements inside the **`Parse`** and **`Template.Execute`** functions we created to verify they are being used.

Source Code

- Source Code - see the final source code for this lesson.
- Diff - see the changes made in this lesson.

6.2 fmt.Errorf

When creating our **Parse** function inside our **views** package, we could have returned the error we received without doing anything to it:

```
htmlTpl, err := template.ParseFiles(filepath)
if err != nil {
  return Template{}, err
}
```

While this works, we can improve it by adding more context about what we were doing when the error occurred. This is easier to see with an example, so let's write some code to illustrate this.

Open up **cmd/exp/exp.go** and replace its contents with the following code:

```
package main

import (
        "errors"
        "log"
)

func Connect() error {
        // try to connect
        // pretend we got an error
        return errors.New("connection failed")
}

func CreateUser() error {
        err := Connect()
        if err != nil {
                return err
        }
        return nil
}

func CreateOrg() error {
        err := CreateUser()
        if err != nil {
                return err
        }
        return nil
```

```
}

func main() {
        err := CreateUser()
        if err != nil {
                log.Println(err)
        }
        err = CreateOrg()
        if err != nil {
                log.Println(err)
        }
}
```

This code can be run on the Go Playground if you do not want to code it. We won't be using it for anything other than illustrative purposes.

In the code we added we have a `Connect` function that always returns an error, but we can imagine a scenario where perhaps a service is experiencing an outage and `Connect` only results in an error during that time.

Next we have the `CreateUser` function which relies on `Connect`. We can pretend that this is something like connecting to a service to create a new user account, but that service it experiencing an outage which results in an error.

This code also has a `CreateOrg` function which calls the `CreateUser` function. A situation like this might occur if someone is signing up for a new piece of software and is creating both the organization and the user account at the same time.

If we run this code, we can see that calling `CreateUser` directly, or calling `CreateOrg` both result in the same error being output to our terminal.

```
go run cmd/exp/exp.go
```

We don't currently have any way to really clarify what was going on when the error occurred. A common technique to address this is to wrap errors with additional context. In the past this was done with third party libraries, but Go 1.13

added new ways to work with errors. Most notably, we can use **`fmt.Errorf`** to add wrap an error with additional context.

Below is the same code as before, but with additional context via **`fmt.Errorf`**.

```
func Connect() error {
  // try to connect
  // pretend we got an error
  return errors.New("connection failed")
}

func CreateUser() error {
  err := Connect()
  if err != nil {
    // We can add more context here!
    return fmt.Errorf("create user: %w", err)
  }
  return nil
}

func CreateOrg() error {
        err := CreateUser()
        if err != nil {
                return fmt.Errorf("create org: %w", err)
        }
        return nil
}
```

This code can also be run on the Go Playground: https://go.dev/play/p/wZqYkDpIiES

With the additional context it now becomes clear where the error originated. Running this code gives us output like below.

```
2022/02/24 15:33:42 create user: connection failed
2022/02/24 15:33:42 create org: create user: connection failed
```

In the second case we can now see that the error occurred while creating an organization.

If you have ever used **`panic`** in your code, you are likely familiar with the stack trace output by it. For instance, if we updated **`Connect`** to panic we might get output like below.

```
panic: connection failed

goroutine 1 [running]:
main.Connect(...)
        /Users/jon/dev/courses-code/lenslocked/cmd/exp/exp.go:11
main.CreateUser(...)
        /Users/jon/dev/courses-code/lenslocked/cmd/exp/exp.go:16
main.main()
        /Users/jon/dev/courses-code/lenslocked/cmd/exp/exp.go:33 +0x28
exit status 2
```

While this does provide additional information, it also means that any code calling our `CreateUser` function would have a hard time handling the error. For instance, if we wanted to implement a wait and retry in our code, it would be harder to do with a panic.

Using `fmt.Errorf` will also enable us to do more advanced error handling in the future. Most notably, we will be able to determine if the underlying error was a specific type of error that we might handle in a unique way.

We will learn more about all of this as we progress through the course and use errors more. For now just remember that you can use `fmt.Errorf` to wrap errors with additional context.

Source Code

- Source Code - see the final source code for this lesson.
- Diff - see the changes made in this lesson.

6.3 Validating Templates at Startup

Inside `main.go` our `executeTemplate` function currently parses the template file (`.gohtml`) every time a web request is made. While this works, it has a few downsides:

1. We parse the same template multiple times even though it isn't changing between requests.
2. We don't know if a template failed to parse until we try to view the corresponding page.
3. Our error handling has to be done while rendering the response to a request, rather than during startup.

The last issue here is probably the most important one. If a template is invalid, it doesn't help us to try parsing it again in a future web request. Without someone updating the template, it will continue to fail to parse. A better setup would be to parse and validate templates before our web server even starts, that way we know ahead of time and can act accordingly.

Our HTTP handlers will need to have access to an already-parsed template in order for us to use this approach. There are a few ways we could make this happen:

1. We can use global variables
2. We can use a custom type with a field for our template
3. We could use a closure to create our handler function

Global variables are generally frowned upon in production-ready code. They can make sense in a few situations, and for smaller side projects they likely won't present an issue, but as a project scales in size global variables can make testing and maintaining code harder. As a result, we won't be using this approach for our templates.

The last two options - a custom type, or a closure - are both good options so we will explore how both work. In both cases we will want to place this code inside a `controllers` package, so we can start by creating the directory and a Go source file.

```
mkdir controllers
code controllers/static.go
```

The custom type approach is exactly what it sounds like. We create a custom type, add a field to it that stores our template, then add an HTTP handler method to that type.

Open **static.go** and add the following code.

```
package controllers

import (
  "net/http"

  "github.com/joncalhoun/lenslocked/views"
)

type Static struct {
  Template views.Template
}

func (static Static) ServeHTTP(w http.ResponseWriter, r *http.Request) {
  static.Template.Execute(w, nil)
}
```

This creates the **Static** type which can be used as an **http.Handler** interface implementation. It also has access to a **views.Template** via the **Template** field.

Now we can parse a **views.Template** inside of our **main()** function, construct a **controllers.Static** instance, and pass that into our Chi router to serve one of our HTML pages.

Open **main.go** and add the following code.

```
func main() {
  r := chi.NewRouter()
```

```
  homeTpl, err := views.Parse(filepath.Join("templates", "home.gohtml"))
  if err != nil {
    panic(err)
  }
  r.Method(http.MethodGet, "/", controllers.Static{
    Template: homeTpl,
  })

  // ...
}
```

While our `Static` type does implement the `http.Handler` interface (because it has the `ServeHTTP` method), Chi's `r.Get` function only works with HTTP handler functions. Instead, we will need to use `r.Method()`.

You can look up the details for the `r.Method` function in the Chi docs, but the short explanation is that this works similar to fucntions like `r.Get`, except we need to pass in an HTTP method as the first argument, and the last argument should be an `http.Handler` implementation rather than an HTTP handler function.

The last approach we can try is creating a closure. We haven't talked about closures yet, but they are functions created dynamically that have access to data declared outside of the function. In our case, that data will be the parsed template.

Aside 6.2. Additional Closure Resources

Closures are used quite frequently in Go, so it is worth taking some time to become familiar with them. The following free articles can help with that:

- What is a Closure?
- 5 Useful Ways to Use Closures in Go

Head back over to **static.go** and add the following code:

```
func StaticHandler(tpl views.Template) http.HandlerFunc {
  return func(w http.ResponseWriter, r *http.Request) {
    tpl.Execute(w, nil)
  }
}
```

Inside **StaticHandler** we are creating and returning a new function that matches the **http.HandlerFunc** definition, but because we have access to the **tpl** variable, we can also use that inside of the function we are creating. In a way this feels like using global variables, but it happens at a much narrower scope and it doesn't have all the shortcomings of global variables.

Using this in **main.go** gives us the following code:

```
func main() {
  r := chi.NewRouter()

  tpl, err := views.Parse(filepath.Join("templates", "home.gohtml"))
  if err != nil {
    panic(err)
  }
  r.Get("/", controllers.StaticHandler(tpl))

  tpl, err = views.Parse(filepath.Join("templates", "contact.gohtml"))
  if err != nil {
    panic(err)
  }
  r.Get("/contact", controllers.StaticHandler(tpl))

  tpl, err = views.Parse(filepath.Join("templates", "faq.gohtml"))
  if err != nil {
    panic(err)
  }
  r.Get("/faq", controllers.StaticHandler(tpl))

  r.NotFound(func(w http.ResponseWriter, r *http.Request) {
    http.Error(w, "Page not found", http.StatusNotFound)
  })
  fmt.Println("Starting the server on :3000...")
  http.ListenAndServe(":3000", r)
}
```

`controllers.StaticHandler` returns an `http.HandlerFunc`, so we can go back to using `r.Get` for our router. I have also opted to remove the `controllers.Stat` approach we used earlier in this lesson. While both approaches work, I prefer to be consistent in my code and opted to only use the `controllers.StaticHandler` closure approach in the end.

Lastly, we no longer need the rest of the code in our `main.go` source file. `executeTemplate`, `homeHandler`, `contactHandler`, and `faqHandler` can all be deleted as this logic was moved or replaced during this refactor. The resulting `main.go` source is shown below.

```
package main

import (
        "fmt"
        "net/http"
        "path/filepath"

        "github.com/go-chi/chi/v5"
        "github.com/joncalhoun/lenslocked/controllers"
        "github.com/joncalhoun/lenslocked/views"
)

func main() {
        r := chi.NewRouter()

  tpl, err := views.Parse(filepath.Join("templates", "home.gohtml"))
        if err != nil {
                panic(err)
        }
        r.Get("/", controllers.StaticHandler(tpl))

        tpl, err = views.Parse(filepath.Join("templates", "contact.gohtml"))
        if err != nil {
                panic(err)
        }
        r.Get("/contact", controllers.StaticHandler(tpl))

        tpl, err = views.Parse(filepath.Join("templates", "faq.gohtml"))
        if err != nil {
                panic(err)
        }
        r.Get("/faq", controllers.StaticHandler(tpl))

        r.NotFound(func(w http.ResponseWriter, r *http.Request) {
                http.Error(w, "Page not found", http.StatusNotFound)
```

```
        })
        fmt.Println("Starting the server on :3000...")
        http.ListenAndServe(":3000", r)
}
```

Be sure to verify all of your pages still work before proceeding to the next lesson.

Source Code

- Source Code - see the final source code for this lesson.
- Diff - see the changes made in this lesson.

6.4 Must Functions

While coding in Go, we will occasionally encounter errors. Maybe our database goes offline, or perhaps we have a typo in our template that prevents it from parsing. When this occurs, we need to decide how to go about handling the error. Most notably, should we use the `panic()` function, or return an error?

As a general rule, I reserve `panic` for special cases where an error shouldn't occur, and if an error does occur, it suggests that a programmer made a mistake. For instance, if we were to access an invalid index in a slice, that isn't an error we can really recover from. Somewhere in our code we have faulty logic.

We can see this with an example. Open `cmd/exp/exp.go` and update it with the following code:

```
package main

import "fmt"

func main() {
```

```
        numbers := []int{1, 2, 3}
        fmt.Println(numbers[4])
}
```

If we run this code, our code panics and prints out a runtime error.

```
panic: runtime error: index out of range [4] with length 3

goroutine 1 [running]:
main.main()
        /Users/jon/Dev/courses-code/lenslocked/cmd/exp/exp.go:7 +0x1d
exit status 2
```

On the other hand, an error should be returned anytime the failure state can be reached with completely valid code. This makes it clear to developers that there might be an error and they can either opt to handle it however they see fit.

For instance, if we look back at the `Connect` function we had in our `exp.go` file:

```
func Connect() error {
        return errors.New("connection failed")
}
```

If we call `Connect()` within our code, it is clear that this code might result in an error and we can decide how to handle it.

```
func main() {
  // We know Connect can result in an error because it returns an error
  err := Connect()
  if err != nil {
    // handle the error or panic if we want
  }
}
```

On the other hand, imagine that the `Connect` function used panic instead.

```
func Connect() {
        panic("connection failed")
}

func main() {
  Connect()
}
```

With the panic version, we have no indicators that this code can fail. It doesn't return an error, and unless we read the source code for `Connect`, we likely won't realize it can result in a panic until it happens.

Going back to our `main.go` source file, we are using `panic` every time we attempt to parse a template file:

```
tpl, err := views.Parse(filepath.Join("templates", "home.gohtml"))
if err != nil {
  panic(err)
}
```

In this case we are opting to use `panic` because these errors will almost always be the result of a programmer error. For instance, if we have a typo in our template that makes parsing fail, that is a programmer error.

With functions like our `views.Parse` where most errors are the result of a programmer error, a common pattern in Go is to also provide a must function that makes using it easier. These can come in two variants:

1. When many functions in the package return a specific type or an error (eg `(Thing, error)`), the must function will often take in the same arguments and panic if the error is not nil. Otherwise it returns the original type.

2. Must functions may also match the name of the function they are wrapping, panicking if the error is not nil and returning the remainder of the return values otherwise.

We can see examples of both of these in the standard library. The first is demonstrated in the **`html/template`** package's Must:

```
func Must(t *Template, err error) *Template {
        if err != nil {
                panic(err)
        }
        return t
}
```

The second can be seen in the **`regexp`** package with Compile and MustCompile:

```
func Compile(expr string) (*Regexp, error) {
  // ...
}

func MustCompile(str string) *Regexp {
        regexp, err := Compile(str)
        if err != nil {
                panic(`regexp: Compile(` + quote(str) + `): ` + err.Error())
        }
        return regexp
}
```

Must functions are sometimes frowned upon because they result in a panic, but as we discussed earlier, there are legitimate times where panicking is the correct choice. When writing code where panicking with an error is a common occurrence, providing a must function can drastically simplify the resulting code.

In our code the **`views.Parse`** function could benefit from a must function. We only use it when starting up our application and parsing templates for the first time, and we know that almost all errors that occur during a call to **`Parse`** are developer errors. We can even verify this by looking at our existing code and seeing that this is indeed what we do inside **`main()`** every time **`Parse`** returns an error.

Let's go ahead and add a **`Must`** function to our code. Open up **`views/template.go`** and add the following code:

```
func Must(t Template, err error) Template {
        if err != nil {
                panic(err)
        }
        return t
}
```

Now return to **`main.go`** and clean up the code in the **`main()`** function:

```
tpl := views.Must(views.Parse(filepath.Join("templates", "home.gohtml")))
r.Get("/", controllers.StaticHandler(tpl))
// Or inline everything and skip the `tpl` variable.
r.Get("/contact", controllers.StaticHandler(
        views.Must(views.Parse(filepath.Join("templates", "contact.gohtml")))))
r.Get("/faq", controllers.StaticHandler(
        views.Must(views.Parse(filepath.Join("templates", "faq.gohtml")))))
```

It is also a good idea to add some invalid code to one of your templates and verify that the server panics and does not start the web server in this case. Once that is all working revert to the correct templates and proceed to the next lesson.

Source Code

- Source Code - see the final source code for this lesson.
- Diff - see the changes made in this lesson.

6.5 Exercises

Let's look at some exercise problems to help reenforce what we have learned.

6.5.1 Ex1 - Add a new static page to your app

We introduced the **views** package and the **controllers** package, along with functions and types that allow us to create new static pages. Try adding a few more to your application to make sure you understand how everything works.

6.5.2 Ex2 - Experiment a bit with errors

At this point we have seen how to use **fmt.Errorf** to wrap errors.

```
func B() error {
        err := A()
  if err != nil {
          return fmt.Errorf("b: %w", err)
  }
  return nil
}
```

This adds additional context if an error gets logged, but what if we need to determine whether the underlying error was of a specific type? For instance, what if we need to know whether an error stemmed from an **ErrNotFound** error?

```
package main

import (
        "errors"
        "fmt"
)

func main() {
        err := B()
  // TODO: Determine if the `err` variable is an `ErrNotFound`
}

var ErrNotFound = errors.New("not found")

func A() error {
```

```
        return ErrNotFound
}

func B() error {
        err := A()
  if err != nil {
         return fmt.Errorf("b: %w", err)
  }
  return nil
}
```

Look up Go's **`errors`** package and its documentation to see if you can learn how to determine if a wrapped error is a specific error variable.

*Hint: If you get stuck, look at **`errors.Is`**.*

As a followup, read about **`errors.As`** and see if you can use it or think of cases where it might be useful.

Source Code

- Source Code - see the final source code for this lesson.
- Diff - see the changes made in this lesson.

Chapter 7

Enhancing our Views

7.1 Embedding Template Files

Our code currently parses template files from the local file system as it starts up. Once they are parsed it will no longer check for file updates, but the templates need to be there when the app starts up. We can see this if we build our binary and then run the binary from somewhere else.

First we build a binary named `app`. We do this by using the `go build` command and giving it the `-o` flag which is followed by our binary name.

```
# This builds a binary named "app"
go build -o app .
```

Aside 7.1. Windows may need .exe

If you are using windows, you may need to replace `app` with `app.exe`. Both MacOS and Linux should work fine without any extension.

We now have a binary in our file system named **app**. We can see this if we open up the directory or run **ls** in the terminal.

We can now run this binary in our terminal:

```
./app
```

This should start our server up and it will start serving on port 3000. If you have **modd** running, you may need to stop it so it frees up that port, as only one application can use a port at a time.

If we were to run this binary from somewhere else, we would see an error about the template files not existing.

```
cd ..
./lenslocked/app # error!
```

As we learned earlier, paths in our code are relative to where the code is run from, and as we discussed in this lesson, our application needs to parse template files (our **.gohtml** files) when it starts up.

Before proceeding, clean up the binary file we created.

```
cd lenslocked # move back to the lenslocked dir
rm app # deletes the app binary
```

One way to address this issue is to embed the template files into the binary that we build. This would allow us to run the binary from anywhere because the templates would no longer be read from disk; instead they would be read from the binary itself.

While this will increase the size of our final binary, it makes it much easier to release our binary without needing to also upload template files to our server.

It can also be useful when building CLIs that other developers will run on their own computers.

Aside 7.2. How does embedding work?

Imagine you had a small HTML template and you wanted to include it in your code without reading a file from disk. How would you make this happen?

One way is to create a string variable and set its contents to the HTML template you want to include. This would allow the rest of your code to read from the string, and you wouldn't need to read from disk at all.

Embedding works using a similar technique. At compile time, any files that are meant to be embedded are turned into code that is included in your program. There is a bit more going on behind the scenes because file names and paths may need preserved, but the idea is roughly the same. All of the embedded files are read and included into the binary we build. As a result, our binary will be larger because it needs to include all of our file contents.

In past versions of Go it was possible to embed files into a binary, but third party libraries needed to be used. In Go 1.16 the embed package was introduced, which makes embedding files easy with the standard library.

Our templates are a good fit for embedding for a few reasons:

1. They are typically fairly small, so they won't bloat our binary
2. Templates are often crucial for our app to work correctly
3. Defined before compile time by developers so we can embed them

Embedding our templates will also make deployment easier since we won't need upload template files to our production server.

Let's start by creating a Go source file in the `templates` directory. We will use this file to tell the compiler about our embedded templates, and in our code to access those embedded files.

```
code templates/fs.go
```

The name `fs` will make more sense as we start writing our code. Open up the file and add the following Go code.

```
package templates

import "embed"

//go:embed *
var FS embed.FS
```

In Go, there are some special comments that are called directives. These are used to provide the compiler with some information at compile-time. One of these is the `//go:embed` comment. It tells the embed package that we want to embed some files at compile-time and then store those files in the variable declared directly after the comment.

In our code, the two lines doing this are shown below.

```
//go:embed *
var FS embed.FS
```

The `*` is what is known as a glob pattern. It helps us define which types of files to match. We could have used something like `*.gohtml` which would have only included files with the `.gohtml` extension, but we are opting to embed all files including our `fs.go` source file into the `FS` variable for now.

If we want to access any of the files inside the `templates` directory, we can now do so via the `FS` variable.

Aside 7.3. Embed filepaths use the forward slash

We didn't need a slash in our embed file path, but imagine that we wanted to embed some templates in a nested directory. Say `images/*.{jpg,png}`, which is all files in the `images` directory ending with the `jpg` or `png` extension. In the past we needed to use operating system specific slashes; forward slash for MacOS and Linux, and backslash for Windows. With the embed package, all paths use the forward slash.

One reason for this is that we cannot use `filepath.Join` inside of a comment, and another is that this just makes for a more universal experience when accessing embedded files.

Now, how do we go about using these embedded files?

Thus far we have been using the `html/template` package's `ParseFiles` function. We can see this by opening up `views/template.go`. With the release of the `embed` package in Go 1.16, the `html/template` package also had a new function added - ParseFS.

```
func ParseFS(fs fs.FS, patterns ...string) (*Template, error)
```

`ParseFS` works like `ParseFiles`, except it needs something that matches the fs.FS type as its first argument. `fs.FS` comes from the `io/fs` package, and it is an interface.

```
type FS interface {
        // Open opens the named file.
        //
        // When Open returns an error, it should be of type *PathError
        // with the Op field set to "open", the Path field set to name,
        // and the Err field describing the problem.
```

```
        //
        // Open should reject attempts to open names that do not satisfy
        // ValidPath(name), returning a *PathError with Err set to
        // ErrInvalid or ErrNotExist.
        Open(name string) (File, error)
}
```

The `FS` variable we created in `templates/fs.go` has the type `embed.FS` - will this work here?

To check this, we only need to see if `embed.FS` implements the `Open()` function required by `fs.FS`. We can see in the docs that it does, so we can pass our `embed.FS` into the `template.ParseFS` function!

Putting this all together, we can add the following function to our `views/template.go` source file.

```
func ParseFS(fs fs.FS, pattern string) (Template, error) {
        htmlTpl, err := template.ParseFS(fs, pattern)
        if err != nil {
                return Template{}, fmt.Errorf("parsing template: %w", err)
        }
        return Template{
                htmlTpl: htmlTpl,
        }, nil
}
```

The code we added is nearly identical to the `Parse` function we created, except our new code uses `template.ParseFS` and accepts an `fs.FS` argument.

The last step is to head to `main.go` and update our code to use the new `ParseFS` function with the `embed.FS` we set up.

```
package main

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

```
        "github.com/go-chi/chi/v5"
        "github.com/joncalhoun/lenslocked/controllers"
        "github.com/joncalhoun/lenslocked/templates"
        "github.com/joncalhoun/lenslocked/views"
)

func main() {
        r := chi.NewRouter()

        r.Get("/", controllers.StaticHandler(
                views.Must(views.ParseFS(templates.FS, "home.gohtml"))))
        r.Get("/contact", controllers.StaticHandler(
                views.Must(views.ParseFS(templates.FS, "contact.gohtml"))))
        r.Get("/faq", controllers.StaticHandler(
                views.Must(views.ParseFS(templates.FS, "faq.gohtml"))))

        r.NotFound(func(w http.ResponseWriter, r *http.Request) {
                http.Error(w, "Page not found", http.StatusNotFound)
        })
        fmt.Println("Starting the server on :3000...")
        http.ListenAndServe(":3000", r)
}
```

It is also worth noting that we no longer need to worry about the `templates` directory part of our pattern because our embedded file system was built inside the `templates` directory.

Restart your server and verify everything is working.

Now if we were to build a binary and run it from somewhere else it would continue to work even if the `.gohtml` files aren't present. These are now included in the binary we are building with Go.

Source Code

- Source Code - see the final source code for this lesson.
- Diff - see the changes made in this lesson.

7.2 Variadic Parameters

When defining our **ParseFS** function, we only allow a single pattern to be passed into the function.

```
// views/template.go
func ParseFS(fs fs.FS, pattern string) (Template, error) {
  // ...
}
```

If we were to look at the **ParseFS** function inside of the **html/template** package, we would see that it accepts something a bit different.

```
// inside the html/template package
func ParseFS(fs fs.FS, patterns ...string) (*Template, error)
```

Using three dots (**...**) before a type turns it into a variadic parameter. What this does is allows developers to pass in zero or more arguments of that type. In other words, the following are all valid ways to call the **template.ParseFS** function.

```
// Zero patterns. Not very useful in this case, but the code will compile
template.ParseFS(embed.FS)

// One pattern. What we do now for our static pages.
template.ParseFS(embed.FS, "home.gohtml")

// Several patterns. Especially useful if you need to include additional
// files that have reusable HTML components, like an alert.
template.ParseFS(embed.FS, "home.gohtml", "footer.gohtml", "alert.gohtml")
```

Variadic parameters can accept a variable number of arguments, so they must be the last parameter defined in a function. The following is NOT valid.

```
// NOT valid!
func ParseFS(fs fs.FS, patterns ...string, n int)
```

Let's write our own function with a variadic parameter to explore how they work a bit further. Open **cmd/exp/exp.go** and add the following code.

```
package main

import (
        "fmt"
)

func main() {
        Demo()
        Demo(1)
        Demo(1, 2, 3)
}

func Demo(numbers ...int) {
        for _, number := range numbers {
                fmt.Print(number, " ")
        }
        fmt.Println()
}
```

When we call a function that uses a variadic parameter, pass individual values into the function call and just include as many as we want. Here we can see that we call **Demo()** with no numbers, just the single **1** value, and a third time with **1**, **2**, and **3** as if the function took three arguments.

```
Demo()
Demo(1)
Demo(1, 2, 3)
```

Inside the function is a bit different because we don't know for sure how many arguments we are going to receive. To account for this, our code always needs to treat the variadic parameter as a slice. That means it can be ranged over like any other slice.

```
func Demo(numbers ...int) {
        for _, number := range numbers {
                fmt.Print(number, " ")
        }
        fmt.Println()
}
```

We can also access the length of the slice and lookup values at a specific index, just like any other slice. Add the following code to your **exp.go** to see this in action:

```
func main() {
        fmt.Println(Sum())
        fmt.Println(Sum(4))
        fmt.Println(Sum(4, 5, 6))
}

func Sum(numbers ...int) int {
        sum := 0
  for i := 0; i < len(numbers); i++ {
                sum += numbers[i]
        }
  return sum
}
```

To summarize, a function that accepts a variadic parameter will accept individual values and converts them into a slice for your function to use.

What happens if we already have a slice and need to pass that into a function with a variadic parameter? We can also use the triple dots (**...**) to "unravel" a slice into individual arguments. Add the following code to **exp.go** to see this in action.

```
func main() {
        fib := []int{1, 1, 2, 3, 5, 8}
        Demo(fib...)
        fmt.Println(Sum(fib...))
}
```

In these example we have been looking at `int` variadic parameters, but any data type can be used. `template.ParseFS` uses strings, and we can create a demo function for those as well:

```
func main() {
        strings := []string{"the", "quick", "brown", "fox"}
        fmt.Println(Join(strings...))
}

func Join(vals ...string) string {
        var sb strings.Builder
        for i, s := range vals {
                sb.WriteString(s)
                if i < len(vals)-1 {
                        sb.WriteString(", ")
                }
        }
        return sb.String()
}
```

All of this begs the question - why don't we just accept a slice as our last argument? We could use functions like:

```
func Demo(numbers []int)
func Sum(numbers []int) int
func Join(vals []string) string
```

The primary motivation for using variadic parameters is to make a developers life easier, and they will most frequently be used in cases where developers are commonly hand coding the function call. Take our `ParseFS` function - this is very frequently used with hard-coded strings defining what the template file is on disk. As a result, it makes sense to accept a variadic parameter to make the developers life easier.

We are going to update our `ParseFS` function to accept a variadic parameter. Open up `views/template.go` and make the following changes:

```
func ParseFS(fs fs.FS, pattern ...string) (Template, error) {
        htmlTpl, err := template.ParseFS(fs, pattern...)
        if err != nil {
                return Template{}, fmt.Errorf("parsing template: %w", err)
        }
        return Template{
                htmlTpl: htmlTpl,
        }, nil
}
```

The first change is the function definition. We changed the last argument - **`patterns`** to be a variadic parameter. After that we need to update the following line, as our **`patterns`** variable is now going to work like a slice inside of our function. To pass a slice into another function that accepts a variadic parameter, we need to use the **`...`** suffix as we saw earlier. That leads to the line:

```
htmlTpl, err := template.ParseFS(fs, pattern...)
```

With that our code is now ready to accept multiple patterns in the **`ParseFS`** function call. In the future we can include additional template files if we need to use them. For instance, we might create reusable components and include them in our final template.

Source Code

- Source Code - see the final source code for this lesson.
- Diff - see the changes made in this lesson.

7.3 Named Templates

Moving forward we are going to be altering our **`.gohtml`** template files, but our dynamic reloading with **`modd`** doesn't take this into account. It currently

only rebuilds our program if there is a change to a `.go` file. Let's update the `modd.conf` file to also rebuild our code if there is a change to one of our templates.

Open `modd.conf` and update it with the following contents.

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

# Rebuild when .go or .gohtml files change.
# Exclude all test files of the form *_test.go, since these don't  affect
# our web server and are handled in the `go test @dirmods` above.
**/*.go !**/*_test.go **/*.gohtml {
  prep: go build -o lenslocked .
  daemon +sigterm: ./lenslocked
}
```

Restart `modd` and now we are ready to proceed with updating some templates.

Thus far we have been writing small, simple templates with only a handful of HTML lines per file. As our pages start to become more complex, we will start to find many cases where having reusable templates can benefit us. For instance, we likely don't want to write the HTML for a navigation bar on every single page. Not only would this be hard to update, it would take a while to setup initially with no real benefit since we know the navbar won't change between pages.

The `html/template` package allows us to create named templates that can be reused in other templates. For instance, let's imagine we wanted to design our website but we need some filler text to use in the design process. We could start by creating a template with filler text.

Open `home.gohtml` and add the following named template.

```
<h1>Welcome to my awesome site!</h1>

{{define "lorem-ipsum"}}
<p>
```

```
  Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor
  incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis
  nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat.
  Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore
  eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt
  in culpa qui officia deserunt mollit anim id est laborum.
</p>
{{end}}
```

The `{{define "..."}}` code starts a named template block. From this point onwards, anything in the template file until we reach `{{end}}` will be included as part of the named template. The text inside the quotation marks is the name of the template. In our named template, the name is `lorem-ipsum`.

Aside 7.4. What is lorem ipsum?

Lorem ispum is text commonly used as a placeholder in graphic, print, and other industries to help fill in visual mockups where real text wasn't available. It is often used in web design as well as a way of filling in blocks of text quickly when the real contents aren't known yet, or are dynamic.

If we were to restart our server and head to the home page, we wouldn't see any of the lorem ipsum text at this time. That is because the `{{define ...}}` block is only defining a template, not actually executing it or adding it to our final template that we render. If we wanted to render the lorem ipsum template, we would need to use the `{{template ...}}` action. We can even use it multiple times inside of a template.

Update `home.gohtml` with the following code.

```
<h1>Welcome to my awesome site!</h1>
{{template "lorem-ipsum"}}
{{template "lorem-ipsum"}}
```

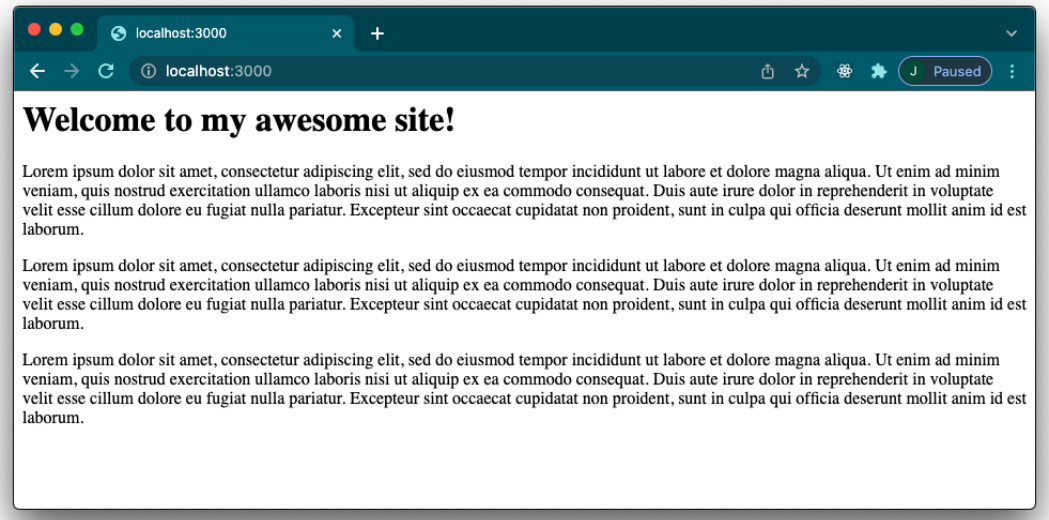
```
{{template "lorem-ipsum"}}

{{define "lorem-ipsum"}}
<p>
  Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor
  incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis
  nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat.
  Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore
  eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt
  in culpa qui officia deserunt mollit anim id est laborum.
</p>
{{end}}
```

The **`template`** keyword says we want to use a named template, and the part in quotation marks is the name of the template we want to use. We can also pass in a third argument - data we want to provide to the template - and we will see that later in the course.

It doesn't matter if the template is defined after it is used, or if it is defined in a completely different file. As long as the named template is included in the template files we parse, it will still work. We will see this later when we start breaking our named templates into separate files, but it will require some updates to our Go code so we will wait until later to make those changes.

Now that we use the **`template`** action, we will see three paragraph blocks on our home page, each with the lorem ipsum text inside of it.



This example is pretty silly, but as we progress through the course we will see how named templates can be quite powerful and helpful for developing web applications.

Source Code

- Source Code - see the final source code for this lesson.
- Diff - see the changes made in this lesson.

7.4 Dynamic FAQ Page

The best way to learn about templates is to dive into using them. In this lesson we are going to convert the FAQ template into a dynamic template that can render any questions and answers provided to it. We will also get to see how named templates make this easier by allowing us to reuse a template for a question and answer item.

The first thing we need to do is define some questions and answers. I am going to do this inside of our controllers package for now, since we don't have any data sources with questions and answers to pull from, and will instead be hard-coding them into our Go files.

Open **controllers/static.go** and add the following function:

```
func FAQ(tpl views.Template) http.HandlerFunc {
        questions := []struct {
                Question string
                Answer   string
        }{
                {
                        Question: "Is there a free version?",
                        Answer:   "Yes! We offer a free trial for 30 days on any paid plans.",
                },
                {
                        Question: "What are your support hours?",
                        Answer:   "We have support staff answering emails 24/7, though response
                },
                {
                        Question: "How do I contact support?",
                        Answer:   `Email us - <a href="mailto:support@lenslocked.com">support@l
                },
        }
        return func(w http.ResponseWriter, r *http.Request) {
                tpl.Execute(w, questions)
        }
}
```

Now we need to walk through what is happening in this code. First we have the **questions** variable. This is defined as a slice of an inline struct.

```
questions := []struct {
  Question string
  Answer   string
}{
  {
    Question: "Is there a free version?",
    Answer:   "Yes! We offer a free trial for 30 days on any paid plans.",
  },
  {
    Question: "What are your support hours?",
    Answer:   "We have support staff answering emails 24/7, though response times may be a bit
```

```
  },
  {
    Question: "How do I contact support?",
    Answer:   `Email us - <a href="mailto:support@lenslocked.com">support@lenslocked.com</a>`,
  },
}
```

While this might seem confusing at first, it is actually the same as the following code:

```
type QuestionAnswer struct {
  Question string
  Answer   string
}
questions := []QuestionAnswer{
  {
    Question: "Is there a free version?",
    Answer:   "Yes! We offer a free trial for 30 days on any paid plans.",
  },
  {
    Question: "What are your support hours?",
    Answer:   "We have support staff answering emails 24/7, though response times may be a bit
  },
  {
    Question: "How do I contact support?",
    Answer:   `Email us - <a href="mailto:support@lenslocked.com">support@lenslocked.com</a>`,
  },
}
```

Rather than defining the **`QuestionAnswer`** type, we are instead defining that struct inline. This technique is often used in table-driven testing, and in cases where we don't plan to use the type outside of the current scope, so declaring a type isn't necessary.

The rest of the lines are defining the individual questions and answer structs inside of the **`questions`** slice. The end result is a slice with three objects, and each object has a **`Question`** and an **`Answer`** field.

The last bit of code in **`FAQ()`** should look somewhat familiar. In it we are returning a function that executes the template just like our **`StaticHandler()`**

function does, but in this case we are passing the **questions** variable into the template as data to be used.

```
return func(w http.ResponseWriter, r *http.Request) {
  tpl.Execute(w, questions)
}
```

Next we want to update **main.go** to use this **FAQ** function.

```
r.Get("/faq", controllers.FAQ(
                views.Must(views.ParseFS(templates.FS, "faq.gohtml"))))
```

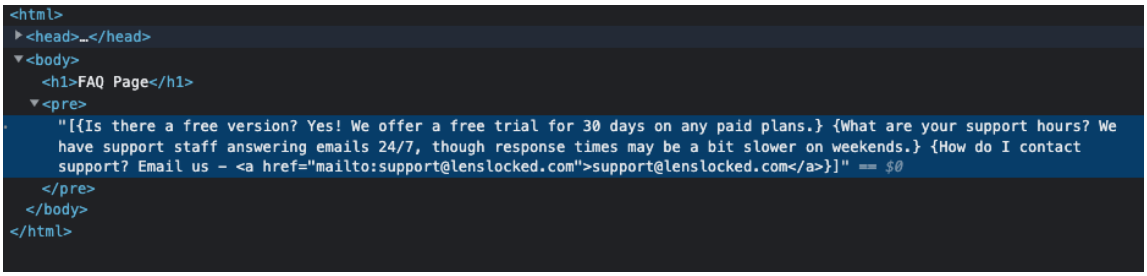
This change is pretty subtle - the **StaticHandler** part needs to be changed to instead use the **FAQ** function.

Finally, we want verify that the data we are passing into **tpl.Execute** is available inside of our template. Oftentimes when developers have bugs with a template, it is because they expected data to be passed into the template, but their code has a bug and the data never gets passed in.

Open **faq.gohtml** and update it with the following HTML:

```
<h1>FAQ Page</h1>
<pre>{{.}}</pre>
```

Finally, head to <localhost:3000/faq> and see what the results are. You should now see a **<pre>** tag in the HTML with our question and answer data inside of it.



Aside 7.5. A bug in the videos

In the videos I made a bug in the original code for **`Template.Execute`** and forgot to pass the data into the template we execute. Open **`views/template.go`** and update the **`Execute()`** method with the following code:

```
func (t Template) Execute(w http.ResponseWriter, data interface{}) {
        w.Header().Set("Content-Type", "text/html; charset=utf-8")
  // This line needs updated
        err := t.htmlTpl.Execute(w, data)
        if err != nil {
                log.Printf("executing template: %v", err)
                http.Error(w, "There was an error executing the template.", http.StatusInternalSe
        return
        }
}
```

The key change here is adding **`data`** as the last argument to **`htmlTpl.Execute`**.

When we wrote the code **`{{.}}`** in our template, we were telling our template to output *all* of the data passed into our template. We did this inside of a **`<pre>`** tag to help preserve any formatting, though in this case it likely didn't matter.

Once we have verified that our template has the data we need, we are ready to start using a named template to render each question and answer. Head back to **`faq.gohtml`** and add the following named template:

```
{{define "qa"}}
<li><b>{{.Question}}</b> {{.Answer}}</li>
{{end}}
```

The HTML used here was derived from our original questions and answers HTML. I simply copy-pasted it, then replaced the question with **`{{.Question}}`**,

and the answer with `{{.Answer}}`. This template now expects us to pass in data that has two fields - a `Question` field, and an `Answer` field. This matches the struct type we created in our Go code.

Next we need to tell our FAQ template to use the `qa` template for each question and answer we have.

```
<h1>FAQ Page</h1>
<ul>
  {{range .}}
    <p>Question: {{.Question}}</p>
    <p>Answer: {{.Answer}}</p>
  {{end}}
</ul>

{{define "qa"}}
<li><b>{{.Question}}</b> {{.Answer}}</li>
{{end}}
```

If we look at the `FAQ` function inside `static.go`, we will see that our data is a slice of structs. To iterate over a slice in a template, we need to use the `range` action along with a second argument that is the item we want to iterate over. We pass our `questions` slice in `FAQ()` directly into our tempate when we call `tpl.Execute(w, questions)`, so the slice is accessed via the period (`.`).

The `range` block also needs an `end`, so we provide that a few lines below. Now everything inside of this block will be executed for every item in our `questions` slice. That means our template will now generate three "Question: ..." lines, and three "Answer: ..." lines. We can see this if we start up our server and visit the FAQ page.

We want to use the `qa` template, so we need to update the inside of the `range` block with a call to that template. We want to pass in the entire question and answer struct, so we also provide the `.` as a final argument. This gives us the following code for our `faq.gohtml`:

```
<h1>FAQ Page</h1>
<ul>
  {{range .}}
    {{template "qa" .}}
  {{end}}
</ul>

{{define "qa"}}
<li><b>{{.Question}}</b> {{.Answer}}</li>
{{end}}
```

Now when we look at the FAQ page it is almost correct. As we learned before, the **`html/template`** package will help us avoid XSS attacks, and as a result it has rendered our **`<a>`** tag in the final answer as text, not HTML. Normally we would want this to happen, but since the answers aren't something users will be providing, we can safely bypass any escaping done by the **`html/template`** package. To do this, we need to change the type of the **`Answer`** field to instead be **`template.HTML`**. Open **`static.go`** and make the following change to **`FAQ()`**:

```
func FAQ(tpl views.Template) http.HandlerFunc {
        questions := []struct {
                Question string
                Answer   template.HTML
        }{
    // ... unchanged
  }
  // ... unchanged
}
```

Our answers happen to have HTML in them. Normally we might want to worry about unsafe HTML provided by users, but since our answers are always hand-coded by a developer, we can safely turn this into the **`template.HTML`** type.

```
func FAQ(tpl views.Template) http.HandlerFunc {
        questions := []struct {
                Question string
                Answer   template.HTML
```

```
        }{
    // ...
  }
}
```

We should now have a proper mailto link inside of our final answer on the FAQ page. If we wanted to add new questions to the FAQ page, it is as simple as adding a new item to our `questions` slice in the `FAQ()` function.

```
{
  Question: "Where is your office?",
  Answer:   "Our entire team is remote!",
},
```

Aside 7.6. Practice and experimentation

We covered a lot in this lesson, and some of it might feel confusing. This is absolutely normal, and unfortunately there isn't any easy way around it. Like all development, it takes practice and experimentation before all the details start to become clear.

The best advice I can give is to take the template we created and to try making changes to it. Change the data being passed into the template. Change the template itself to render the data in a different way. Add a new field to the data and see if you can use it in the template. Over time this will all become more natural, but until then don't be afraid to change things and see what happens.

Source Code

- Source Code - see the final source code for this lesson.
- Diff - see the changes made in this lesson.

7.5 Reusable Layouts

Most web applications have common components that need to be present on almost every page. For instance, many applications have a navigation bar at the top and a footer of some sort at the bottom of the page, and these are present no matter where you navigate to within the application.

In this lesson we are going to apply what we have learned about named templates to create a reusable layout. This won't be a very complex one, but the goal is to see how we can reuse core parts of our page, like a navbar or a footer, without writing the same HTML over and over again. At the end of this lesson our pages will each share a template that renders HTML like below.

```
<html>
  <body>
    <!-- Insert each page's HTML here -->
    <p>Copyright Jon Calhoun 2028</p>
  </body>
</html>
```

The opening html and body tags will be part of the header, and later we can add more content such as a navbar, a header tag, CSS to include, and more things we will need on every page. The copyright info and closing tags are the footer.

There are a few ways to do this. In this lesson we will explore two of them. The first is a little easier to understand, so we will use it in the course, but the second approach is what I prefer in my personal projects so I wanted to include that as well.

7.5.1 Approach #1: Header and Footer Templates

The first approach is to define a header and footer template separately, then to include them in each page we create. If someone was trying to create a layout template, this is likely the solution they would come up with on their own.

First, open **home.gohtml** and udpate it with the following code. Take note that we have removed the lorem ipsum code as well.

```
{{template "header"}}
<h1>Welcome to my awesome site!</h1>
{{template "footer"}}
```

We haven't defined a header or footer template yet, so this won't work, but you can see how if we had them defined this would look exactly like using any other named templates.

Aside 7.7. Passing data to templates

As mentioned previously, if we wanted to pass data to one of these named templates we could by using an additional argument. For instance, we could pass all the data to the header template with **{{template "header" .}}**.

We want to define the header and footer templates outside of **home.gohtml** so that we can share these named templates across multiple pages. I'll be naming the file **layout-parts.gohtml**.

```
code templates/layout-parts.gohtml
```

Then add the contents below.

```
{{define "header"}}
<html>
  <body>
{{end}}
```

```
{{define "footer"}}
    <p>Copyright Jon Calhoun 2028</p>
    <!-- NOTE: This body & html tags are not opened in this named template -->
  </body>
</html>
{{end}}
```

Finally, we need to make sure the template package knows about both files when it is parsing time. To do this, we head to **`main.go`** and add the **`layout-parts.gohtm`** file to the **`ParseFS`** function call.

```
r.Get("/", controllers.StaticHandler(views.Must(
  views.ParseFS(templates.FS, "home.gohtml", "layout-parts.gohtml"))))
```

I have rearranged the code a bit, moving **`views.Must`** *to the previous line. This was only done to keep the line length shorter to make it easier to read.*

If we restart our server and head to the home page we will see that it now returns the following HTML:

```
<html>
  <body>

<h1>Welcome to my awesome site!</h1>

    <p>Copyright Jon Calhoun 2028</p>

  </body>
</html>
```

The formatting isn't the prettiest, but that doesn't matter to a web browser. You might also see a **`<head>`** tag if you view the source in your browser's developer tools. That is added by some browsers if it is missing, and isn't something our server is producing. We will eventually add our own header though.

With this approach we are rendering each individual page's template, and that template is in charge of calling **`{{template ...}}`** for any layout templates it needs.

While this approach is straightforward and fairly easy to understand, it does have one downside - dangling HTML tags. In the header template we open the **<html>** and **<body>** tags, but we don't close them in that template. If we forget to include the footer template, we might produce incorrect HTML. We also need to make sure that the footer template stays in sync with the header template.

In most cases browsers will still render the page correctly if we forget to close a tag, and keeping the header and footer in sync is easier when the two template definitions are near one another in the source code, but it is still worth keeping in mind.

7.5.2 Approach #2: Named Page Template

The second approach we are going to look at is a bit different. Rather than rendering each page individually and having it call the header and footer templates, we are instead going to render the layout template and have it call a template named **page**.

Create a new template file with the name **layout-page.gohtml**.

```
code templates/layout-page.gohtml
```

Add the following code to the new template file. In this code we still creating the same overall layout, but we are calling a template named **page** inside of the template, and we are using a **.** to pass any data we have into that tempalte.

```
<html>
  <body>
    <!-- Note: The . passes all of our data into the "page" template so it also has access to i
    {{template "page" .}}

    <p>Copyright Jon Calhoun 2064</p>
  </body>
</html>
```

Aside 7.8. An older course version used the name "yield"

In the previous version of the course I used a template named `yield` rather than a template named `page`. This stemmed from a Ruby on Rails convention. In the course update I started using `page` as I believe this was clearer, but you can use whatever template name you want as long as it is defined for each page.

If we tried to execute this template now, it would fail. We haven't defined a template with the name `page`. To make this approach work, each page (home page, contact page, etc) needs to define a template named `page`.

Rather than overwriting our existing home template, we are going to create a new one so that both of these approaches are checked in at the end of the lesson. Create a new file named `home-page.gohtml`.

```
code templates/home-page.gohtml
```

Aside 7.9. The videos and written content differ a bit

The video content and the writeup for this lesson teach the same material, but in the videos I do not create a new file named `home-page.gohtml` and instead overwrite `home.gohtml`. We end up deleting or changing all of these files in the next lesson to use a Tailwind CSS layout, so don't worry about which route you take.

Next we need to define a `page` template inside the new `home-page.gohtml` template file.

```
{{define "page"}}
<h1>Welcome to my awesome site!</h1>
{{end}}
```

Finally, we need to update **main.go** to use the new template files.

```
r.Get("/", controllers.StaticHandler(views.Must(
                views.ParseFS(templates.FS, "layout-page.gohtml", "home-page.gohtml"))))
```

When executing a template, the very first file parsed will be the default template called when we call the **Execute** method on the template. As a result, we need to pass the **layout-page.gohtml** template into **ParseFS** first, as we want this template to render and then call the **page** template. There are other ways to handle this, like using ExecuteTemplate and provided the name of the template you want to start with, but for now listing **layout-page.gohtml** as the first template is the simplest solution.

If we wanted to add new pages using this approach we would need to make sure each defines a **page** template, and then parse the template files in a similar manner to our **home-page** template. Create a file named **contact-page.gohtml** so we can see this in action.

```
code templates/contact-page.gohtml
```

Add the following code to the new template file.

```
{{define "page"}}
<h1>Contact Page</h1>
<p>
  To get in touch, email me at
  <a href="mailto:jon@calhoun.io">jon@calhoun.io</a>.
</p>
{{end}}
```

Finally, head to **main.go** and update the files being parsed for the contact page.

```
r.Get("/contact", controllers.StaticHandler(views.Must(
                views.ParseFS(templates.FS, "layout-page.gohtml", "contact-page.gohtml"))))
```

Now when we view the page source for the contact page we should see the following HTML. *The spacing and formatting may vary a bit.*

```
<html>
  <body>
    <h1>Contact Page</h1>
    <p>
      To get in touch, email me at
      <a href="mailto:jon@calhoun.io">jon@calhoun.io</a>.
    </p>
    <p>Copyright Jon Calhoun 2028</p>
  </body>
</html>
```

What approach should you use?

Both approaches work in practice, so it is a matter of preference. I tend to use the **page** approach because I prefer to have all my HTML tags closed in a single template, but I feel the first approach is easier for beginners to grasp so I will be using the first approach in the course. For now the source files for both approaches will get committed to our repo, then in the next lesson we will clean them up and proceed with the first approach.

Source Code

- Source Code - see the final source code for this lesson.
- Diff - see the changes made in this lesson.

7.6 Tailwind CSS

At this point in the course we are going to use Tailwind CSS v2 along with a CDN. Using a CDN isn't optimal for Tailwind as it limits some of what we can do, but since we are just getting familiar with Tailwind in development, it is a great choice.

Later in the course we will update our application to use Tailwind CSS v3 to explore what the upgrade process looks like, as well as how to setup our environment to build and minimize the CSS files for deployment. For now, I recommend following along with Tailwind v2 which can be found at https://v2.tailwindcss.com/.

First we are going to delete some layout files we are no longer going to be using.

```
rm templates/layout-parts.gohtml
rm templates/*-page.gohtml
```

This should delete the following files:

```
templates/contact-page.gohtml
templates/home-page.gohtml
templates/layout-page.gohtml
templates/layout-parts.gohtml
```

Next we are going to create a new layout file named **`tailwind.gohtml`**.

```
code templates/tailwind.gohtml
```

Inside of this file we are going to copy/paste the Tailwind CSS starter template.

```
<!doctype html>
<html>
<head>
  <meta charset="UTF-8" />
  <meta name="viewport" content="width=device-width, initial-scale=1.0" />
  <link href="/path/to/tailwind.css" rel="stylesheet">
  <!-- ... -->
</head>
<body>
  <!-- ... -->
</body>
</html>
```

Next we need to break this starter template up into two parts:

- A header template
- A footer template

We do this with the **`define`** action in our template files.

```
{{define "header"}}
<!doctype html>
<html>
<head>
  <meta charset="UTF-8" />
  <meta name="viewport" content="width=device-width, initial-scale=1.0" />
  <link href="/path/to/tailwind.css" rel="stylesheet">
</head>
<body>
{{end}}

<!-- Each page's content goes here. -->

{{define "footer"}}
</body>
</html>
{{end}}
```

Our HTML currently expects a CSS file at **`/path/to/tailwind.css`** which isn't correct. We need to replace that line with a link to a CDN. We can find this code at the Using Tailwind via CDN docs.

```
<link href="https://unpkg.com/tailwindcss@^2/dist/tailwind.min.css"
  rel="stylesheet">
```

Replace the **<link>** tag in the **tailwind.gohtml** template with the CDN version.

Next we need to update all of our views to use this layout. We will start with **home.gohtml** which should be updated with the following code.

```
{{template "header" .}}
<h1>Welcome to my awesome site!</h1>
{{template "footer" .}}
```

Note that we also added a period as the last argument for both **template** actions. This passes in any data we have to the header and footer templates, and while we don't use this now, it will be useful in the future when we start creating a navigation bar that shows information about the current user - like an avatar image derived from their email address.

Next we update **contact.gohtml** and add the header and footer template calls.

```
{{template "header" .}}
<h1>Contact Page</h1>
<p>
  To get in touch, email me at
  <a href="mailto:jon@calhoun.io">jon@calhoun.io</a>.
</p>
{{template "footer" .}}
```

The last template we need to update is **faq.gohtml**. This one is a little different since I am opting to put the footer template before we define the **qa** template. This is a preference of mine - I like to finish all the page's HTML before starting to define any named templates specific to the page.

```
{{template "header" .}}
<h1>FAQ Page</h1>
<ul>
  {{range .}}
    {{template "qa" .}}
  {{end}}
</ul>
{{template "footer" .}}

{{define "qa"}}
<li><b>{{.Question}}</b> {{.Answer}}</li>
{{end}}
```

Finally, we need to update **`main.go`** to use the new **`tailwind.gohtml`** template.

```
r.Get("/", controllers.StaticHandler(views.Must(
  views.ParseFS(templates.FS, "home.gohtml", "tailwind.gohtml"))))
r.Get("/contact", controllers.StaticHandler(views.Must(
  views.ParseFS(templates.FS, "contact.gohtml", "tailwind.gohtml"))))
r.Get("/faq", controllers.FAQ(
  views.Must(views.ParseFS(templates.FS, "faq.gohtml", "tailwind.gohtml"))))
```

Aside 7.10. Line breaks in your code

Feel free to add line breaks to your code as you see fit. **`go fmt`** might modify what you do a bit, but there are multiple ways to break up the lines depending on what you feel is easier to read. For instance, the following two lines both do the same thing despite looking different.

```
r.Get("/", controllers.StaticHandler(views.Must(
  views.ParseFS(templates.FS, "home.gohtml", "tailwind.gohtml"))))

// is equal to...

r.Get("/", controllers.StaticHandler(views.Must(views.ParseFS(
  templates.FS,
  "home.gohtml", "tailwind.gohtml",
))))
```

The second option needs a comma after `"tailwind.gohtml"`, as Go requires all arguments to end with a closing tag (eg `)` or `}`) or a comma, but both are functionally equivalent. You can see some additional examples of breaking up lines if you are interested.

In the course I tend to limit lines of code to 80 characters, as this works better with eBooks and is a personal preference since I often have multiple files open side-by-side for editing, but beyond that it is personal preference as to when or where to break up lines of code. Sometimes it may even make more sense to use multiple statements with variables, but I opt not to do that here because I find the end result harder to read.

If we were to restart our server and view any of our pages we would see that they all look a bit more bland. What happened to the header text? Why aren't our links a different color or underlined?

Tailwind CSS strips out most default styling so that we start with a clean slate. In the end, this makes it easier to style everything because we don't have to worry about what pre-existing styles might be present, but on a page like ours where we aren't yet using any Tailwind CSS classes, it leaves us with a pretty bland looking page. We will address that in a future lesson.

Source Code

- Source Code - see the final source code for this lesson.

- Diff - see the changes made in this lesson.

7.7 Utility-first CSS

CSS is used to define how various HTML components on a website should look. For instance, we might write some CSS to set the background color of the entire website:

```
body {
  background-color: #ff00ee;
}
```

We could add this into a CSS file and include it in our HTML page, or we could add it directly in the HTML inside of a **<style>** tag. Open **tailwind.gohtml** and add the following inside of the **<head>** tag.

```
<style>
  body {
    background-color: #ff00ee;
  }
</style>
```

Our page now has a pink background, because **#ff00ee** is a hex value that defines a shade of pink.

When defining styles, we will frequently reuse specific styles. For instance, we might make all default buttons blue, or we might want all our buttons to have the same rounded corner radius. To support this, CSS has classes that can be used to define a set of styles, then we can apply those styles to HTML components.

Update the **<style>** section in your **tailwind.gohtml** file with the following:

```
body {
  background-color: #ff00ee;
}

.button {
  background-color: #0000ff;
  color: #ffffff;
  border-radius: 4px;
  padding: 0.5rem;
  margin: 1rem;
}
```

Now add an HTML element with a **button** class in **home.gohtml**.

```
{{template "header" .}}
<h1>Welcome to my awesome site!</h1>
<a href="#" class="button">This is my button</a>
{{template "footer" .}}
```

You may not understand what all the CSS is doing, but you should see how adding the **button** class to the **<a>** tag causes it to start looking a little more like a button.

We won't be using any of this code, so you can remove it all. Delete the entire ***<style>*** *tag inside of* ***tailwind.gohtml****, and remove the* ***<a>*** *tag from* ***home.gohtml****.*

In CSS we define classes that can be added to HTML to change how it looks. In the sample above, the class name was **button** and we signal this in CSS with a period before the class name. CSS can also target specific elements, like **h1** tags. This is what we saw with the **body { ... }** CSS.

CSS frameworks are tools that provide developers with a set of premade CSS styles that can be used design a website much faster than writing everything from scratch. They also tend to help with creating a consistent style across all pages, and can also help prevent issues where a styles differ between browsers.

Aside 7.11. Browser-specific CSS

Historically, browsers like Internet Explorer were notoriously hard to style pages for, and CSS would often need code specific for each browser to make a page look the same on every browser. This has improved over time, but there can still be differences as each browser interprets CSS in their own way. Frameworks can often take this into account for us, making it easier to style pages without needing to learn the quirks of every browser.

There are two common patterns that are used by CSS frameworks:

1. Classes define a component, like a button, and contain all of the styles for that component.
2. Classes define utility styles, and multiple classes can be combined to design each element.

Bootstrap CSS is an example of the first approach, and we can see an example of this in their button docs.

```
<!-- Creates a blue button -->
<button type="button" class="btn btn-primary">Primary</button>
<!-- Creates a gray button -->
<button type="button" class="btn btn-secondary">Secondary</button>
```

Tailwind is a utility-first CSS framework that uses the second approach. What this means is that instead of providing developers with CSS classes that define a component on the page (like a **`button`** class), it instead provides developers with classes that each apply a specific style and can be combined to design components. Below is an example of a button using Tailwind CSS classes.

```
<a class="px-4 py-2 bg-indigo-600 hover:bg-indigo-800 text-white rounded"
  href="/signup">Sign up</a>
```

It is pretty clear that the Tailwind approach takes more code to work, but it also provides some benefits:

- We have complete control over each button's look and feel.
- Changing how one button looks won't affect other buttons. With Bootstrap changing the `button` class will affect every button on the website, and this can sometimes lead to issues where designing one page leads to other pages no longer looking correct.
- Designing a custom looking app in HTML becomes much easier.
- It is easier to jump into older projects because developers only need to remember Tailwind classes, not component-specific class names.

Aside 7.12. Use the CSS framework you prefer

I don't want to turn this into a Tailwind sales pitch, but personally I have found that for my own projects, Tailwind is significantly easier for me to build and maintain with. Others hate Tailwind's approach to CSS, and prefer something like Bootstrap. You are welcome to use whatever you prefer, but we will be using Tailwind in the course and I encourage everyone to give it a try.

Let's see what Tailwind looks like in our own code. Open `home.gohtml` and update it with the following HTML.

```
{{template "header" .}}
<div class="px-6">
  <h1 class="py-4 text-4xl semibold tracking-tight">
    Welcome to my awesome site!
  </h1>
</div>
{{template "footer" .}}
```

Now when we reload our home page the welcome message is larger, has some spacing, and looks a bit better.

Diving into the code we changed, the first thing we have is the **`<div>`** tag with the **`px-6`** class. The div is simply a container that has HTML elements inside of it, and the **`px-6`** class is a variant of the Tailwind padding classes that adds some padding on the left and right sides of the **`<div>`**.

Inside our div we have our original **`<h1>`**, but we added a few classes. The first, **`py-4`** is similar to **`px-6`**, except it adds padding on the y axis (above and below the **`h1`** tag), and because we used the number **`4`** this adds a little less padding than a **`6`** would. You can see this by changing the value from 4 to 6 and reloading the page.

Aside 7.13. Tailwind Sizes

Tailwind doesn't provide classes for every possible size number you can type in. For instance, **`px-15`** won't do anything, because Tailwind doesn't provide that class. Rather than providing every size, Tailwind provides set that will suffice for most designers.

If you are ever unsure about a specific size, I suggest referring to the Tailwind Padding Docs and seeing if that value is present. Here you can also refer to the exact CSS used and the exact unit size of each padding variant.

The next class in our **h1** is the **text-4xl**. This is a class provided by Tailwind to set the font size. Additional classes can be found in the font sizes docs.

semibold is a font weight. In our case it make the text thicker, but font weights can also make fonts thinner.

tracking-tight is the last class, and it is letter spacing class. It defines how close each letter should be to the letters around it.

Each class in Tailwind slightly alters how an HTML component looks rather than having a single class like **header** that has all of these styles. In a way, using Tailwind feels very similar to writing styles directly into the HTML. One key difference is that Tailwind pulls out very common styling and makes it easier to apply them consistently across your pages.

At first the utility-first approach seems tedious. We need to repeat things a lot. For instance, if we wanted to use a similar look on our FAQ page, we would need to copy over the same HTML! Open **faq.gohtml** and update it with the following code.

```
{{template "header" .}}
<div class="px-6">
  <h1 class="py-4 text-4xl semibold tracking-tight">FAQ Page</h1>
  <ul>
    {{range .}}
      {{template "qa" .}}
    {{end}}
  </ul>
</div>
{{template "footer" .}}

{{define "qa"}}
<li><b>{{.Question}}</b> {{.Answer}}</li>
{{end}}
```

Even adding bullet points before each list item and indenting them now requires adding additional classes to our HTML! Update the **<ul>** tag with the following classes which will add the bullet point disc, and will indent each list item.

```
<ul class="list-disc list-inside">
```

In reality, this isn't as much of an issue as it might appear to be at first. Most template libraries offer ways to isolate a component and reuse the code, so we won't find ourselves copy/pasting the same HTML for every component and will instead define a component in a template, then reuse that template for the component.

Let's try this out with our `qa` template. Update `faq.gohtml` with the following HTML.

```
{{template "header" .}}
<div class="px-6">
  <h1 class="py-4 text-4xl semibold tracking-tight">FAQ Page</h1>
  <ul class="grid grid-cols-2 gap-16">
    {{range .}} {{template "qa" .}} {{end}}
  </ul>
</div>
{{template "footer" .}} {{define "qa"}}
<li class="my-4 border-t border-indigo-400 py-1 px-2">
  <span class="block text-lg text-gray-800 semibold">{{.Question}}</span>
  <span class="block text-sm text-gray-500">{{.Answer}}</span>
</li>
{{end}}
```

Aside 7.14. Tailwind breakdown

Before moving on, here is a quick rundown of what each class is doing:

- `grid` says that we want to use a grid to organize the items inside the `<ul>`.
- `grid-cols-2` defines how many columns we want our grid to have.
- `gap-16` defines the spacing between each grid item.

- `border-t` says that we want a border on the top of the `<li>` element.
- `border-indigo-400` sets the color of the border. Since we used `border-t` only the top border will be shown and it will have the indigo color.
- `block` is a display variant. In this case we are saying that we want the question and the answer to both be on a their own line.
- `text-gray-500` (and `800`) are setting the font color to various shades of gray.

Learning Tailwind is going to involve a bit of experimenting. I recommend reading the docs and tweaking some of the values here to see how each alters the look of the page.

Our FAQ page now looks a great deal better. The design may not be perfect, but we were able to quickly spruce things up and give it a nicer look and feel. More importantly, we didn't need to copy/paste the HTML and CSS for each question and answer because we used templates to do the heavy lifting for us!

Templates can be used at whatever level you prefer within your application, so if we wanted to we could create a reusable `h1` template so we didn't have to retype the classes in the `h1` tags. Personally, I find this a bit cumbersome in Go templates, but in something like React it is a bit easier so you might see it more frequently. The key here is that if you ever feel like you are rewriting a component all the time and Tailwind is making it painful, a named template might be the solution.

We have one page left to update - the contact page. Let's just get it looking decent before moving on.

```
{{template "header" .}}
<div class="px-6">
  <h1 class="py-4 text-4xl semibold tracking-tight">Contact Page</h1>
  <p class="text-gray-800">
    To get in touch, email me at
    <a class="underline" href="mailto:jon@calhoun.io">jon@calhoun.io</a>.
  </p>
</div>
{{template "footer" .}}
```

7.7.1 Utility vs Component CSS

From what I have seen, people tend to fall into two camps when looking at utility-first CSS frameworks like Tailwind:

- They love it, or
- They hate it

Personally, I love Tailwind and have stopped using Bootstrap and similar frameworks altogether in new projects. Tailwind is perhaps a hair slower initially, but long term I find it easier to maintain and add new features to projects because I only need to remember the same utility classes, and I don't need to worry about changes altering other pages.

If you are on the fence, I suggest giving it a try for at least one project. It is much easier to evaluate if it is right for you if you actually use it. Many developers who felt it was awful at first glance changed their tune after using it.

In addition to the docs, Tailwind's creators have a lot of awesome resources that teach design, have prebuilt tailwind components, etc. I recommend checking them out: https://tailwindcss.com/resources

I don't want this to be an infomercial, but I own both Tailwind UI components and Refactoring UI and recommend both. I get no commission from you buying either, I just find them to be useful resources when trying to get better at design and frontend work.

Sadly, I cannot use the Tailwind UI components in this course because you need a license to use it, and many of you won't have one, but I did want to point it out as an option for prebuilt components.

Aside 7.15. Learning Tailwind CSS further

We have discussed most CSS classes used in the code so far, but as we progress through the course I likely won't explain every CSS class in detail. I suggest referencing the Tailwind Docs if you want to learn more about the classes available.

Source Code

- Source Code - see the final source code for this lesson.
- Diff - see the changes made in this lesson.

7.8 Adding a Navigation Bar

In this lesson we are going to add a navigation bar to our application. We will be adding it inside of the existing `header` template, but if you prefer to not include the navbar on some pages you could define it as its own template.

Our first version will be far from perfect, but I find it helps to get some HTML on the page to begin with. Open `tailwind.gohtml` and update the `header` template with the following HTML.

```
{{define "header"}}
<!doctype html>
<html>
<head>
  <meta charset="UTF-8" />
  <meta name="viewport" content="width=device-width, initial-scale=1.0" />
  <link href="https://unpkg.com/tailwindcss@^2/dist/tailwind.min.css"
    rel="stylesheet">
</head>
<body>
  <header>
    <nav>
      <div>Lenslocked</div>
      <div>
        <a href="/">Home</a>
        <a href="/contact">Contact</a>
        <a href="/faq">FAQ</a>
      </div>
      <div>
        <a href="#">Sign in</a>
        <a href="#">Sign up</a>
      </div>
    </nav>
  </header>
{{end}}
```

The navbar looks pretty bad, but it has the core components you will find on most navigation bars - a logo or company name, some links to various pages, and links to sign in or sign up. The sign up and sign in links don't work, but the others should. Let's style the navbar a bit using Tailwind so that it starts to look like what we want.

Aside 7.16. Learning Tailwind

Teaching something like design, HTML, CSS, and Tailwind would take countless hours and wouldn't fit in the scope of this course, so you may need to experiment a bit with Tailwind on your own to learn it. Luckily, Tailwind simplifies this a bit and isn't something you need to learn all at once. Instead, I find it works better to just use Tailwind when designing pages and gradually learn what you need for each particular page or component as you go.

We can start by setting the color of the navbar, as well as the color of the text inside of it. We could use a plain background color with something like **bg-blue-800**, but instead I want to try out the gradients provided by Tailwind.

```
<header class="bg-gradient-to-r from-blue-800 to-indigo-800 text-white">
```

The classes we added will create a background gradient that goes from **blue-800** to **indigo-800**. More info about colors provided by Tailwind can be found at the colors documentation. We are also declaring that text inside the header should be white.

Next we want to add some padding and get all of our content on the same line. We will do this on the **<nav>** tag so that the background color spans the entire page, and our padded content is inside of colors background. We can add padding with **px-** and **py-** classes, trying out various numbers until we find something we like. We can also use **flex** to define that we want to use the CSS flex property, and **items-center** works with flex to align the items vertically centered. Finally, **space-x-12** will add horizontal spacing between each element.

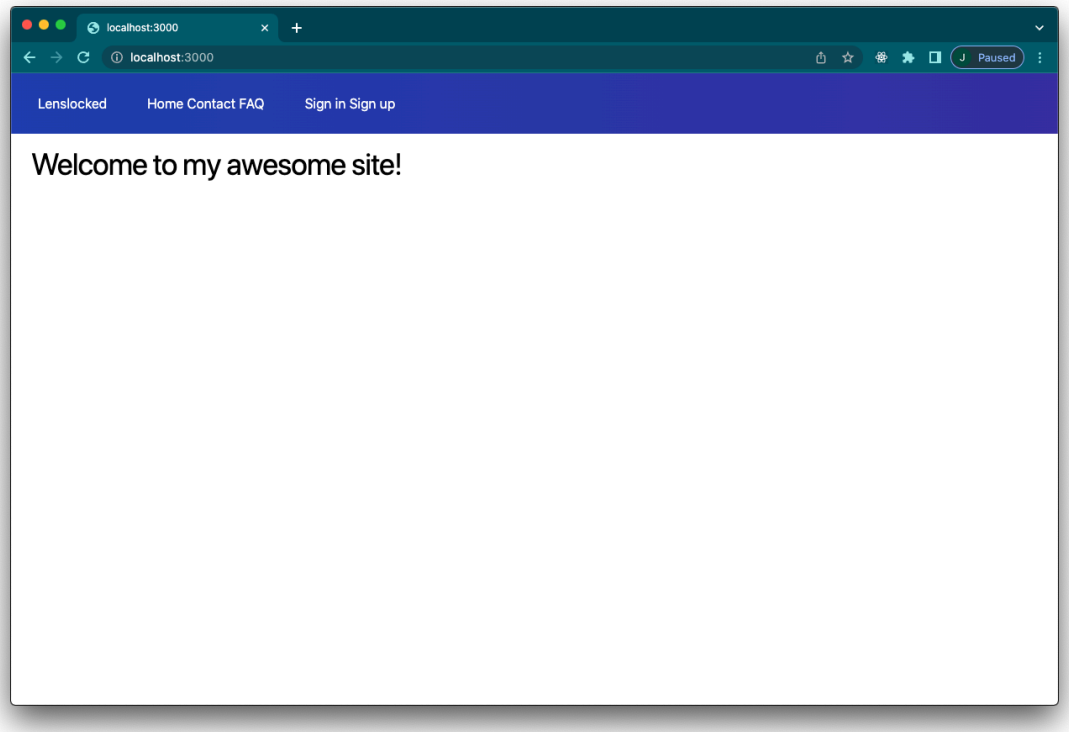
```
<nav class="px-8 py-6 flex items-center space-x-12">
```

Aside 7.17. Different HTML than the videos

In the videos the spacing between items is done by adding **pr-12** (padding right 12) to individual divs in the row. Either approach is fine, but after coming back to this material for the eBook I opted to update the HTML to use **space-x-12**

which is newer to Tailwind, but is simpler to use. The differences shouldn't matter moving forward, as both achieve the same look for our navbar, but it is worth noting in case you see a difference in your code.

Our navbar components are now on a single row with some spacing between each section.



Next we may want to make our company name stand out a bit. One way to do this is to use another font style. Another is to increase the size of the company name.

```
<div class="text-4xl font-serif">Lenslocked</div>
```

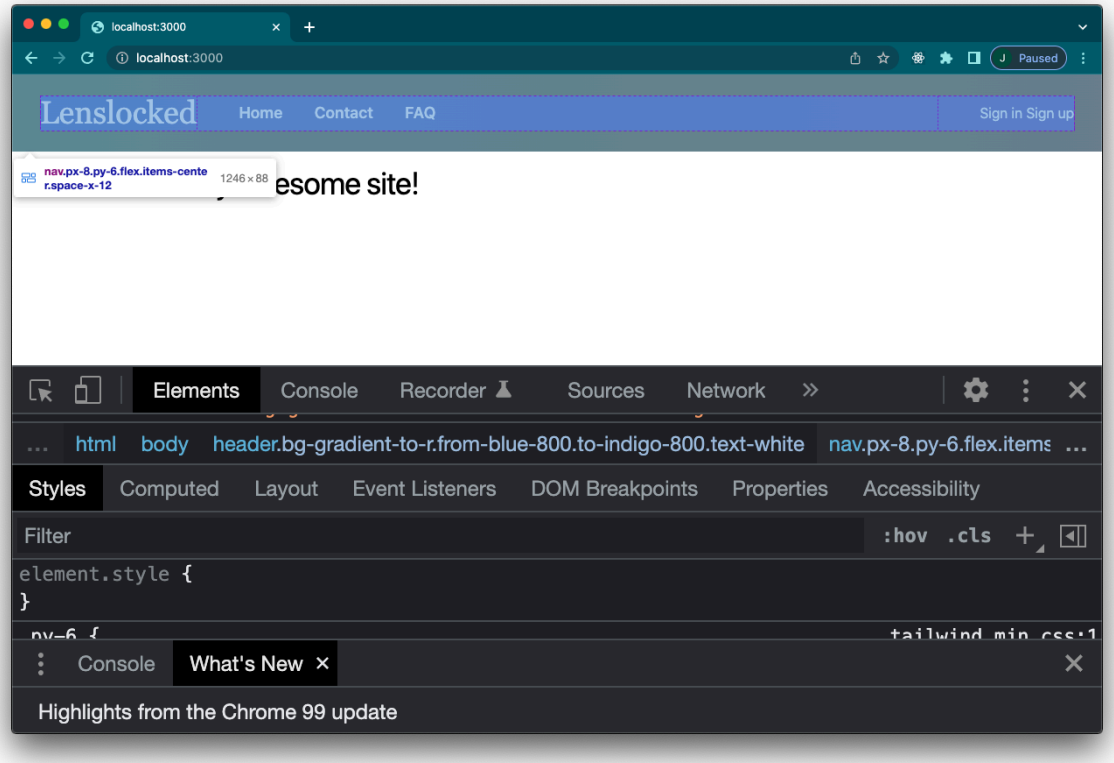
It is common to move the sign up and sign in links to the top right corner of the navbar. One way to achieve this with Tailwind is to tell our links section to grow to expand any unused space with the **`flex-grow`** class. This only works for direct children of an HTML element with the **`flex`** class which our **`<div>`** wrapper around our links is. Using **`flex-grow`** will also move the sign up and sign in buttons to the far right side as a byproduct of expanding to use unused space.

We can also add a bit of styling to each link, including a color change when the user hovers their mouse over the text. I typically opt to make changes like this using multi-select in VS Code, allowing me to write the CSS once while it is written to each line.

```
<div class="flex-grow">
  <a class="text-lg font-semibold hover:text-blue-100 pr-8" href="/">
    Home
  </a>
  <a class="text-lg font-semibold hover:text-blue-100 pr-8" href="/contact">
    Contact
  </a>
  <a class="text-lg font-semibold hover:text-blue-100 pr-8" href="/faq">
    FAQ
  </a>
</div>
```

Aside 7.18. Tip: Use your browser's inspection tools to see spacing

If you open up your browsers inspection tools, you can likely see the spacing between items to see exactly which HTML elements are taking up what space. Doing this for our navbar ar this point will show us that the links are taking up most of the horizontal space as we expected.



We can also see that Tailwind's **space-x-12** appears to be adding a margin to the left of every element but the first one to achieve the spacing we desired.

Finally, we have our sign up and sign in links. We want to style the sign up link to look like a button, and once again we will use **space-x-** to add spacing between each item. We also need to add a **hover:** class to change the button color when the user hovers over it, giving more indication that it is indeed a button.

```
<div class="space-x-4">
  <a href="#">Sign in</a>
  <a href="#" class="px-4 py-2 bg-blue-700 hover:bg-blue-600 rounded">
```

```
    Sign up
  </a>
</div>
```

Putting this all together the final HTML should match the code below.

```
<header class="bg-gradient-to-r from-blue-800 to-indigo-800 text-white">
  <nav class="px-8 py-6 flex items-center space-x-12">
    <div class="text-4xl font-serif">Lenslocked</div>
    <div class="flex-grow">
      <a class="text-lg font-semibold hover:text-blue-100 pr-8" href="/">
        Home
      </a>
      <a class="text-lg font-semibold hover:text-blue-100 pr-8" href="/contact">
        Contact
      </a>
      <a class="text-lg font-semibold hover:text-blue-100 pr-8" href="/faq">
        FAQ
      </a>
    </div>
    <div class="space-x-4">
      <a href="#">Sign in</a>
      <a href="#" class="px-4 py-2 bg-blue-700 hover:bg-blue-600 rounded">
        Sign up
      </a>
    </div>
  </nav>
</header>
```

Feel free to alter this navbar styling as a way to explore Tailwind a bit further. Designing pages is a great way to learn Tailwind's utility classes.

Source Code

- Source Code - see the final source code for this lesson.
- Diff - see the changes made in this lesson.

7.9 Exercises

The following exercises don't have solutions, but are worth dedicating some time towards.

7.9.1 Ex1 - Experiment with Tailwind

You may decide it isn't for you, but take some time to try out Tailwind and design a few pages or components using it.

7.9.2 Ex1 - Create a Footer

Many websites include a footer with information like the copyright, links to various pages within the site, links to related communities, and more. Check out a few of your favorite sites, look for their footer, and try to create something similar for your app. This will give you practice with both templates and using Tailwind.

7.9.3 Ex3 - Explore Go's embed package

Read the embed docs and try to think about other ways you might be able to use the package.

Source Code

- Source Code - see the final source code for this lesson.
- Diff - see the changes made in this lesson.

Chapter 8

The Signup Page

8.1 Creating the Signup Page

Creating a web application can feel daunting at first. Even a task as simple sounding as user accounts can turn into a large, overwhelming project. I find it helps to break these large undertakings down into smaller, more manageable tasks before proceeding.

In this section we are going to focus on one smaller aspect of our overall user management system - the sign up page. We will not concern ourselves with how to securely store passwords, databases, or anything else; our goal is simply to create a page that shows a sign up form, and then to write code on our server to parse the information from that form when a user submits it. We will focus on processing and using that information in a future section.

With most new pages I find it helps to start with creating a template and rendering it. Let's create the signup template.

```
code templates/signup.gohtml
```

Inside of this template add an HTML form so that users can provide us with information.

```
{{template "header" .}}
<form>
  <div>
    <label>Email Address</label>
    <input />
  </div>
  <div>
    <label>Password</label>
    <input />
  </div>
  <div>
    <button>Sign up</button>
  </div>
</form>
{{template "footer" .}}
```

The form won't look pretty or work, but it should suffice as a starting point. Next we need to render the page. For now we can do this using the **`StaticHandler`** function in our **`controllers`** package. Open **`main.go`** and add the following code.

```
r.Get("/signup", controllers.StaticHandler(
        views.Must(views.ParseFS(templates.FS, "signup.gohtml", "tailwind.gohtml"))))
```

Now that we have a signup page, we can update our navbar to link to it. Open **`tailwind.gohtml`** and update the **`href`** on the sign up button link.

```
<a href="/signup" class="px-4 py-2 bg-blue-700 hover:bg-blue-600 rounded">
  Sign up
</a>
```

Restart your web server and click the "Sign up" button in the navbar (top right). It should now take you to a sign up page. The form doesn't appear to have any inputs, but if you inspect the page source you will see that there are inputs

present, they are just unstyled due to Tailwind stripping out all default styles. If you would like to see what the form looks like without the Tailwind effect, remove the `{{template "header" .}}` line from `signup.gohtml` and refresh the page.

We will take care of styling the form in the next lesson. Our goal for the rest of this lesson is to make the form work. That is, we want it to submit the data from the form to our Go server when we click the "Sign up" button. Right now when we click the sign up button it doesn't appear to do anything useful.

We tell the browser what to do when we click a form's submit button with two attributes on the `form` HTML tag.

1. The `action` attribute
2. The `method` attribute

Open `signup.gohtml` and update the `<form>` tag with the attributes below.

```
<form action="/signup" method="post">
```

Aside 8.1. A Difference Between the Book and Video

In the written version of the course the path `/signup` is used to process the signup form, while in the videos the path `/users` is used as this form creates a new user. Both are valid approaches, but keep in mind that there is a slight difference between the two version of the course.

The `action` attribute in this code is telling our browser that when a user clicks the submit button for the form, it should send the information to the web server

using the path `/signup`. The `method` attribute tells our browser what HTTP method to use, though most browsers only support `POST` and `GET` without some extra JavaScript.

Both of these attributes are documented here: https://developer.mozilla.org/en-US/docs/Web/HTML/Element/form

Submitting the form now results in a `405` status code response. This stands for `Method Not Allowed`, which is a fancy way of saying we tried to use an HTTP method (`POST` in this case) that our web server doesn't currently support. We will fix this in a future lesson, but for now know that seeing a 405 error when you submit the form is expected.

When we submit our form, the data in the `<input>` fields will be submitted to the web server, btu we need to add some attributes to these inputs to help our server out. We will be using the following HTML attributes:

- `name` - Specifies the key used for the value when the form is submitted.
- `id` - Any html element can have this. In our case we will use it to link the `<label>` html to the `<input>` it is a label for, which can improve accessibility.
- `type` - The type of the input. We will use the `email` and `password` types.
- `placeholder` - Specifies text to show inside the input when it is empty.
- `required` - Tells the browser if an input is required to submit the form. It is possible for users to bypass this check if they edit the HTML, so it is always important to still validate data on your server as well!
- `autocomplete` - Helps browsers fill in autocompletion data more accurately.

Below is the updated HTML using these attributes. Update the form in `signup.gohtml`.

```
<div>
  <label for="email">Email Address</label>
  <input
    name="email"
    id="email"
    type="email"
    placeholder="Email address"
    required
    autocomplete="email"
  />
</div>
<div>
  <label for="password">Password</label>
  <input
    name="password"
    id="password"
    type="password"
    placeholder="Password"
    required
  />
</div>
```

We also added a **`for`** attribute to the two labels, and the value of this attribute is going to match the **`id`** attribute of the input the label relates to. In this particular example we have matching **`name`**, **`id`** and **`type`** attributes, but in many cases those three may have different values and the **`for`** value should match the **`id`** value.

Aside 8.2. HTML attributes

More information for all of the attributes we are using can be found on the MDN input docs and the MDN label docs. Some individual attributes, like autocomplete, have more thorough docs of their own.

`name` is probably the only attribute we are using for our input tags that is required, but being more explicit and adding tags you can will only improve the end user's experience.

Lastly, we need to define the type of the button. In our case we want it to submit the form, so it will have the type **`submit`**.

```
<button type="submit">Sign up</button>
```

Button docs: https://developer.mozilla.org/en-US/docs/Web/HTML/Element/button

Our form still doesn't look great, but visiting the sign up page should now show up an input box with the placeholder text. If you have entered data in local development in the past you might also see autocompleted data; deleting this will then show you the placeholder text.

Source Code

- Source Code - see the final source code for this lesson.
- Diff - see the changes made in this lesson.

8.2 Styling the Signup Page

Our sign up form has everything it needs to be functional, but it looks pretty ugly and is hard to use since it has no styling. Let's take some time to add some design to our page.

The first thing we want to do is make every page span at least the entire height of the screen, and to add a light gray background. This will make it a bit easier to call attention to something by placing it inside of an area with a white background, which we will see in a bit.

Open **`tailwind.gohtml`** and update the **`<body>`** tag.

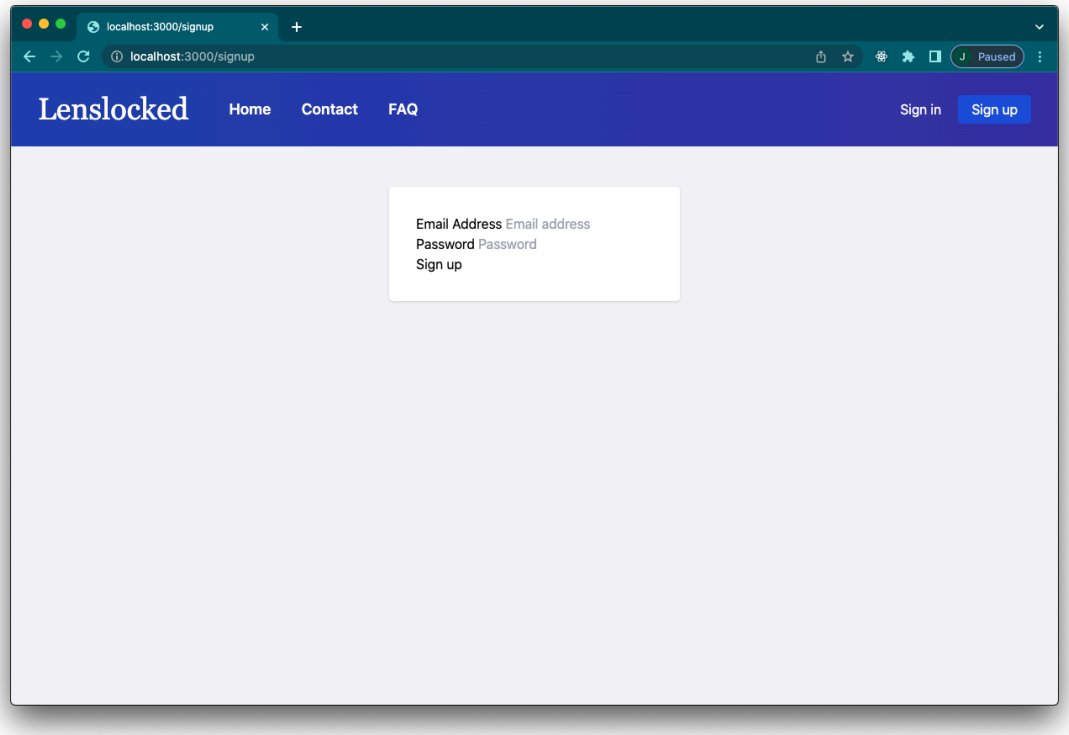
```
<body class="min-h-screen bg-gray-100">
```

Next, we are going to wrap the entire sign up form in a ** ** tag, and give that entire div a white background with some padding, rounded corners, and a slight shadow. Open **signup.gohtml** and the div code below.

```
<div class="px-8 py-8 bg-white rounded shadow">
  <form action="/signup" method="post">
    <!-- Unchanged form HTML -->
  </form>
</div>
```

Our sign up form is in a white panel, but it spans the entire width of the screen. One way to address this is to put that ** ** panel inside of another div that uses the flex to center things. We can also add some padding on the Y axis to separate the form from the navbar.

```
<div class="py-12 flex justify-center">
  <div class="px-8 py-8 bg-white rounded shadow">
    <form action="/signup" method="post">
      <!-- ... -->
    </form>
  </div>
</div>
```



Our form is starting to look a bit better, but it could use a title. We can use the **<h1>** tag to add one.

```
<div class="py-12 flex justify-center">
  <div class="px-8 py-8 bg-white rounded shadow">
    <h1 class="pt-4 pb-8 text-center text-3xl font-bold text-gray-900">
      Start sharing your photos today!
    </h1>
    <form action="/signup" method="post">
      <!-- ... -->
    </form>
  </div>
</div>
```

Now we can style the inputs to make them easier to use. Below is the HTML for the input fields. Every class used to style the inputs is documented in the Tailwind docs, but we won't cover them all here.

```
<div class="py-2">
  <label for="email" class="text-sm font-semibold text-gray-800">
    Email Address
  </label>
  <input
    name="email"
    id="email"
    type="email"
    placeholder="Email address"
    required
    autocomplete="email"
    class="w-full px-3 py-2 border border-gray-300 placeholder-gray-500
      text-gray-800 rounded"
  />
</div>
<div class="py-2">
  <label for="password" class="text-sm font-semibold text-gray-800">
    Password
  </label>
  <input
    name="password"
    id="password"
    type="password"
    placeholder="Password"
    required
    class="w-full px-3 py-2 border border-gray-300 placeholder-gray-500
      text-gray-800 rounded"
  />
</div>
```

The “Sign up” button could use some styling as well. It currently just looks like text on the page with no indication it can be clicked.

```
<div class="py-4">
  <button class="w-full py-4 px-2 bg-indigo-600 hover:bg-indigo-700
    text-white rounded font-bold text-lg">
    Sign up
  </button>
</div>
```

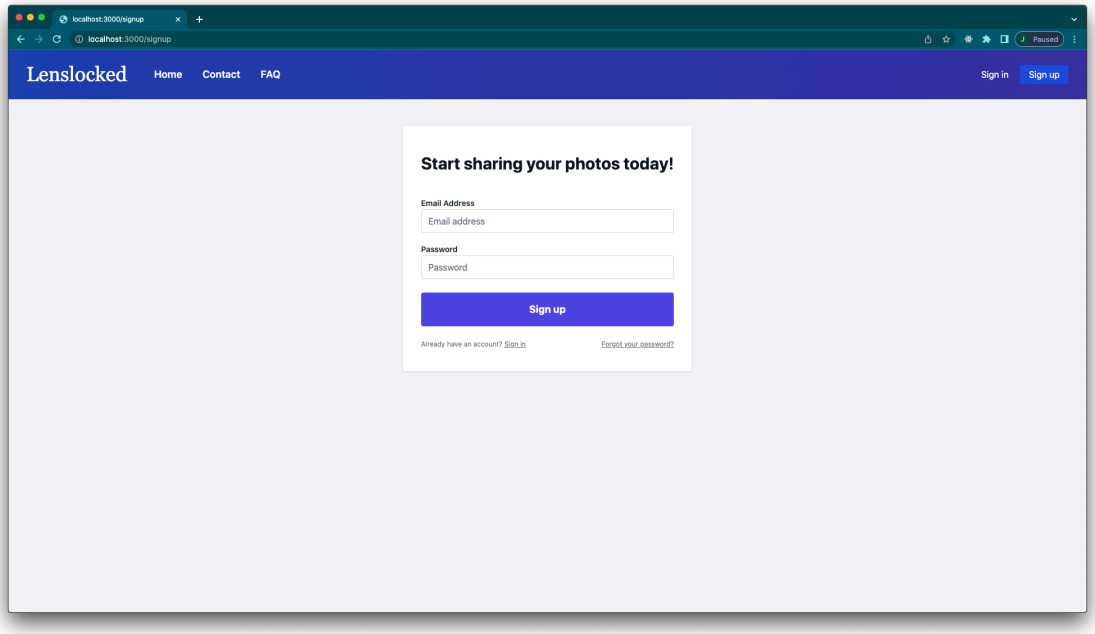
Lastly, there are a few extra links we can add to our signup form that users might be looking for. Most notably, users might remember they have an account already and want a link to the sign in page, or they might realize they

have an account and forgot their password so we can add a "Forgot your password?" link. Below is some HTML to add inside of our `<form>` tag below the submit button.

```
<div class="py-2 w-full flex justify-between">
  <p class="text-xs text-gray-500">
    Already have an account?
    <a href="/signin" class="underline">Sign in</a>
  </p>
  <p class="text-xs text-gray-500">
    <a href="/reset-pw" class="underline">Forgot your password?</a>
  </p>
</div>
```

The footer links won't work right now, but it is a good idea to include them when designing pages so they are taking into account for the overall look and feel of the page. As we continue adding functionality we will update the paths if they turn out to be incorrect.

Putting this all together we have a pretty nice looking sign up form.



The final HTML for **signup.gohtml** is shown below.

```
{{template "header" .}}
<div class="py-12 flex justify-center">
  <div class="px-8 py-8 bg-white rounded shadow">
    <h1 class="pt-4 pb-8 text-center text-3xl font-bold text-gray-900">
      Start sharing your photos today!
    </h1>
    <form action="/signup" method="post">
      <div class="py-2">
        <label for="email" class="text-sm font-semibold text-gray-800">
          Email Address
        </label>
        <input
          name="email"
          id="email"
          type="email"
          placeholder="Email address"
          required
          autocomplete="email"
          class="w-full px-3 py-2 border border-gray-300 placeholder-gray-500
            text-gray-800 rounded"
        />
      </div>
      <div class="py-2">
        <label for="password" class="text-sm font-semibold text-gray-800">
          Password
        </label>
        <input
          name="password"
          id="password"
          type="password"
          placeholder="Password"
          required
          class="w-full px-3 py-2 border border-gray-300 placeholder-gray-500
            text-gray-800 rounded"
        />
      </div>
      <div class="py-4">
        <button class="w-full py-4 px-2 bg-indigo-600 hover:bg-indigo-700
          text-white rounded font-bold text-lg">
          Sign up
        </button>
      </div>
      <div class="py-2 w-full flex justify-between">
        <p class="text-xs text-gray-500">
          Already have an account?
          <a href="/signin" class="underline">Sign in</a>
        </p>
        <p class="text-xs text-gray-500">
          <a href="/reset-pw" class="underline">Forgot your password?</a>
```

```
        </p>
      </div>
    </form>
  </div>
</div>
{{template "footer" .}}
```

Source Code

- Source Code - see the final source code for this lesson.
- Diff - see the changes made in this lesson.

8.3 Intro to REST

When developing web applications, you will frequently hear the terms "server" and "client". With web applications, client typically refers to an end user's browser, and the server is what we are building with our Go code. The client can be something other than a browser. For instance, if we used the tool Postman to communicate with our server, it would be the client.

REST stands for REpresentational State Transfer, and it is an architectural style that helps provide some standards that make it easier for clients and servers to communicate. When a serve applies this style, it is often referred to as RESTful.

One of the guiding principles of a RESTful system is that it should be stateless; every request should be self contained. Put into simpler terms, this means that every request should have all of the information required for the server to proceed, and the server shouldn't have to communicate back and forth with a user for a single action.

A stateful interaction is one you might have with a a customer service representative over the phone. You call the customer support number and the rep-

resentative asks for information to look up your account. They might then put you on hold while they look up your order. A few minutes later you explain your issue and they put you on hold while they look into the issue. Finally the representative comes back on the line and explains the issue.

If at any time during these holds you would be disconnected, you would need to start the entire process over the next time you called. The next customer service representative wouldn't have the context needed to help you.

A stateless interaction can be picked up at any point and is closer to emailing a customer support representative. Even if your internet stops working, or if an email fails to send, you can resend it and because the email contains enough context the customer support agent should be able to help you without reconfirming every detail the previous agent confirmed.

A RESTful web service can achieve this in a variety of ways, one of which is using cookies to store important information, and then the browser will provide that information along with a web request. For instance, once a user logs in we will set a cookie in their browser. On all future web requests their browser will include the cookie allowing us to verify who they are without needing to ask for their password over and over again.

Requests made to a REST web service consist of at least two things:

- An HTTP verb (GET, POST, PUT, DELETE)
- A path (`/galleres/123`)

The request may contain other information such as a request body, url query parameters, cookies, or headers, but those are not required.

I frequently refer to an HTTP verb and path pair as an **endpoint**. For instance, `GET /signup` is an endpoint our application currently supports.

In a typical REST service, the paths used in endpoints are all about resources. For instance, our application will have users, images, and galleries which will

all be resources. When we want to define paths for the gallery resource, they will likely end up looking roughly like the endpoints defined in the table below.

HTTP Method	Path	What happens
`GET`	/galleries	Read a list of galleries
`GET`	/galleries/:id	Read a single gallery
`POST`	/galleries	Create a gallery
`PUT`	/galleries/:id	Update a gallery
`DELETE`	/galleries/:id	Delete a gallery

Many resources also need endpoints to render forms. These tend to look similar to the endpoints above, but with a `GET` HTTP verb and the intended action appended to the path. Below are a couple examples.

HTTP Method	Path	What happens
`GET`	/galleries/new	Read a form for creating galleries
`GET`	/galleries/:id/edit	Read a form for editing a gallery

By shaping our endpoints around HTTP methods and resource-oriented paths, we can use an endpoint to help us determine what resource the client is interested in as well as what action they want to take. A `GET /galleries` suggests the user wants to view the galleries, while a `POST /galleries` suggests the user wants to create a new gallery.

It is worth noting that resources and paths should be used a guideline, not a hard set of rules. In some cases a custom endpoint that might not be 100% RESTful might be much clearer. For instance, when a user signs up they are technically creating a new account, so a RESTful path for the sign up page might be `GET /users/new`, but given that it is very common for websites to have their sign up page at `GET /signup` I typically opt to use that path.

REST only pertains to how a client and a server communicate. A RESTful service could organize the code in any way that the developer chooses and still be RESTful. In practice, using REST to organize an applications code can

make it easier to navigate. This is especially true with MVC, and we will be creating controllers for each resource we allow users to interact with.

It is also common to organize views (templates in our case) similarly. One way to do this is to organize the `templates` directory with subdirectories.

```
templates/
  galleries/
    show.gohtml # GET /galleries/:id
    list.gohtml # GET /galleries
    new.gohtml  # GET /galleries/new
    edit.gohtml # GET /galleries/:id/edit
```

As with everything in life, there are always exceptions. As I noted earlier, the signup page could be organized in `templates/users/new.gohtml`, or it could be stored at `templates/signup.gohtml`. Arguments could be made for both, and it is more important to be consistent than to determine which is optimal. REST should be used as a guideline, not a strict set of rules to follow.

Aside 8.3. Additional reading

Codecademy has an article titled What is Rest? that can provide some additional information. There are also plenty of articles to be found via a Google searchs, but be wary of any that try to dictate definitive rules. There is no golden standard for what is or isn't REST compliant, and I often find that websites that preach to heavily on how you must design a RESTful system can slow you down rather than help you excel.

Source Code

- Source Code - see the final source code for this lesson.
- Diff - see the changes made in this lesson.

8.4 Users Controller

In this lesson we are going to apply what we learned about REST to structure our code. Our sign up page is currently rendered via the `controllers.StaticHandler` function. If we wanted our code to apply REST a bit better, we could instead create a controller for users, and the sign up page would be rendered with a new user method on the users controller.

To implement this, we first need to create a source file for our users controller. We can name this `users.go` so it is easy to find all of the controller code related to the user resource.

```
code controllers/users.go
```

Next we need to declare the `Users` type. We will attach all of our user-related HTTP handlers to this type as methods, and any data that the handler functions need access to can be stored as a field on the struct. For now we don't have any data needs, so `Users` will be an empty struct.

```
package controllers

type Users struct{}
```

We need an HTTP handler function to contain the code related to rendering the sign up page. We could name this `Signup`, or we could name it `New` since we are creating a new user. I am going to opt to use `New`, but this is simply a personal preference.

```
func (u Users) New(w http.ResponseWriter, r *http.Request) {
  // TODO: Render the signup page
}
```

In the future we will use functions like `Edit` and `Update` to process and render the sign in form, and `Create` to process the sign up form.

As we saw in the past, if we try to parse the template into a view inside of our HTTP handler function, we also need to handle potential parsing errors. Instead, it is better to make an already parsed template available to our `New` method. We can do achieve this by updating our `Users` type to have a `Templates` field that contains the various templates our methods need as fields.

```
type Users struct {
  Templates struct {
    New views.Template
  }
}
```

With this structure we can parse the templates we need at startup, verify they parsed correctly, and then assign them to the corresponding field. We will write that code in a bit, but for now we can assume that this `New` template is available and use it inside of our HTTP handler function.

```
func (u Users) New(w http.ResponseWriter, r *http.Request) {
        u.Templates.New.Execute(w, nil)
}
```

We aren't passing any data into our `New` template at this time, but in the future we could parse URL query params and use those to pre-populate some of the signup form. When you purchased this course you received an link to sign up that did this.

Finally, we need to update `main.go` to use the new controller. Head to the `main` function and replace the `r.Get("/signup", ...)` line with the following Go code.

```
var usersC controllers.Users
usersC.Templates.New = views.Must(views.ParseFS(
  templates.FS, "signup.gohtml", "tailwind.gohtml"))
r.Get("/signup", usersC.New)
```

In the end we didn't add any new functionality, but this should make it easier to organize all of our handlers related to users moving forward.

Aside 8.4. Differences in the videos

In the videos the users controller is created with the line of code:

```
// Videos use this:
usersC := controllers.Users{}
// vs the approach here:
var usersC controllers.Users
```

This is functionally equivalent to what we do in the book, but I typically tend to use the **`var name type`** approach when I know I am starting with a zero value. This didn't happen in the video because I was demonstrating how nested anonymous structs can oftentimes be annoying to instantiate, and it is easier to start with the zero value and simply assign values like we do on the line:

```
usersC.Templates.New = ...
```

Source Code

- Source Code - see the final source code for this lesson.
- Diff - see the changes made in this lesson.

8.5 Decouple with Interfaces

Our users controller currently has a dependency on the **views** package. This isn't an issue, since our app won't ever rely on anything else to render HTML, but we could remove this coupling altogether using interfaces. In this lesson we are going to explore how to do that in Go.

The first thing we want to do is create an interface that defines the methods we need. If we look at our **controllers/users.go** source file we can see exactly which methods on the **views.Template** type that we use:

```
func (u Users) New(w http.ResponseWriter, r *http.Request) {
  // We need the `Execute` method!
        u.Templates.New.Execute(w, nil)
}
```

The only method we ever use on the **views.Template** type is the **Execute** method. Interfaces allow us tell our code, "I don't care what specific type is assigned to this variable, as long as it has this **Execute** method that I need". Let's create a **template.go** source file in the **controllers** package where we can create our first interface type.

```
code controllers/template.go
```

Add a new interface type with the name **Template** and put the **Execute** method inside of it.

```
package controllers

type Template interface {
        Execute() // What arguments does this take?
}
```

When we go to add the `Execute` method we might not know exactly what arguments it takes. We want the `views.Template` type to match our new interface, so we can look there to see what the method accepts as arguments. This is in `views/template.go`.

```
// Existing code in views/template.go
func (t Template) Execute(w http.ResponseWriter, data interface{}) {
  // ...
}
```

It looks like our `Execute` method needs to have two arguments, the first being an `http.ResponseWriter`, and the second being an `interface{}`. Updating our own template interface gives us the following code for `controllers/template.g`

```
package controllers

import "net/http"

type Template interface {
        Execute(w http.ResponseWriter, data interface{})
}
```

If inside of our `controllers` package we had code that used additional methods on our templates, we might need to add those here, but at this time the only method we use is `Execute()`.

Aside 8.5. Naming templates

It is also worth noting that the name of our `Template` interface type could be anything. For one-method interfaces, a common pattern in Go is to add an "er" on to the method name. For instance, we could have called this interface `Executer`.

```
type Executer interface {
        Execute(w http.ResponseWriter, data interface{})
}
```

In this particular case I feel that **Executer** sounds a bit morbid, and **Template** is a bit clearer on the intended use for our interface, but either approach would work.

Now that we have a **Template** interface type declared, we can use this as a field on our **controllers.Users** type. Open **controllers/users.go** and make the following changes.

```
type Users struct {
        Templates struct {
                New Template
        }
}
```

The change is subtle, but now the **New** field has the type **Template**, which is the **controllers.Template** interface. Before it was using the **views.Template** type. With this change we are telling the Go compiler that any type can be assigned to the **New** field, as long as it has all the methods defined by the **Template** interface. That means we can assign a **views.Template** to this field, or we could even assign an **html/template.Template** to this field since that type also has a valid **Execute** method.

We are also going to update the code in **static.go** to use our new **Template** interface.

```
package controllers

import (
        "html/template"
        "net/http"
)

type Static struct {
        Template Template
}

func (static Static) ServeHTTP(w http.ResponseWriter, r *http.Request) {
```

```
        static.Template.Execute(w, nil)
}

func StaticHandler(tpl Template) http.HandlerFunc {
        // ... unchanged
}

func FAQ(tpl Template) http.HandlerFunc {
        // ... unchanged
}
```

We have now completely decoupled our **controllers** package form the **views** package. We can verify that this change removed the dependency on the **views** package by looking at our imports. There aren't any source files inside the **controllers** package that import the **views** package after our updates!

```
import (
  // This import no longer exists in any of your controllers code
  "github.com/joncalhoun/lenslocked/views"
)
```

While this might not seem like a big deal at this time, cyclical dependencies can be problematic for beginners to Go. A cyclical dependency occurs when following the imports for our packages can result in a loop. The easiest example is a loop with two packagess. Imagine the following example:

```
// controllers/users.go
package controllers

import (
  // ... other imports
  "github.com/joncalhoun/lenslocked/views"
)

// ... rest of source code that uses the views package


// views/template.go
package views

import (
```

```
  // ... other imports
  "github.com/joncalhoun/lenslocked/controllers"
)

// ... rest of source code that uses the controllers package
```

If we tried to compile this code, the Go compiler would give us an error. In Go we can't have import cycles, and in the example above the views package imports the controllers package, and the controllers package imports the views package.

Our code doesn't do this, but beginners often introduce import cycles inadvertently. In many cases it can be the result of using patterns that work in other languages, but won't work in Go. For instance, in Ruby import cycles are permitted, so using a pattern you have seen in Ruby on Rails might not work in Go.

Using interfaces as we did can help avoid import cycles. Our `controllers` package can still make use of our `views.Template` implementation, but it never needs to import the `views` package removing any possibility of an import cycle.

Aside 8.6. More complex import cycles

It is possible to have import cycles that span many packages. For instance, the following would be an import cycle spread across three packages.

```
A imports B
B imports C
C imports A
```

As long as a cycle can be created, the Go compiler will produce an error.

Interfaces can also give us more freedom to make changes to our code. In `main.go` we currently use the `views.Template` type when setting up our users controller and when calling `controllers.StaticHandler()`, but in the future we could replace `views.Template` with any type that has an `Execute()` method matching our needs. We could even use the `html/template` package directly, as `template.Template` also meets our interface requirements.

It is also worth noting that code in `main.go` or the `views` package remained unchanged throughout this refactor, yet our code continues to compile and work correctly. If we were to restart our Go server every page we created will still render without any errors. One of the interesting aspects of Go interfaces is that we don't need to declare that a type implements a specific interface. As a result, it is very easy to write code that starts with a concrete type - like `views.Template` - and later is refactored to accepts an interface, just like we did with the `controllers` package.

In our particular case introducing the `controllers.Template` interface and decoupling our code wasn't 100% necessary. Our controllers and views packages won't ever have a cyclical dependency, as our views package will never import the controllers package. Still, this is a great technique for helping remove cyclical dependencies, so I felt it was worth looking at. It can also help illustrate how the logic in our controllers can be almost entirely agnostic of what data format our server is returning. For instance, we could easily create a new type with an `Execute()` method that renders JSON instead of HTML, turning our server into a JSON API.

Aside 8.7. Interfaces are an advanced topic

If you are feeling overwhelmed, keep in mind that interfaces are a fairly advanced technique that become useful later in your career. This is especialyl true when testing, or when building long-lived applications that need to evolve over time. Early on most of your code won't need to be decoupled, so adding interfaces like we did in this lesson aren't necessary. Still, you are very likely to encounter something

like this and it is better to have an idea of what is happening so you can better understand it.

Source Code

- Source Code - see the final source code for this lesson.
- Diff - see the changes made in this lesson.

8.6 Parsing the Signup Form

Our sign up page is looking pretty good, so our next goal is to parse the form when it gets submitted to our Go server. We won't be doing anything with the data just yet, but by the end of this lesson we will have access to all of the data in the form after a user submits it. That is we will be learning how to parse incoming form data. This will prepare us to learn about databases, user management, and more so we can start creating user accounts.

At this point submitting our form results in an `405` error. Let's fix that by adding a `POST /signup` endpoint. For now we can return a temporary response just to let us know it is working. Open `controllers/users.go` and add the following method.

```
func (u Users) Create(w http.ResponseWriter, r *http.Request) {
        fmt.Fprint(w, "Temporary response")
}
```

Next, open `main.go` and add a route for the new endpoint.

```
var usersC controllers.Users
usersC.Templates.New = views.Must(views.ParseFS(
  templates.FS, "signup.gohtml", "tailwind.gohtml"))
r.Get("/signup", usersC.New)
// This following line is the only new line
r.Post("/signup", usersC.Create)
```

Aside 8.8. Difference between videos and writeup

In the videos we use `r.Post("/users", ...)` both here and in our signup HTML form. Either path is fine. Technically `/users` is a bit more RESTful, but posting a signup form to the path `/signup` is an incredibly common exception to REST. I am only sticking with `/signup` here because we already used that path in our HTML form. In the videos I happened to use `/users` forgetting what I had done in the written version. Just be sure to keep your `signup.gohtml` form and this path in sync and either will work.

At this point it is a good idea to test everything to verify it is working. Restart the Go server, navigate to the sign up page in a browser, and submit the form. If you see "Temporary response" in the new page, everything is working as intended. We are now ready to learn how to parse the incoming form data.

When a form is submitted to our web application it will be included in the web request that the user is making. In Go terms, it will be part of the http.Request parameter passed into our handler function. We can see this in the Go source code.

```
type Request struct {
        // ... other fields

        // PostForm contains the parsed form data from PATCH, POST
        // or PUT body parameters.
        //
```

```
        // This field is only available after ParseForm is called.
        // The HTTP client ignores PostForm and uses Body instead.
        PostForm url.Values

// ... other fields
}
```

The `PostForm` field contains data from a POST request, but before we can access the form, we first need to tell our code to parse the request body. We do this by calling the ParseForm method on the `http.Request` type in Go.

If we head to the docs for the `ParseForm` function we can get additional information about what the function does, but the key takeaway is that when we call `ParseForm` it will parse the HTML form and store the data in the `PostForm` field.

Aside 8.9. Idempotent functions

PostForm is an idempotent function. What this means is that our code could call the `PostForm` function multiple times in a single web request, and everything would work the same as when we call `PostForm` once. While this isn't important for us now, it will prove useful in the future when we learn about Cross Site Request Forgery and how to prevent these attacks.

At this point we know we need to call `ParseForm` on our http request, and after that we can access the `PostForm` field. The final step is figuring out what format the data is in inside the `PostForm` field. Looking at the docs, we can see that `PostForm` has the type url.Values. We can then proceed to look up this type in the docs and see that it has a Get method that seems useful for retrieving values based on their key. But what is the key?

If we look back at our HTML form, the `name` attribute we used on each input is the key we will be using when we call `PostForm.Get()`.

Putting this all together, along with some error handling, we can update **`Users.Create`** with the following Go code.

```
func (u Users) Create(w http.ResponseWriter, r *http.Request) {
        err := r.ParseForm()
        if err != nil {
                http.Error(w, "Unable to parse form submission.", http.StatusBadRequest)
                return
        }
        fmt.Fprintf(w, "<p>Email: %s</p>", r.PostForm.Get("email"))
        fmt.Fprintf(w, "<p>Password: %s</p>", r.PostForm.Get("password"))
}
```

If we restart our web server and submit the sign up form we should now see a response that has the email address and password we submitted!

Rather than calling **`r.PostForm.Get`**, another option is to use the **`FormValue`** method on the **`http.Request`** type. This would look like the following code.

```
r.FormValue("email")
```

Doing this automatically calls **`ParseForm`** for us, but it also ignores any errors returned by **`ParseForm`** and there are cases where multiple values can be assigned to single key (like an array), and **`FormValue`** only returns the first value assigned to the key. In 99% of cases both of these defaults are what you would want, but in some very specific situations it might not be the right choice for your application.

We can update our code to use **`FormValue`** and we end up with the following code.

```
func (u Users) Create(w http.ResponseWriter, r *http.Request) {
        fmt.Fprintf(w, "<p>Email: %s</p>", r.FormValue("email"))
        fmt.Fprintf(w, "<p>Password: %s</p>", r.FormValue("password"))
}
```

We won't be able to do anything with the form data until we learn about databases and building a user management system, but we have successfully parsed an incoming form in our Go code!

Aside 8.10. Third party libraries for form parsing

Packages like gorilla/schema exist and can make it much easier to parse forms into Go structs. For now I am sticking with the standard library to avoid hiding any details, but these work similar to how JSON parsing works in Go. We create struct types, define struct tags, and libraries like `gorilla/schema` can take an incoming request and parse the data into the struct.

Source Code

- Source Code - see the final source code for this lesson.
- Diff - see the changes made in this lesson.

8.7 URL Query Parameters

While learning how to parse a form in the previous lesson, it is very likely that one would encounter the term, "**query parameters**". While very similar to the form values we parsed in the previous lesson, URL query params do have a few differences. In this lesson we are going to explore what those are, and where we might use query parameters.

When submitting our sign up form, we used the POST HTTP method. Using the POST method allows us to attach a body to the web request, and our browser puts all of the form data into that request body. Later when we call

`FormValue` in our code, the request body gets parsed and we have access to the form data.

Another way to send data to a server is via URL query parameters. These are key-value pairs added to the URL after a question mark (`?`). For instance, the following URL has a query parameter with the key `page`, and the value `3`.

```
https://example.com/widgets?page=3
```

If we want to support multiple key-value pairs, we use the `&` character to separate them. The example below adds the key `color` and the value `green`.

```
https://example.com/widgets?page=3&color=green
                            |   |   |     |
                          key   |   |     |
                                |   |     |
                              value |     |
                                  key     |
                                      value
```

In both of these cases the path is still `/widgets`, and routers will only consider things before the `?` character when routing.

Aside 8.11. Encoding URL query parameters

Many characters need to be encoded when creating a URL with URL query parameters, so be sure to use something like url.Values if creating URLs with query params.

When submitting a form using POST, adding parameters to the URL isn't necessary because we can use the request body, but GET requests do not have a

request body. As a result, the only way to attach data to a GET request is to use headers and URL query parameters.

As we learned earlier, GET is intended to represent a read-only interaction with a web servers, so why would we need to add data to the query? One such reason is to filter the data we are requesting. For instance, adding pagination support can be done using URL query parameters.

Another use for URL query parameters in to pre-populate form values. This can be helpful if a user was referred to a website via an email and we want to fill in some of the information for them. When signing up for this course, the link emailed to you used this technique to fill in both the email address and the license key fields on the sign up form.

Pre-filling form data isn't necessary for an application to work, so it is often saved as a bonus feature once all of the development is done, but we are going to add support for pre-filling data in our sign up page to see how this works in Go. Open **`controllers/users.go`** and update **`New`** with the following code.

```
func (u Users) New(w http.ResponseWriter, r *http.Request) {
        var data struct {
                Email string
        }
        data.Email = r.FormValue("email")
        u.Templates.New.Execute(w, data)
}
```

The **`data`** variable we declare uses an anonymous struct with a single field - Email.

```
var data struct {
  Email string
}
```

Using an anonymous struct in cases like this can be nice since we don't have extra types polluting the entire package namespace. Instead, we just declare what we need directly in our function and proceed with it.

After that we assign the form value to the `Email` field, and finally we pass the `data` variable into `Execute` method call so that our template can access the data.

Moving on to your `signup.gohtml` template, we need to make use of the data being provided. We do this by adding a `value` attribute, and setting its value to the `Email` field. Find the email input and add the following line to it.

```
<input
  name="email"
  id="email"
  type="email"
  placeholder="Email address"
  required
  autocomplete="email"
  class="w-full px-3 py-2 border border-gray-300 placeholder-gray-500
    text-gray-800 rounded"
  value="{{.Email}}"
/>
```

After restarting our web server, we can now head over to http://localhost:3000/signup?email= and we should see the email address being pre-populated.

Aside 8.12. Encoding the @ character

It is possible you will need to update the URL to have the `@` character URL encoded. In that case the url will be http://localhost:3000/signup?email=user%40gmail.com, as the `@` character encodes into `%40` for URLs.

If the email query param is not present, our form will default to an empty email address. We can test this by clicking the "Sign up" button on the top right of our page.

When a user visits our sign up page, they currently have to click on an input box to start typing. Another way to improve the user experience is to have their cursor start inside of the first empty input box. We can do this with the **autofocus** HTML attribute.

```
<!-- This would be auto-focused with the cursor when a page loads -->
<input autofocus />
```

Before adding the email autocompletion we would have simply added this to the email input, but now that we can pre-fill the email address, we may want to add some logic deciding which field to autofocus. There are a number of ways to do this, but we are going to opt for the simplest option for our use case - checking to see if an **Email** field is present.

```
<div class="py-2">
  <label for="email" class="text-sm font-semibold text-gray-800">
    Email Address
  </label>
  <input
    name="email"
    id="email"
    type="email"
    placeholder="Email address"
    required
    autocomplete="email"
    class="w-full px-3 py-2 border border-gray-300 placeholder-gray-500
      text-gray-800 rounded"
    value="{{.Email}}"
    {{if not .Email}}autofocus{{end}}
  />
</div>
<div class="py-2">
  <label for="password" class="text-sm font-semibold text-gray-800">
    Password
  </label>
  <input
    name="password"
    id="password"
    type="password"
    placeholder="Password"
    required
    class="w-full px-3 py-2 border border-gray-300 placeholder-gray-500
      text-gray-800 rounded"
```

```
    {{if .Email}}autofocus{{end}}
  />
</div>
```

The new lines here are shown below.

```
<!-- Email input -->
<input ...
  {{if not .Email}}autofocus{{end}}
  />

<!-- Password input -->
<input ...
  {{if .Email}}autofocus{{end}}
  />
```

In the first case we are saying if there is not an **`Email`** field value (in other words, if the **`Email`** field is empty), we want to add **`autofocus`** to the email input. In the second we are saying we want to add the **`autofocus`** attribute to the password input if the email is present, because that means we were able to prefill that information.

There are a number of better ways to do this with more complex forms, but for now this is sufficient and should give you a taste of the type of logic you can add to a template. It is also important to remember that details like this aren't typically worth the effort until most of the application is complete, but it can be nice to see some of these small details that make an application feel more polished.

Source Code

- Source Code - see the final source code for this lesson.
- Diff - see the changes made in this lesson.

8.8 Exercises

Below are some exercises to help you practice and expand upon what we have been learning.

8.8.1 Ex1 - Experiment With Form Fields

We added a few inputs to our form, but it is worth adding a few more to make sure you understood everything we did.

For this exercise you can use the signup page, or create a new page entirely, but your goal is to add new inputs to a form, then use them to submit data to your server and try to parse them on the server.

If you want to get really complex, you could try looking at various input types. Eg a checkbox, or if you are feeling really ambitious you could even try to look at the file type.

If you get stuck at any point don't worry - we will continue creating forms as we progress and will eventually support file uploads. This is meant to be a chance to experiment and push yourself outside your comfort zone.

8.8.2 Ex2 - Adjust Your Paths

In at attempt to make sure you understand everything, try to alter the paths of a few of our endpoints. For instance, change the signup page to instead be located at `/users/new` and see what all needs to be changed to make this happen.

Alternatively, try alternating the user create endpoint between `POST /signup` and `POST /users` to see what is required to make this happen.

Source Code

- Source Code - see the final source code for this lesson.
- Diff - see the changes made in this lesson.

Chapter 9

Databases and PostgreSQL

9.1 Intro to Databases

Currently, our application doesn't persist any data. Not only does this lead to a pretty boring web app, but it also isn't very practical. Almost all useful features require storing user-provided data in some form or another. Whether that is storing a user's tweets so that we can share them with others, or storing images that a user uploads so they can share the gallery with their friends and family.

In this section we are going to learn about databases; specifically, we will be diving into PostgreSQL databases.

Aside 9.1. This section can be skipped if you are familiar with Postgres

All of the lessons in this section introduce and teach various aspects of SQL and Postgres. If you are already familiar with using SQL, you can likely skip the entire section.

Technically, we could use almost anything to persist data. For instance, we could simply open up a text file for each user and store information in it. Or we could use a CSV (comma separated values) file, which is basically a spreadsheet stored in a text file.

```
# users.csv
jon calhoun, jon@calhoun.io, pw-hash, ...
Lauren Smith, laurensmith@gmail.com, dif-pw-hash, ...
...
```

Unfortunately, a text file isn't likely to work very well in the long run. If we had millions of users, we would need to read through every row in the file to find a specific user every time they logged in. If we wanted to speed that up, we would need to invent a way of storing users in a sorted manner. Then we would need to make sure that sorted list of users stays updated even when users change their email address, new users sign up, etc.

We might also have to worry about race conditions where multiple processes attempt to alter the file at the same time and ultimately lead to invalid data.

Aside 9.2. More on race conditions

If you want to learn more about race conditions and what they are, I have a video on YouTube that goes into detail and shows examples of how race conditions can occur. You can find it at https://www.youtube.com/watch?v=lqDGzTh9kRg

Rather than reinventing the wheel and potentially introducing bugs, it makes more sense to use an existing database solution that can handle many of these details for us.

There are a variety of different database categories, and even within a specific database category there are often various implementations. Below are a few

popular database types, along with some popular databases that fall into that category.

- **Relational Databases** - PostgreSQL and MySQL are popular relational databases, though many others exist.
- **Document Stores** - MongoDB is likely the most popular document store. Another common one is Google's Firestore.
- **Graph Databases** - Neo4j is a popular database in this space, and others like Dgraph have come and gone.
- **Key-value Stores** - BoltDB is a key-value store written in Go.

If you hear the term NoSQL, it is referring to basically any category of database that is NOT a relational database. In the examples above graph, document, and key-value stores are all NoSQL.

In a relational database, commonly called SQL databases, there are tables for each resource that define the data a resource can have, and then we establish relationships between those resources. For instance, if we had a blogging platform we might have an `authors` table that stores a person's name, email address, and avatar. Then we might also have a `blog_posts` table that stores markdown, whether the post is published, tags for the post, and title. Each table has a predefined set of data that it stores.

With those tables we can define relationships between them. For instance, an author may have many blog posts, so we could define this relationship. In the future this would allow us to query all of the blog posts written by a specific author, or if we wanted to render a specific blog post we could look up that post's author to get up-to-date information about them.

In a document store, documents are a bit more self containing. For instance, rather than looking up a blog post, then looking up the author for that post as we would in a SQL database, a document store would likely store information

about the author inside of the blog post document. The upside to this is that we don't need to look at two tables in a database to render a blog post. We simply look up the blog post document in the document store. The downside is that if an author's information gets updates, we now have to go update it in all of those blog post documents. As a result, a document store might have faster reads but slower writes.

Graph database focus on shaping data around nodes and relationships, and while this sounds similar to a relational database, the key difference is that graph databases aren't restricted to predefined tables and relationships. I don't personally have a lot of experience using graph databases, so I unfortunately cannot give much additional insight into great use cases for them.

Key-value stores are databases that store data strictly using unique keys, and each key points to a specific piece of information. One upside to these databases is their simplicity once one learns how to use a key-value store. They can also be quite fast in many cases. One downside is that filtering and querying data in a key-value store is harder than in a relational database.

Every database has pros and cons when compared to other databases. If a database is great at one thing, it is making a tradeoff to achieve those results. Regardless of what you may read or hear, no single type of database is superior to all others. It simply depends on your needs and the tradeoffs that make the most sense for your application. As a result, many large companies use a variety of databases for different tasks. For instance, some companies use one type of database to power their application, and then convert the data into another database variant for internal tooling.

We will be using PostgreSQL in the course because it is a great all-around choice. It is:

- Very popular, which makes it easier to find articles and help when stuck.
- Free & open source, which means everyone can use it without paying high licencing fees.

- Scales very well without requiring extreme shifts in how we code or design our system.
- Is not overly complex to learn and get started with, but has a lot of power and complexity that can be learned as your application needs grow.
- Is well supported in Go.
- Has transactions to help us prevent race conditions.

In general SQL databases are a great overall option and most applications can scale to millions of users with a SQL database without any problems. It may require some optimizing over time, but this almost always stems from a poorly structured database rather than a shortcoming in the database engine itself.

At some point you will likely encounter an article that refers to SQL as a thing of the past; an outdated technology that you should move away from. Before believing it, I urge you to read this article: Relational Databases Arent Dinosaurs, Theyre Sharks.

Relational databases are still widely used because they work extremely well for most web applications. There are definitely situations where organizations grow very large and need to explore other options, but in a vast majority of cases a relational database is going to be a good choice to start with.

The rest of this section is going to be a crash course on installing and using Postgres. Along the way we will also learn a bit more about relational databases with some hands-on examples. While I will try to explain every query we make in this course, it may be beneficial to learn more about SQL after finishing the course so that you can expand on what is covered here. To help with this, the section ends with additional resources rather than exercises. Many of these are hands-on and will be much more useful than any exercises I could write.

Source Code

- Source Code - see the final source code for this lesson.

- Diff - see the changes made in this lesson.

9.2 Installing Postgres

Before we can use Postgres, we first need to install it on our machine. There are a few ways to do this, including installing it directly onto your computer, but we are going to instead use Docker in this course. If you don't already have Docker installed, I suggest starting that now as the download is about 500mb. Instructions for installing can be found on the Docker Website.

Aside 9.3. This is not a Docker course

Before proceeding, it is important to remember that this is not a Docker course. Docker is a very complex and large piece of software with a wide array of functionality. Learning all the details of Docker could take weeks on its own, and there are courses dedicated entirely to Docker. Rather than trying to learn Docker completely, this course will focus on the bits that are necessary while skipping mostly everything else. If Docker is intriguing to you, I suggest you research it further after completing the course.

When developing software, a common problem is developers having slightly different development environments. For instance, one developer might have Postgres v14.2 installed, while another has v13.6 installed. In many cases these differences won't matter, but in some they can result in bugs that present themselves in one version, but not another.

Docker is a tool that, among other things, can help alleviate this problem by helping ensure the tools we use in development are the same for every developer. This is done by using containers that specify all of the information about

the environment, dependencies, and the software that will be run, allowing us to run it consistently across all of our development machines.

Another benefit to Docker is that it can work across various operating systems. If we were to install Postgres directly onto our machine, the process, default usernames, default password, and more may all be different depending on what operating system everyone is using. If we instead use Docker and a container running Postgres, we can avoid this problem and ensure every developer who works on our application can easily replicate our database setup.

This all works because Docker containers are similar to little virtual machines. Each container can defines everything it needs, then when we use Docker to run a container it is as if Docker launched a small linux virtual machine that eventually runs software like Postgres.

First, we need to make sure Docker is installed and working on our machine. We can do this by opening our terminal and checking the versions for both **`docker`** and **`docker compose`**.

```
docker version
# You should see a version output. This may be large.
docker compose version
# Again, you should see a version. This will likely be a shorter output.
```

There are a few ways to launch containers using Docker. The **`docker`** command itself can be used, or we can use **`docker compose`** which will read from a **`docker-compose.yml`** file that defines all of the things our application needs to be running. The latter is nice for projects like ours where we want to define all of the things we need and check it into source control so that other developers can also launch the same docker containers.

Create a **`docker-compose.yml`** file.

```
code docker-compose.yml
```

Add the following to the YAML file we just created. We will explain what each line does in detail once it is setup and running.

```
version: "3.9"

services:
  # Our Postgres database
  db: # The service will be named db.
    image: postgres # The postgres image will be used
    restart: always # Always try to restart if this stops running
    environment: # Provide environment variables
      POSTGRES_USER: baloo # POSTGRES_USER env var w/ value baloo
      POSTGRES_PASSWORD: junglebook
      POSTGRES_DB: lenslocked
    ports: # Expose ports so that apps not running via docker compose can connect to them.
      - 5432:5432 # format here is "port on our machine":"port on container"

  # Adminer provides a nice little web UI to connect to databases
  adminer:
    image: adminer
    restart: always
    environment:
      ADMINER_DESIGN: dracula # Pick a theme - https://github.com/vrana/adminer/tree/master/des
    ports:
      - 3333:8080
```

We can start things up using Docker compose's **up** subcommand. It is worth doing this now as this may require a few downloads to happen in the background. These downloads will be cached for the future, but they still need downloaded the first time.

```
docker compose up
```

It is worth running this command now, as it may require a few downloads to happen in the background. These downloads will be cached for the future, but they still need downloaded the first time.

Aside 9.4. Possible errors

If you already have Postgres installed, it is possible that you will run into an error because both the docker version of Postgres and the one you have installed both want to use port `5432`, and only one application can listen on a port at a time.

One way to resolve this is to close Postgres, but restarting might lead to it starting back up again, so this is a short term solution. Postgres could also be uninstalled, but again this might not be an ideal solution. If neither of those options work, the last option is to use a different port for our docker Postgres. We do this by opening up the `docker-compose.yml` file and updating the port we use there.

```
db: # The service will be named db.
  # ... don't change this stuff

  # Update the port on our machine (the first one) to be an unused port
  ports:
    - 5433:5432 # format here is "port on our machine":"port on container"
```

This allows us to connect port `5432` inside the docker container to port `5433` on our local machine, and port `5433` should be available. If it isn't, pick another one that is free.

In future lessons where we connect to Postgres with Go the new port will need to be used, so keep that in mind.

Now let's break down what each line does while Docker downloads what it needs. The first line, `version: "3.9"` defines the version we will be using. After that we have our list of services.

```
services:
  db:
    ...
  adminer:
    ...
```

This is telling Docker that we will have two services, one named **db**, and another named **adminer**. Everything indented inside of each of these is used to describe that particular service.

Inside the **db** service, we have the following:

```
# Our Postgres database
db: # The service will be named db.
  image: postgres # The postgres image will be used
  restart: always # Always try to restart if this stops running
  environment: # Provide environment variables
    POSTGRES_USER: baloo # POSTGRES_USER env var w/ value baloo
    POSTGRES_PASSWORD: junglebook
    POSTGRES_DB: lenslocked
  ports: # Expose ports so that apps not running via docker compose can connect to them.
    - 5432:5432 # format here is "port on our machine":"port on container"
```

The **db** service is meant to describe our database service, hence the name **db**. The first line inside the service declaration defines the image that the container will be created from.

```
image: postgres
```

The Docker Hub has an official postgres image, as well as images for many other common pieces of software. The value **postgres** is the name of this image.

Next we have the restart line. This tells Docker that if the service stops running, it should restart it.

```
restart: always
```

The environment block is used to provide environment variables for Docker containers.

```
environment: # Provide environment variables
  POSTGRES_USER: baloo # POSTGRES_USER env var w/ value baloo
  POSTGRES_PASSWORD: junglebook
  POSTGRES_DB: lenslocked
```

In the postgres image docs there is a section describing environment variables that will be used when first installing Postgres. We are using a few to set the username, password, and database name.

It is important to note that these environment variables are only used the first time the container is launched. In the future when we run our docker containers, it will keep the same database and user until we explicitly tell docker to delete our containers and start from scratch.

Finally, we have the ports section.

```
ports:
  - 5432:5432 # format here is "port on our machine":"port on container"
```

When we run services with a **`docker compose`**, they will default to a private network. This means any code running on our local machine wouldn't be able to connect to the Postgres database. To address this, we define a port on our local machine, and a port inside the Docker container, and we link the two together. In this particular case we are using the same port for both, which means later when we connect to **`localhost:5432`** we will find a Postgres database running there.

In addition to Postgres, I have also included the **`adminer`** service which is a simple way to view our database, run some SQL queries in a browser, and do

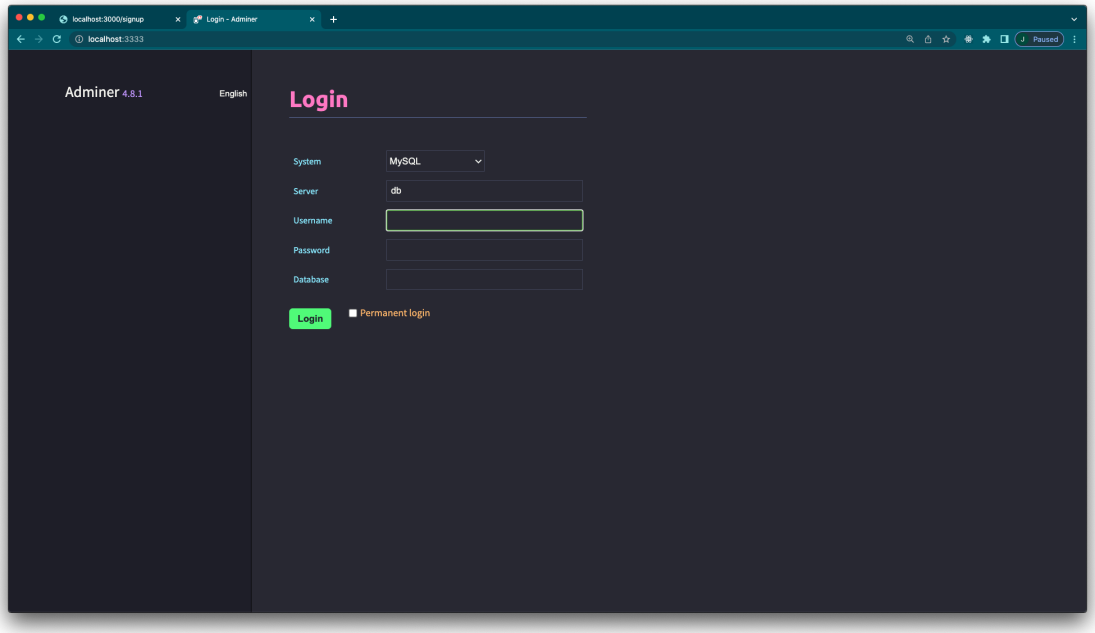
other development related tasks. It is using the adminer image, which uses the software from adminer.org.

The adminer service is setup in a manner similar to our db service. It has an image, it will always restart, and we even use an environment variable to set the theme. The most notable difference is the ports section.

```
ports:
  - 3333:8080
```

This time we aren't mapping a local port to the same port inside the container; instead, we are mapping the port `3333` on our local machine to the port `8080` inside of the adminer container. This isn't absolutely necessary as we likely don't have any services running locally on port 8080, but it is worth knowing for the future when you might have multiple Docker services that all use port 8080 internally, but if we want to expose them they will each need a unique port on our local machine.

Once our docker compose finishes running everything, visit



Once you have Adminer loaded in your browser, fill in the following information.

Option	Selection
System	Postgres
Server	db
Username	baloo
Password	junglebook
Database	lenslocked

You can check the permanent login option as well if you want.

The system option tells adminer what type of database we are connecting to. The server option tells it what server to connect to, but because everything is running inside of a private Docker network, we can simply provide the database service name (`db`). If we were to change our service name inside the `docker-compose.yml` file, this value would also need updated.

The username, password, and database are all values we set in our `docker-compose.yml` file.

Aside 9.5. Rebuilding the services from scratch

If you happened to use the wrong postgres username, password, or some other value, simply stopping and restarting Docker compose won't update then. This is because Docker has already built the container and only restarts it to save time. To take the container down and build a new one, use the `down` subcommand.

```
docker compose down
```

This will tell Docker to take down the containers and build fresh ones the next time we run `docker compose up`.

If everything is working, you should be able to log in with adminer and see a dashboard. It will be pretty sparse until we start adding to our database, but successfully logging in means we have Postgres running locally via Docker!

If you want to stop the docker containers, head to your terminal and press `ctrl+c` wherever they are running. You may need to press it a second time to force it to stop.

We can also use the `-d` flag to run our docker containers in detached mode. In other words, this will free up our terminal to do other things while the services continue running.

```
docker compose up -d
```

To stop services running in detached mode, simply use the `stop` subcommand

from the directory with your `docker-compose.yml` file.

```
docker compose stop
```

It is possible that Docker will run the next time you start up your computer and start running these services in the background. If that happens, the `stop` subcommand will also stop these if you want, or if you are on Windows or Mac OS you can use the Docker Desktop app to stop them.

Source Code

- Source Code - see the final source code for this lesson.
- Diff - see the changes made in this lesson.

9.3 Connecting to Postgres

We connected to our database via Adminer in the last lesson to verify that Postgres is running. In this lesson we are going to take it a step further and explore how to connect to our database using the `psql` CLI. This will enable us to execute queries in the database with SQL statements.

First, we should make sure our database is running. We can check by running the following:

```
docker compose ls
```

This will show us a list of any apps running via docker.

```
NAME                STATUS
lenslocked          running(2)
```

The name `lenslocked` here stems from the directory we have our `docker-compose.yml` file in. If your directory is different, it might show up differently here.

If nothing is running, only the `NAME` and `STATUS` headers will be shown, in which case you will want to run the `up` subcommand.

```
docker compose up
```

If we had installed Postgres directly onto our machine, we would have the CLI tool `psql` available to use. This is command line application that allows users to connect to a Postgres database and execute SQL queries.

We didn't install Postgres directly, and instead opted to use Docker. As a result, we may not have the CLI available on our local machine. Instead, we need to use the `psql` binary that is inside of the Docker container where Postgres is installed and running.

To run a binary inside of a Docker container, we are going to use the `docker exec` command. We also need to use a few flags and arguments, including a username and database we want to connect to. The complete command is shown below, and the flags are case sensitive.

```
docker compose exec -it db psql -U baloo -d lenslocked
```

Aside 9.6. The videos use a different command

When recording the videos, I used a command similar to the following:

```
docker exec -it lenslocked-db-1 /usr/bin/psql -U baloo -d lenslocked
```

While this command works, it is a bit less versatile since we need to know the name of a container to run `psql` inside of. As a result, I will be updating the course to use the `docker compose exec` command. I won't be recording every video again, but I will add a snippet of video showing the updated command.

Let's break this command down. First we have:

```
docker compose exec
```

This lets us execute a binary inside of a container that is being run by docker compose.

Next is the `-it` flag, which is actually two flags: `-i` and `-t`. We can merge them into a single flag in this case to save on some typing. These two flags are common when running docker exec as they make it so we can interact with the terminal session after running a command. This makes it possible to run SQL queries with `psql` after running our docker exec command, much like we would if `psql` was installed locally.

Next we have `db`, which is the name of the service we want to use to execute our command. This comes directly from our `docker-compose.yml` file.

The next argument is `psql`, which is the binary that we want to run. In some cases we may need to use an absolute path, but that isn't necessary for `psql` in our container.

After listing the binary we want to run, we can also pass in flags for the binary. In our case we want to run the following command inside of our container.

```
psql -U baloo -d lenslocked
```

The `-U` flag lets us set our username, and the `-d` flag lets us state which database we want to connect to. The values for these were `baloo` and `lenslocked` in the `docker-compose.yml` we created.

Putting this all together, we again have the following command:

```
docker compose exec -it db psql -U baloo -d lenslocked
```

Run the command and verify you are connected to your database with `psql`. Your terminal should have output similar to that below.

```
psql (14.2 (Debian 14.2-1.pgdg110+1))
Type "help" for help.

lenslocked=#
```

You should also be able to write SQL queries into the terminal at this point.

Aside 9.7. Document useful commands

I find it useful to document useful commands in a project `README.md` file for both myself and future developers. For instance, if a specific command is used to compile the code, or if there are developer tools that others might not be familiar with in the future.

The `docker compose exec` command we just composed is another example of this. Once you have a working command, it is a good idea to add it to a README file so you can reference it in the future if you forget.

Now that we have `psql` running we can perform a few queries to test it out. First we can create a table to store data in.

```
CREATE TABLE users (
  id SERIAL PRIMARY KEY,
  email TEXT
);
-- Output:
-- CREATE TABLE
```

Next we can insert a user into the table.

```
INSERT INTO users (email) VALUES ('jon@calhoun.io');
-- Output:
-- INSERT 0 1
```

And finally we can query for all users in the users table.

```
SELECT * FROM users;
-- Output:
--  id |     email
-- ----+---------------
--   1 | jon@calhoun.io
-- (1 row)
```

We will learn more about how each of these commands works in future lessons. For now the goal is just to verify that we can use the `psql` CLI to interact with our database.

To quit out of psql, press `ctrl+d` or type and press enter.

9.4 Update: Docker Container Names

In the screencasts we previously used `lenslocked_db_1` for the docker exec command, but due to an update in Docker this is now `lenslocked-db-1` on

my machine. I have also moved the screencasts code to a directory named `lenslocked-casts` which will further change the container name.

Given that this information can change based on your directory name, your OS, and your version of Docker, it is best to learn how to look at any running containers and evaluate the name you should use on your own. To do this, I suggest first making sure your docker containers are running, then to use `docker ps` to see what the names are.

```
docker compose up -d
docker ps
```

This should result in an output similar to the table below.

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	P
46a59ecf6101	adminer	"entrypoint.sh docke..."	2 minutes ago	Up 2 minutes	0
8aa687c556bb	postgres	"docker-entrypoint.s..."	2 minutes ago	Up 2 minutes	0

When looking at thi stable, the `NAMES` section should provide you with the information you need for `docker exec`.

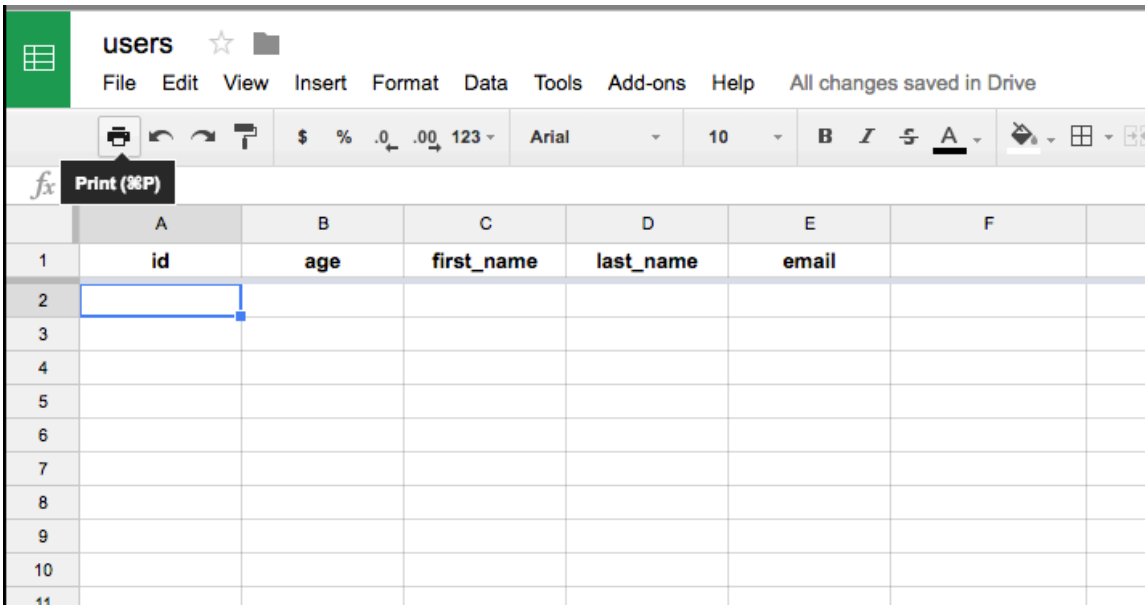
```
docker exec -it <NAME> /usr/bin/psql -U baloo -d lenslocked
```

9.5 Creating SQL Tables

Postgres is a relational database with a strict structure. If we want to store any information, we need to create a table and define the format that the data will be stored in. We do this by creating a table in the database.

Tables are a little easier to understand if you think of them like a spreadsheet. In order to start storing data, we first need to create a spreadsheet and label

each column.



Once we have a spreadsheet setup we can add data to it by adding a new row. Each row represents a record in our database, and each column is a specific piece of information for that record. In the image above, we are storing an id, age, first_name, last_name, and email address for each record in the users spreadsheet.

A table is typically created with a SQL statement formatted similar to the one below.

```
CREATE TABLE table_name (
  field_name TYPE CONSTRAINTS,
  next_field_name TYPE(args) CONSTRAINTS
);
```

We always declare the table name first, and then with each row we define a field (also called a column) and then we declare what type this field will be, followed by any constraints we have on the field.

Some types require additional arguments, which is why you will sometimes see parenthesis after the type with some values in it. In the example above the first field has a **`TYPE`** with no arguments, while the second has **`TYPE(args)`** with arguments in parenthesis. **`TYPE`** isn't a real type in Postgres, but you could create a **`VARCHAR(80)`** which is like a string, but has a limit of 80 characters.

After the types we see the constraints. These are rules that we can set on each column that the database will enforce, such as whether a field needs to be unique across all records, or if a field can be null.

We will discuss both types and constraints as we proceed through the course, but if you are curious about what options are available you can find them in the PostgreSQL Docs: data types and constraints.

If we wanted to create a table to store users using SQL, we might use a statement like the one below.

```
CREATE TABLE users (
  id SERIAL,
  age INT,
  first_name TEXT,
  last_name TEXT,
  email TEXT
);
```

This table doesn't have any constraints, but it demonstrates how a CREATE TABLE statement might look with several fields. Each field is like a column in our spreadsheet we looked at earlier and each row added to the table is a new user entry.

As we discussed earlier, the outer portion tells the our database that we want to create a table, and the table will be named **`users`**.

```
CREATE TABLE users( ... );
```

Everything inside of this defines the users table that we are trying to create. In this particular case we have:

- A field named `id` with a type of `SERIAL`
- A field named `age` with a type of `INT`
- A field named `first_name` with a type of `TEXT`
- A field named `last_name` with a type of `TEXT`
- A field named `email` with a type of `TEXT`

Running this on our current database won't work because we created a table named users in the last lesson. If we wanted to create a new table named users, we would first need to delete the existing table. We do this with the `DROP TABLE` command. We can even append an `IF EXISTS` so it only runs if the table exists, and otherwise just skips the SQL command.

```
DROP TABLE IF EXISTS users;
```

Over the next couple lessons we will explore Postgres data types and constraints, then we will put this all together to create a more realistic users table for our app.

Source Code

- Source Code - see the final source code for this lesson.
- Diff - see the changes made in this lesson.

9.6 Postgres Data Types

Types provide a way to define what kind of data you want to store in a particular column. For example, you might say "I only want to store integers in this

column" or "I only want to store strings that are smaller than 140 characters in this column".

PostgreSQL supports a wide variety of data types, but in this course we will only need to use a handful of them. Below are a few common data types:

Type	Description
`int`	This is used to store integers between -2147483648 and 2147483647.
`serial`	This is used to store integers between 1 and 2147483647. The big difference b
`varchar`	This is like a string in Go or other programming languages, except we have to
`text`	This is a type that is specific to PostgreSQL (and may not be available in all fo

In Postgres I prefer `text` over `varchar` in most cases, but it is important to know that both exist. Not all SQL databases have the same types, and `text` is one that is Postgres specific.

As we progress through the course these types will start to make more sense as we use them to define our database tables.

9.7 Postgres Constraints

Constraints are rules that we can apply to columns in our table. For example, we might want to ensure that every user in our database has a unique `id`, so we could use the `UNIQUE` constraint. We could also verify that an integer column is above or below a specific value.

Like data types, there are many constraints available in Postgres. For now we will only be using a few:

Constraint	Description
`UNIQUE`	This ensures that every record in your database has a unique value for the
`NOT NULL`	This ensure that every record in your database has a value for this field. V
`PRIMARY KEY`	This constraint is similar to combining both `UNIQUE` and `NOT NULL` but

While it is common to validate data inside our own Go code, many constraints need to also be set at the database layer to ensure they are always enforced. For instance, if two users submit a form at the same time it is possible for your code to process both forms at roughly the same time and, depending on the timing, each might check and verify that a field is unique before either has inserted data into the database. As a result, it is possible to end up with a duplicate on a field that should be unique if validations only occur in your application code. On the other hand, if we set up constraints on your database one of the two submissions would fail when our Go code attempted to create a record with duplicate data.

In short, it is important to use database level constraints, and we will discuss each of these that we need as the need arises.

9.8 Creating a Users Table

Now that we have seen a few data types and constraints in Postgres, we are going to put that knowledge together to create a new users table that incorporates it all. First, we need to connect to our database.

```
docker exec -it lenslocked-db-1 /usr/bin/psql -U baloo -d lenslocked
```

Aside 9.8. Container names may vary

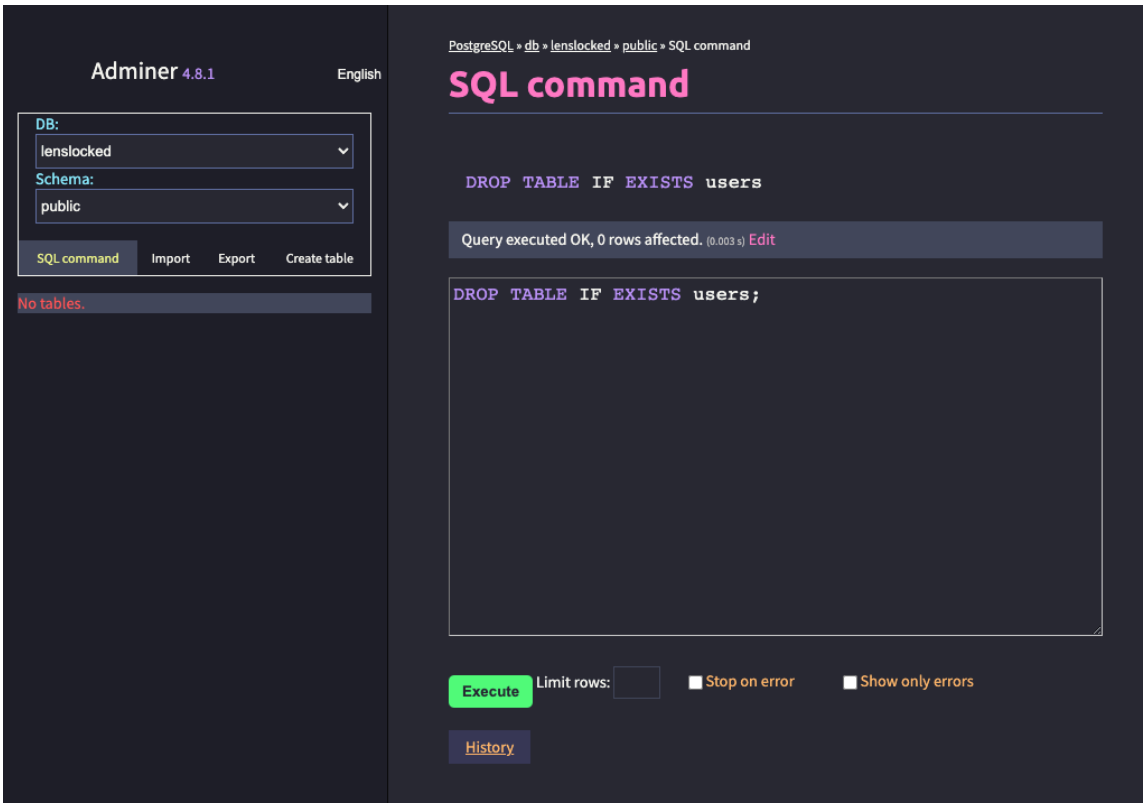
The command to run `psql` inside of a docker container should be similar to the one above, but the name of the container (`lenslocked-db-1`) might vary based on your local setup. Again, use `docker ps` to see what containers you have running.

We likely already have a users table from previous lessons, so we can drop that

to make room for a new table named users. This command gets run inside of **psql**.

```
DROP TABLE IF EXISTS users;
```

Alternatively, it can be run inside of Adminer by clicking the "SQL Command" link on the left of the page, then entering the SQL into the input box.



Regardless of how we run the command, it will drop the users table if it exists. In other words, it will delete both the users table, and all of our users data stored inside of it! This is important to remember, as running a DROP TABLE command in production could result in losing valuable data.

We are now prepared to create a table for users. It will be similar to the ones we have seen already, but we are going to add a few constraints.

```
CREATE TABLE users (
  id SERIAL PRIMARY KEY,
  age INT,
  first_name TEXT,
  last_name TEXT,
  email TEXT UNIQUE NOT NULL
);
```

You should see the output **CREATE TABLE** after running the query. If you instead see **ERROR: relation "users" already exists** that means you already created a users table. To remove the table and recreate it you would need to run the **DROP TABLE** bit we did earlier in this lesson.

Now let's discuss the constraints we added. **PRIMARY KEY** is a special constraint that is only used once per table. It tells our database that this is the main way we will be referencing the resource. The PRIMARY KEY constraint is typically set to an **id** field with a type of **SERIAL** like we are doing here, giving other tables an easy way to uniquely reference a user.

Behind the scenes the **PRIMARY KEY** constraint results in a few constraints being applied. The first is the **UNIQUE** constraint, which forces every user to have a different value for the specific field. In our users table, we are saying that no two users can have the same ID.

PRIMARY KEY also adds the **NOT NULL** constraint, which means that we cannot create a user record with a null ID.

Finally, PRIMARY KEY creates an index for the ID field. This makes it faster to lookup users via their ID, but indexes also need to be kept up-to-date, so it also means anytime we create or update a user the ID index might need updated. This is why most SQL databases only index fields we intend to search quite frequently.

All of these constraints make it easier to associate other records to a specific

user. For instance, when we create a gallery resource in our database, it will have a `user_id` field that has the value of one and only one user's ID. That user will be considered the owner of the gallery, having permission to update and delete the gallery. If two users could have the same ID, it would introduce ambiguity that we do not want, but since we know every user will have a unique ID we can reliably use it.

Later when we get to the `email` field we use the constraints `UNIQUE` and `NOT NULL`. These constraints are similar to what the PRIMARY KEY constraint applies, but in this case we add the constraints to ensure that every users has a unique email address.

Aside 9.9. Why not set email as the primary key?

When we create other tables in our database, they will reference specific users via their primary key - their ID. For instance, when we create the gallery table, it will have a `user_id` field that has the ID of the user who created and owns the gallery.

We could instead use the email address as the primary field and then use `user_email` to link our galleries to users, but what happens if a user wants to update their email address? We would then need to go update every gallery record that the user owns as well! Our database would also need to update the index of users it creates based on the primary key, causing even more work.

In relational databases, the primary key should be set to a field that doesn't change over time. While an email address doesn't change often, it is better to use a unique ID field that shouldn't ever change for a specific user.

9.9 Inserting Records

Now that we have seen how to create tables and define the structure of our database we are going to start learning how to manipulate data. We will begin by inserting records, then we will look at how to query, update, and delete data.

Aside 9.10. Open psql or Adminer for the following lessons

In the next few lessons we will be writing various SQL statements and executing them against our database. You are welcome to use Adminer or `psql` to execute the SQL.

We will use the `INSERT` statement to add new records to a table in our database. There are basically two ways to use the INSERT command:

1. By providing values for every column in the table in the same order as the columns of the table.
2. By providing the columns that you want to set values for, and then values for each of those columns.

The first is a bit less flexible because we need to provide every value, and if our table has a new field added it could break our previously working code. It also means we need to provide an ID rather than letting the database provide one for us. Still, it is worth seeing what that looks like, so let's start there. Execute the following SQL statement against the database.

```
INSERT INTO users
VALUES (1, 22, 'John', 'Smith', 'john@smith.com');
```

In this case the values represent the `id`, `age`, `first_name`, `last_name`, and `email` fields in that order. This order is important, and is dictated by how the table is created. If we were to mix that order up, values would be placed in the wrong columns!

After running the insert command you should see the output `INSERT 0 1`. If you happen to see something like `ERROR: duplicate key value violates unique constraint "users_email_key"` that means you likely already inserted a record with the same email address you provided.

If we want to see all of the records in our table we can run the following query.

```
SELECT * FROM users;
```

This will return an ascii table similar to the one below.

```
 id | age | first_name | last_name |     email
----+-----+------------+-----------+----------------
  1 |  22 | John       | Smith     | john@smith.com
(1 row)
```

Rather than executing insert queries with every value, we will be using named values. An example of this is shown below, though running it might result in an error the first time.

```
INSERT INTO users (age, email, first_name, last_name)
VALUES (30, 'jon@calhoun.io', 'Jonathan', 'Calhoun');
```

At this point the query is likely going to result in an error like the one below.

```
ERROR:  duplicate key value violates unique constraint "users_pkey"
DETAIL:  Key (id)=(1) already exists.
```

What is happening here is that our **SERIAL** value in the database tried to use the value **1**, but we already created a user with that value. The SERIAL type doesn't know about any IDs that we set manually, so it will result in an error when it increments and attempts to use a value already taken.

If we were to run the exact same **INSERT INTO** query again, it will likely work as long as the id **2** hasn't been taken by a user record yet.

In addition to not being aware of ID values we set manually, the SERIAL type will also increment the value to be used every time we attempt to insert a record regardless of whether it was successful or not. As a result, we will get an error when SERIAL encounters a value that is taken for our user ID, and it also means that if we attempt to insert a user with a null email address, the SERIAL type for ID will still increment even though our insert had an error. As a result, there is no guarantee that every ID will be used even if using the SERIAL type. The only guarantee we get is that every ID will be greater than any previously created IDs and the SERIAL type will try to increment one at a time.

If you want to test this, try creating a record with an email address that already exists in the database. You should see an error. Next, create a record with an unused email address. Finally, query your records and examine the results. Were any IDs skipped?

```
SELECT * FROM users;
```

With that out of the way, we can focus on the SQL INSERT statement that uses named values.

```
INSERT INTO users (age, email, first_name, last_name)
--                  ^ these are the names of columns
VALUES (30, 'jon@calhoun.io', 'Jonathan', 'Calhoun');
--      ^ these values must match the order above
```

We start by naming the columns we want to insert data into. These come immediately after the name of our table and are all inside a set of parenthesis. After

that we use the `VALUES` keyword and then provide values for the columns in the order we declared them in our SQL statement.

Aside 9.11. Comments in SQL

`--` is how you denote a comment in SQL, and new lines can be placed in between a SQL statement, as it will not be completed until it ends with a semicolo (`;`).

Using named values provides several benefits:

- We do not need to declare an `id` value, and can instead let the database set this for us using the SERIAL type.
- We can also skip other fields that can be auto-populated. A common example for this is a `created_at` column that is automatically populated with the current timestamp.
- We don't have to guess at the order of columns which could result in data in the wrong column. Instead, we define the columns we want to set data for and it is clear from our SQL statement what that order is.
- Our code won't break if a new column is added to the table!

In short, queries with named values are a much better option for our needs.

9.10 Querying Records

When inserting data, SQL feels quite picky compared to other database options. We need to declare tables and stick to a predefined structure rather than just

throwing data at the database in whatever format seems best at the time. The upside to this approach is that SQL can be quite powerful when it comes to querying for data. With time and practice one will be able to make complex queries answering almost any question they have about their data, but for now we will focus on learning the basics.

In order to query data from a SQL database, we create a **SELECT** statement. We have already seen this when we looked at all the users in our database.

```
SELECT * FROM users;
```

When we run this query, we see an ASCII table with the users in our database. The values in your database might not match the output below perfectly, but you should see all the records in your users table.

```
 id | age | first_name | last_name |     email
----+-----+------------+-----------+----------------
  1 |  22 | John       | Smith     | john@smith.com
  2 |  30 | Jonathan   | Calhoun   | jon@calhoun.io
(2 rows)
```

Aside 9.12. The ASCII table is just a pretty way of visualizing data

When we start running queries with our Go code, we won't need to parse tables like we are seeing. These are shown in **psql** as a way of making the data look a bit prettier and easier to read.

The asterisk (*) is a handy way of saying, "Give me all of the columns from the table." If we only cared about a few fields, we could explicitly request that data.

```
SELECT id, email FROM users;
```

Running this query will return a much smaller set of data as it doesn't need to include any data we didn't request.

```
 id |     email
----+----------------
  1 | john@smith.com
  2 | jon@calhoun.io
(2 rows)
```

That's it for the basics! As we progress through the course and use SQL more and more we will see ways to query related data in more complex ways, but even in those complex queries we are always asking our database to give us a table with some data as the end result.

9.11 Filtering Queries

Thus far all of our SQL queries have returned all of the data in a specific table, but more often than not you won't actually want ALL of the users. Instead, you will only care about a few users that match specific criteria. In those cases, the **WHERE** clause becomes useful. WHERE allows us to add conditions to our query and limit the data we are receiving. Below is an example that only queries for users with a specific email address.

```
SELECT * FROM users WHERE email='jon@calhoun.io';
```

In addition to exact matches, there are a number of other queries we can perform. For instance, we can query for users who are older than 22.

```
SELECT * FROM users WHERE age > 22;
```

We can also use **AND** and **OR** to combine queries, creating slightly more complex queries.

```
SELECT * FROM users
WHERE age < 30 OR last_name = 'Calhoun';
```

Queries can become much more complex and can involve querying data from multiple tables, computations, and more. We will look at these queries in the future as we need them, or you can look at the resources listed at the end of this section to learn more.

9.12 Updating Records

Updating a record in SQL is done using the **UPDATE** query.

```
UPDATE users
SET first_name = 'Johnny', last_name = 'Appleseed'
WHERE id = 1;
```

We start by specifying which table we want to update, then we follow that with the **SET** keyword and a list of fields along with values we want to set that field to. At this point our UPDATE would technically work, but without any filtering it will be applied to ALL records in the database table! For example, the following query will set the **first_name** of every user to "Jon".

```
UPDATE users
SET first_name = 'Jon';
```

While this might be intended every once in a while, it likely isn't what we want all the time. To limit an UPDATE to specific records, we use the **WHERE** clause much like we would with a SELECT statement. In many cases this will only specify the ID of the record we want to update, ensuring that only one item is updated in our database, but we could also update all users with a specific age range if we want.

```
UPDATE users
SET first_name = 'Anonymous', last_name = 'Teenager'
WHERE age < 20 AND age > 12;
```

Generally speaking, the safest way to update records is to use their **id** as part of the SQL statement. This ensures that only one record will match the query.

9.13 Deleting Records

Deleting records is similar to updating them, except we use the **DELETE FROM** keywords. It can be used to delete multiple records at a time, so we need to pair it with a WHERE to ensure it only deletes the records we want to delete.

```
DELETE FROM users
WHERE id = 1;
```

If you forget to include the WHERE clause you can easily delete all records from a table, and once again using the **id = 1** filter to ensure only one record gets updated it the most common method of deleting database entries.

9.14 Additional SQL Resources

While I try to explain all of the code we write in this course, it isn't possible to teach every technology used exhaustively. As a result, readers who are unfamiliar with SQL may benefit from additional resources. Even those familiar with SQL might benefit from additional resources, and many that exist now can be quite fun.

9.14.1 Learning Resources

Below are several great resources for continuing your SQL education. Many are hands-on and will require you to write SQL statements to complete. These are intended to replace exercises for this section.

Select Star SQL - From the website:

> This is an interactive book which aims to be the best place on the internet for learning SQL. It is free of charge, free of ads and doesn't require registration or downloads. It helps you learn by running queries against a real-world dataset to complete projects of consequence. It is not a mere reference page — it conveys a mental model for writing SQL.
>
> I expect little to no coding knowledge. Each chapter is designed to take about 30 minutes. As more of the world's data is stored in databases, I expect that this time will pay rich dividends!

It can be found at https://selectstarsql.com/ and is a great resource to start with.

SQL Murder Mystery - After learning the basics through something like Select Star SQL, this is a great place to practice and hone your skills. It is a murder mystery that you solve by querying data. https://mystery.knightlab.com/

Codecademy's Learn SQL course - This is an interactive course that includes a few small projects and quizzes to help you get familiar with SQL. You can find it at https://www.codecademy.com/learn/learn-sql

w3schools.com - An SQL course that cover nearly every aspect of SQL. Similarly to Codecademy, most of these allow you to try the code samples to see what they actually do. You can find the SQL course at http://www.w3schools.com/sql/

Khan Academy's SQL course - This one can be found at https://www.khanacademy.org/co programming/sql

If none of those pan out, there is an entire Quora question titled "How do I learn SQL?" that has 100+ answers and a large list of potential resources to check out here: https://www.quora.com/How-do-I-learn-SQL

9.14.2 SQL Tools

If you don't care for Adminer or `psql`, there are other options available to interact with your database. Below are two of many options out there.

- PopSQL - Clean UI, native desktop app, team collaboration, and more, but it is a bit pricey if you aren't working at a larger org. There is a free tier for a single user.
- SQLECTRON - Nice free desktop app that is open source. A great alternative if you want something non-commercial with a GUI.

Chapter 10

Using Postgres with Go

10.1 Connecting to Postgres with Go

Now that we have some experience using SQL, we are going to explore using Go to connect to our SQL database. A vast majority of what we will be doing is writing SQL queries like we did in the previous section, but we will also need to lear a few Go specific details along the way.

Go provides us with the database/sql package, which is a set of common functions and types needed to interact with a SQL database. What the standard library does not provide are drivers for each specific SQL database type. For that we will need what is known as a driver. We are using Postgres for our database, so we will need a Postgres driver, but if we were using MySQL, SQLite3, or something else we would need those drivers instead.

By separating the driver and the `database/sql` packages, it is possible for community members who are experts in each specific database variant to provide drivers. It also means that any new SQL databases can provide their own driver rather than relying on the Go team to prioritize it. Finally, it also means there can be multiple drivers for any specific database variant.

Postgres has several drivers available, but my personal favorite is jackc/pgx. It is written purely in Go (vs using cgo), so it should compile and work on any operating system. It is also quite popular and stable, so it is unlikely that we will encounter any bugs using it.

pgx is a third party libary, so we will need to use **go get** to add it to our application. We will be using **v4** in this course, so we will also add that to the **go get** command. Run the following in your terminal.

```
go get github.com/jackc/pgx/v4
```

The Go tooling should install any necessary dependencies and update your **go.mod** file.

Next we need to import the driver provided by this package. We will be working on some experimental code while we go over the basics, so open up that **exp.go** source file.

```
code cmd/exp/exp.go
```

Update the file with the following Go code:

```
package main

import (
        _ "github.com/jackc/pgx/v4/stdlib"
)

func main() {
}
```

Our **main()** function is empty and our code doesn't do anything, but we somehow have an import. We will explore why we need this in the next lesson, but

for now the key takeaway is that when we use the _ character in front of an import, we are telling the Go compiler that we want to include that import even if we don't seem to use the package. This is very similar to using the underscore for unused variables.

We need to open up a database connection before we can send SQL to the database. To do this we will use the Open function form the **database/sql** package.

```
package main

import (
        "database/sql"

        _ "github.com/jackc/pgx/v4/stdlib"
)

func main() {
        db, err := sql.Open("pgx", "host=localhost port=5432 user=baloo password=junglebook dbn
        if err != nil {
                panic(err)
        }
        defer db.Close()
}
```

The **Open** function takes two arguments:

1. The database driver name. This is determined by the SQL driver you are using, and in our case the name chosen by the pgx library is **"pgx"**.

2. The connection string. There are a few valid formats for this, and each SQL variant might allow slightly different formats, but this will be a string that has information needed to connect to your database. The format we are using is specific to Postgres, and each value stems from how we setup the Postgres database in **docker-compose.yml**. The only value we didn't explicitly set up is the **sslmode** which needs to be set to **disable** to work right now since we are connecting to a database running locally.

After calling `Open` we check for an error and panic if we have one. This isn't the best way to handle errors, but will work for now since this is experimental code.

Finally, if we were successful in opening a connection we want to make sure that connection is eventually closed. We do this by using `defer` and the `db.Close()` method. Defer tells our Go code to close the database connection whenever the `main()` function ends.

At this point it might appear that we have successfully connected to our database, but that isn't the entire story. When calling `sql.Open()`, it only initializes a connection to our database. It doesn't actually verify the credentials we provided work, or that the database is actually available. To verify that we can communicate with the database, we need to execute some SQL or call a method like db.Ping.

Add the following code to ping the database, check for an error, and print out a success message if all goes well:

```
err = db.Ping()
if err != nil {
  panic(err)
}

fmt.Println("Connected!")
```

Run the code to verify we can connect to our database. Make sure Docker is running with your database as well, otherwise this won't work!

```
go run cmd/exp/main.go
```

If everything went well we will see the output `Connected!` after our program runs. If we instead see an error we need to debug what is going on. Below are a few common errors and their causes.

- `panic: pq: role "baloo" does not exist` means that the user is incorrect. If you used a different username, you will need to update your code to reflect this in the `sql.Open` function call.

- `panic: pq: database "lenslocked" does not exist` means that the database doesn't exist. This typically happens when the database name is incorrect. If you installed postgres on your own, you may need to create the database. If you are using `docker compose` this step should be handled for you, but it is worth verifying that you have the correct database name compared to what is in your `docker-compose.yml`. If you need to reset things with docker, you can run `docker compose down` followed by `docker compose up` to completely reset your containers.

Another possible source of an error is already having Postgres installed. In that case, the existing Postgres installation will very likely already be using port `5432`, so when we run `docker compose up` it won't be able to bind the docker version of Postgres to the `5432` port, as it is already in use.

One way to resolve this is to close Postgres, but restarting might lead to it starting back up again, so this is a short term solution. Postgres could also be uninstalled, but again this might not be an ideal solution. If neither of those options work, the last option is to use a different port for our docker Postgres. We do this by first opening up the `docker-compose.yml` file and updating the port we use there.

```
db: # The service will be named db.
  # ... don't change this stuff

  # Update the port on our machine (the first one) to be an unused port
  ports:
    - 5433:5432 # format here is "port on our machine":"port on container"
```

Then we update our `main()` function to use the new port.

```
func main() {
  // Change the port to 5433 or something else that isn't used
        db, err := sql.Open("pgx", "host=localhost port=5433 user=baloo password=junglebook dbn
  // ... unchanged code
}
```

If opting to go this route, remember to always use this new port (`5433` in the example) instead of what is used in future lessons.

Source Code

- Source Code - see the final source code for this lesson.
- Diff - see the changes made in this lesson.

10.2 Imports with Side Effects

In the previous lesson we saw the underscore (_) used to import a package.

```
import (
        _ "github.com/jackc/pgx/v4/stdlib"
)
```

This tells the compiler that we want to import the `pgx/v4/stdlib` package even though we don't appear to be using it for anything. In this lesson we are going to discuss both what that underscore is, and why we need an unused import in our code.

In Go, the underscore is called the blank identifier. It is used as a placeholder when we have a variable or a package that we do not actually use in our code, but we need it to be assigned to something for one reason or another. For instance, if we wanted to iterate over all of the values in an `[]int`, we could

use the blank identifier to ignore the index of each item in the slice while still utilizing the values in the slice.

```
numbers := []int{10, 20, 30}
// We don't care about the index of each number, so we use the _ for
// the first argument returned by range
for _, n := range numbers {
  fmt.Println(n) // prints 10, 20, 30
}
```

In most Go code we use the blank identifier when we don't care about a specific value being returned from a function or a **`for ... range`** loop. This is necessary because the Go compiler won't let us build any Go code that has unused variables.

In the case of the import, the blank identifier may feel a bit weirder at first glance. If we aren't using the imported package, why are we importing it at all? Unlike a **`for ... range`** loop or a function call with multiple return values, we don't appear to be using any values from the imported package at all.

The answer is the **`init`** function. When a package has an **`init`** function, the code inside that function gets run even if we never actually use the package. The code inside an init function also runs before our **`main`** function ever runs, allowing any init function declared in a package we import to run before our program starts.

In the case of **`pgx`**, and every other SQL driver to my knowledge, the **`init`** function in the package will set up a SQL driver and register it with the **`database/sql`** package. Below is a snippet of code from the **`pgx/v4/stdlib`** package that is used to register the driver.

```
// Don't code this. It is from the jackc/pgx/v4/stdlib package.
func init() {
        pgxDriver = &Driver{
                configs: make(map[string]*pgx.ConnConfig),
        }
```

```
        fakeTxConns = make(map[*pgx.Conn]*sql.Tx)
        sql.Register("pgx", pgxDriver) // <= This is the important line that registers the driv

  // ...
}
```

The whole process of code running inside of an init function and altering the state of our entire program is an example of a side effect. A side effect occurs when some code runs and alters state outside of its local environment. In our case, code inside the **`pgx/v4/stdlib`** package runs via an **`init`** function and then alters the global state within the **`database/sql`** package by calling **`sql.Register`**.

Generally speaking, side effects are frowned upon for a few reasons:

1. You cannot prevent the side effect from occurring. If you happen to be importing the same package for another reason, the driver will still be registered. This could be problematic if two drivers use the same name.
2. It feels like magic, which can be confusing to debug and follow.
3. Testing code with side effects can be challenging and finicky.

Knowing that side effects are generally frowned upon, this leads to the followup question of, "Why do SQL drivers utilize this technique if it is frowned upon?"

An arguably better approach to using **`init`** would be to provide a default driver and let developers register it. The resulting code could be as simple as the code below.

```
sql.Register("pgx", pgx.DefaultDriver())
```

Or if we wanted to avoid global state entirely, the **`sql`** package could be redesigned to use the driver when opening a connection to a database.

```
sql.Open(pgx.DefaultDriver, "host=localhost port=5432 ...")
```

The short answer to why these approaches aren't used is that the current pattern is a byproduct of legacy decisions both by the Go team and developers of third party SQL drivers. At some point in the past, SQL driver developers opted to register drivers this way. Perhaps they looked at the image/png package in the standard library and opted to adopt a similar strategy, or perhaps they came up with on their own. Regardless of how it came about, SQL drivers all tend to use this pattern because it provides a consistent developer experience, even if the actual practice might not be ideal in everyone's eyes.

I mention this because it is a good lesson to learn - sometimes writing code that remains consistent with legacy code is more important for maintainability and usability than writing what feels like a "better" version. By remaining consistent, we ensure that developers won't have to learn how each library works independently and can instead expect them all to behave in a similar manner.

10.3 Postgres Config Type

Our experimental code will successfully connect to our Postgres database, but there are a few issues with the code. For starters, our connection string is hardcoded, making it impossible to change if we need to connect to a different database.

```
sql.Open("pgx", "host=localhost port=5432 user=baloo password=junglebook dbname=lenslocked sslm
```

We also have data like the password hardcoded in this string, which can be a security concern. Anyone who has access to our source code now has access to our database password!

A better way to approach this is to create a config type that has everything we need to connect to a database, then we can update our code to accept a config value when it runs. This will allow us to connect to different databases depending on our needs, and it will also allow us to remove hardcoded passwords from our code in the future.

Open **cmd/exp/exp.go** and add the following code.

```
type PostgresConfig struct {
        Host     string
        Port     string
        User     string
        Password string
        Database string
        SSLMode  string
}

func (cfg PostgresConfig) String() string {
        return fmt.Sprintf("host=%s port=%s user=%s password=%s dbname=%s sslmode=%s", cfg.Host
}
```

The **PostgresConfig** struct type we created has all the fields we need to connect to Postgres. The host, port, username, password, etc. The **String()** method provides a way to turn it all of that data into a string that we can use with **sql.Open**.

To use this new type, we need to declare a **PostgresConfig** inside of our **main** function, then use it's **String** function when we call **sql.Open**.

```
func main() {
        cfg := PostgresConfig{
                Host:     "localhost",
                Port:     "5432",
                User:     "baloo",
                Password: "junglebook",
                Database: "lenslocked",
                SSLMode:  "disable",
        }
        db, err := sql.Open("pgx", cfg.String())
  // ... unchanged
}
```

That's it! We now have a Postgres config type that will make it much easier to create connection strings with different settings. Later in the course we will explore ways to read this config from somewhere else so that we don't have passwords hardcoded and are abel to make things more secure. This will become especially important when we deploy to a production server. For now this should be fine, just don't use any real passwords you don't want other developers on your team seeing.

Source Code

- Source Code - see the final source code for this lesson.
- Diff - see the changes made in this lesson.

10.4 Executing SQL with Go

There are three functions that can be used to execute SQL statements using the `database/sql` package:

1. Query
2. QueryRow
3. Exec -

While each of these functions can execute any SQL statements we provide to it, what they expect to be returned from the database and what they return to us vary between each, so it can be useful to understand how each function works.

`Query` and `QueryRow` are used when you want to query for records in the database. `Query` is used when we expect multiple records back, while `QueryRow` is used when we only want 1 record to be returned. For instance, if you wanted

to find a specific user with the email address "jon@calhoun.io" we would use `QueryRow`, but if we wanted to find all users whose email address uses the "calhoun.io" domain we would use `Query`, as we might find multiple users in our database that match the criteria.

In both cases it is possible for no records to be returned, but how these cases are handled differ between `Query` and `QueryRow`. When we receive no results from our SQL, `Query` will return us an empty set of records, however `QueryRow` is designed to expect exactly one record, so when there are no results returned it will return the sql.ErrNoRows error.

`Exec` is a little different from Query and QueryRow because it is typically used when no records are expected to be returned from the query. For instance, when creating a new table in our database we still want to execute some SQL code, but we don't really expect any records to be returned after the code is executed.

`Exec` has a return value with the type Result, but rather than records this contains metadata about the SQL statement that was just executed. Unfortunately, this metadata doesn't work with a Postgres database, so we won't be using the `Result` type in our code.

Aside 10.1. Reset your database before proceeding

If you have created any tables or written anything to your database you may need to reset things before proceeding. The simplest way to do this is to reset your docker containers by running the `down` and then the `up` commands inside docker compose.

```
docker compose down
docker compose up
```

This will tear down our containers, rebuild them, and finally start up our database container leaving us ready to proceed.

The first thing we are going to do with our Go code is execute SQL statements that create tables. This will enable us to insert and query for records as we proceed through the remainder of this section. In a future lesson we will look at how to manage our SQL database, but for nowe we will hardcode this SQL into our **cmd/exp/exp.go** source file. Open it up and add the following code to the end of the **main** function.

```
_, err = db.Exec(`CREATE TABLE IF NOT EXISTS users (
  id SERIAL PRIMARY KEY,
  name TEXT,
  email TEXT NOT NULL
);

CREATE TABLE IF NOT EXISTS orders (
  id SERIAL PRIMARY KEY,
  user_id INT NOT NULL,
  amount INT,
  description TEXT
);`)
if err != nil {
  panic(err)
}
fmt.Println("Tables created.")
```

The SQL here is creating two tables in our database. The first is a simplified table for users, and the second is a table for orders. We don't expect this operation to return any records, so we opt to use **db.Exec** to execute the SQL. When we begin creating records and querying for them we will see both **db.Query** and **db.QueryRow** in action.

Now let's explore the SQL that we are executing. At a high level, the code being run creates two tables - the **users** table, and the **orders** table.

```
-- Creates the users table if it doesn't exist in the DB
CREATE TABLE IF NOT EXISTS users ( ... );

-- Creates the orders table if it doesn't exist in the DB
CREATE TABLE IF NOT EXISTS orders ( ... );
```

Due to the **IF NOT EXISTS** clause, the two tables are only created if they

don't exist in the database. This allows us to run this code multiple times without any errors popping up. Without the `IF NOT EXISTS` part, running this code a second time would result in an error because we already have the `users` and `orders` tables in our database.

It is important to make sure your database was reset before running this code, otherwise you may have an incorrect users table already created and this code would skip creating the users table because one already exists.

Inside of each `CREATE TABLE` clause we define fields for each table. For the users, these fields are `id`, `name`, and `email`. For the orders table, these fields are `id`, `user_id`, `amount`, and `description`.

The two tables we created have a relationship between them. We want every order to be associated with the user who created the order, and we represent this with the `user_id` field in the orders table. With this we can set a user ID for every order that tells us which user created that specific order.

Aside 10.2. SQL Relationships

If you are new to SQL, one of the many things you will need to learn about are relationships. These are used to define links between various pieces of data that will make it easier for us to query the correct data in the future. For instance, in the example above we could easily find all of a specific user's orders by searching for all orders where the `user_id` field is set to a specific value.

While I will attempt to explain why we are creating relationships as we progress through the course, it is important to note that this is not a course dedicated to teaching SQL from the ground up and you may need to learn a bit more about the subject outside of this course. If you would like to learn more about relationships, there are many tutorials online such as this one: https://code.tutsplus.com/articles/sql-for-beginners-part-3-database-relationships–net-8561

Run the updated code to create the new tables.

```
go run cmd/exp/exp.go
```

Verify that you see the output "Tables created." after running the code and that no errors occurred.

Aside 10.3. Relation already exists

If we want to see what the error would look like for trying to create a table that already exists we can do so by removing the `IF NOT EXISTS` part of the SQL and runing the code multiple times. We will see an error similar to the one below.

```
panic: ERROR: relation "users" already exists (SQLSTATE 42P07)
```

When we see an error that says `relation "xxxx" already exists` it often means that we were attempting to create a table that already exists in the database.

If we want to be especially certain that the tables were created, we can navigate to

Source Code

- Source Code - see the final source code for this lesson.
- Diff - see the changes made in this lesson.

10.5 Inserting Records with Go

To insert a record using SQL, we use an **INSERT** statement. For instance, the following would create a user with the provided name and email address.

```
INSERT INTO users(name, email)
VALUES('Jon Calhoun', 'jon@calhoun.io');
```

To execute this with Go, we can use the **db.Exec** method and provide the exact same SQL.

```
_, err = db.Exec(`
  INSERT INTO users(name, email)
  VALUES('Jon Calhoun', 'demo@user.com');
`)
if err != nil {
  panic(err)
}
```

Coding the name and email address directly into our SQL statement doesn't provide us with a lot of flexibility. After all, we are probably going to want to add users to our app that have a different name and email address! What we need is a way to provide dynamic data to our SQL statements using variables. The following code demonstrates how to do this with Go and Postgres.

```
name := "Jon Calhoun"
email := "jon@calhoun.io"
_, err = db.Exec(`
  INSERT INTO users(name, email)
  VALUES($1, $2);`, name, email)
if err != nil {
  panic(err)
}
fmt.Println("User created.")
```

In this code the **name** variable will be used anywhere **$1** is seen inside the SQL statement, and the value stored in the **email** variable will be used anywhere

we see `$2` in the query. This allows us to create dynamic SQL statements using data stored in our Go variables. It also means we could put a `$1` in multiple places and the `name` variable would be used in all of those instances.

Every SQL database uses slightly different placeholders for constructing dynamic queries with Go. For Postgres, the placeholders are a dollar sign followed by a number - `$1`, `$2`, etc. The number references the order that the variable was passed into the `Exec`, `Query`, or `QueryRow` functions. In our case `name` was the first variable, so it is referenced as `$1`, and `email` was the second variable, so it is referred to as `$2`. These values are 1-based, so there is no `$0` placeholder.

If we were using MySQL, the placeholders would instead be question marks (`?`), and our code would instead look like the following.

```
name := "Jon Calhoun"
email := "jon@calhoun.io"
_, err = db.Exec(`
  INSERT INTO users(name, email)
  VALUES(?, ?);`, name, email)
if err != nil {
  panic(err)
}
```

Other SQL databases might have their own unique placeholder system, so keep this in mind if using anything other than Postgres with the code in this course.

Aside 10.4. Warning: Do NOT build queries on your own!

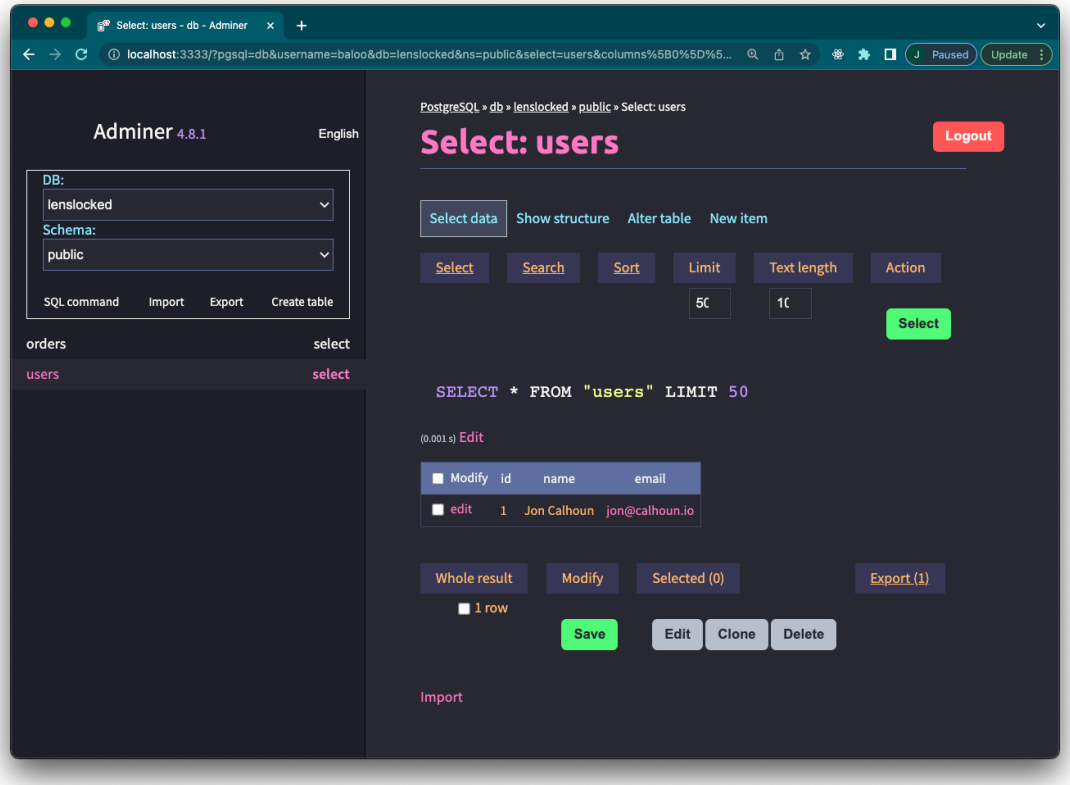
It is **extremely important** that we do not try to build the SQL query string on our own. Doing so could result in an attack known as SQL injection, which we will explore in the next lesson.

Now that we understand how to create a dynamic query, let's add the code to

our application and try it out. Open **cmd/exp/exp.go** and add the following code to the end of the **main** function.

```
name := "Jon Calhoun"
email := "jon@calhoun.io"
_, err = db.Exec(`
INSERT INTO users(name, email)
VALUES($1, $2);`, name, email)
if err != nil {
  panic(err)
}
fmt.Println("User created.")
```

Run the updated code to create a record in the database. After that open Adminer (http://localhost:3333) and verify that the record was created. This can be done by clicking the "select" link next to the "users" table on the far left navigation bar.



Alternatively, we could use docker exec to run `psql` in our docker container and then run a `SELECT` statement.

```
docker exec -it lenslocked-db-1 /usr/bin/psql -U baloo -d lenslocked
```

Then the select statement:

```
SELECT * FROM users;
```

And finally, the output:

```
 id |    name     |     email
----+-------------+---------------
  1 | Jon Calhoun | jon@calhoun.io
(1 row)
```

Source Code

- Source Code - see the final source code for this lesson.
- Diff - see the changes made in this lesson.

10.6 SQL Injection

We learned a bit about code injection earlier in the course when learning about HTML templates; SQL injection is another form of code injection where SQL code written by an attacker is executed on the target's database. For instance, an attacker might try to get our SQL database to run a query that deletes all of the users. Even worse, an SQL injection could lead to sensitive user data being leaked.

In this lesson we are going to intentionally write code that is vulnerable to SQL injection to help illustrate both what SQL injection is, as well as how important it is to avoid. We will then look at how the `database/sql` package helps us prevent this, and why we use the `$1` placeholders to insert variable data.

In the last lesson we wrote the following Go code to create a new user.

```
_, err = db.Exec(`
INSERT INTO users(name, email)
VALUES($1, $2);`, name, email)
```

Rather than using the `database/sql` package to construct this query using variables, imagine that we attempted to construct the SQL on our own. We

might use the **fmt.Sprintf** function to add data to the string, giving us code similar to that below.

```
name := "Jon Calhoun"
email := "jon@calhoun.io"
query := fmt.Sprintf(`
  INSERT INTO users (name, email)
  VALUES ('%s', '%s');`, name, email)
_, err = db.Exec(query)
if err != nil {
  panic(err)
}
```

At first glance, this code looks perfectly safe, but what happens if someone signs up using the following name?

```
name := "','); DROP TABLE users; --"
```

Update the code at the end of **cmd/exp/exp.go** to use the following code to insert a user, then run the code.

```
name := "','); DROP TABLE users; --"
email := "jon@calhoun.io"
query := fmt.Sprintf(`
  INSERT INTO users (name, email)
  VALUES ('%s', '%s');`, name, email)
_, err = db.Exec(query)
if err != nil {
  panic(err)
}
fmt.Println("User created.")
```

After running the code it appears to execute without an errors, but let's take a closer look. Use Docker to run **psql** on the postgres container.

```
docker exec -it lenslocked-db-1 /usr/bin/psql -U baloo -d lenslocked
```

After opening `psql`, execute the following query to see what users are in our database.

```
SELECT * FROM users;
```

After executing the SQL, we now see the following error!

```
ERROR:  relation "users" does not exist
```

This error is occurring because we no longer have a users table! We can verify this in Adminer as well. **What happened?**

Let's debug a bit by printing out the SQL statement we are executing. Add the following code just before teh `db.Exec` in your `main` function.

```
fmt.Printf("Executing: %s\n", query)
```

This will print out the query we are about to executing, making it easier to tell exactly what is going on. The following SQL will be printed in the terminal if we run this code again.

```
INSERT INTO users (name, email)
VALUES ('','); DROP TABLE users; --', 'jon@calhoun.io');
```

At first glance this might seem okay, especially in our terminal where there isn't any syntax highlighting, but if we add syntax highlighting we notice that everything after the `--` characters is different color. This is because the `--` character in SQL is the comment character!

When we used our custom SQL creation code along with the new name value, it didn't create the query we expected at all. Instead, it uses empty strings for both the name and email address and ends the INSERT statement with a semicolon. After that it runs an additional SQL statement to drop the users table, followed by the comment characters to tell SQL to ignore any other text on the line. By executing this SQL, we may create a user with no name or email address, but more importantly it will go on to delete all of our users afterwards.

If we look back at the `name` variable we constructed, this particular one was designed specifically to cause issues with our database. Is it realistic that an attacker could actually construct a string like this?

One of the ways attackers try to execute a SQL injection is to submit thousands of special strings into every HTML forms on a website. This is done using a program (or custom code) in the hope that hoping that one or more of the forms ends up using the value without escaping it, and the escaped value leads to a SQL injection. As a result, attackers do not need to get the injection string right on the first or even one hundredth attempt. They only need it to work once to cause damage to the database.

How do we prevent SQL injection?

Rather than creating queries on our own, we instead use libraries that escape variables for us. In this case, the `database/sql` package along with the use of variable placeholders (`$1`, `$2`, and so on) will handle all the work for us! We can verify this by updating our code to use the same `name` variable, but instead of constructing the SQL ourselves we can use the `sql` package to do it for us.

```
name := "','');  DROP TABLE users; --"
email := "jon@calhoun.io"
_, err = db.Exec(`
  INSERT INTO users (name, email)
  VALUES ($1, $2);`, name, email)
if err != nil {
  panic(err)
}
fmt.Println("User created.")
```

After executing this code we can now check to see if we have a users table, and if it has any entries.

```
SELECT * FROM users;
```

```
 id |             name              |     email
----+-------------------------------+----------------
  1 | ','); DROP TABLE users; -- | jon@calhoun.io
(1 row)
```

We have successfully created a user with an odd name while avoiding any SQL injection!

Regardless of how much validation you think your application does, be sure to always use a package like `database/sql` to ensure you are not susceptible to SQL injection.

Source Code

- Source Code - see the final source code for this lesson.
- Diff - see the changes made in this lesson.

10.7 Acquire a new Record's ID

When inserting new records into a database, it is common to want the database to return some information about that record. For instance, if we created a new order in a database table, we might want to know the ID that the database generated for that order so we can redirect the user to view the completed order in their browser. Since we leave ID generation up to the database, we need to ask the database to return the ID to us after we create an order.

When we use the **Exec** function provided by the **database/sql** package, we get an sql.Result and an error object as the return values. If we examine the **Result** type it looks like we could use the **LastInsertId** method to determine the ID of a newly created record, and that is the intent of the method, but unfortunately it does not work with Postgres databases. There are also cases where it won't work with any database, such as databases tables that use a custom ID type, such as a UUID.

In this lesson we are going to look at how to acquire a new records ID after creating it. We will update our code so that we can both create the new record and acquire its ID all in a single SQL statement. We do this by modifying our SQL query to tell our database to not only insert a record, but to also return some data using the RETURNING clause. Below is an updated example of the SQL that tells the database to return the ID.

```
INSERT INTO users (name, email)
VALUES ($1, $2) RETURNING id;
```

We are requesting data be returned from the database, so we can no longer use **DB.Exec** to execute this statement. Instead, we need to use DB.QueryRow(). **QueryRow** is most commonly thought of when querying for a single record in a SQL database, but it can be used to executing any SQL statement we want, including an **INSERT** that returns an ID. Let's update our code in **main/exp/exp.go** to use the new SQL statement as well as **QueryRow**.

```
name := "New User"
email := "new@calhoun.io"
row := db.QueryRow(`
  INSERT INTO users (name, email)
  VALUES ($1, $2) RETURNING id;`, name, email)
var id int
err = row.Scan(&id)
if err != nil {
  panic(err)
}
fmt.Println("User created. id =", id)
```

The first change is the `RETURNING id` which tells our query to return the ID after it finishes creating the record. We discussed this earlier when looking at the updates to the SQL statement.

The next change is the return value. `QueryRow` returns a `row` and no error. We won't see the error until we try to call `Scan` on the row.

`Scan` allows us to pass in a pointer to a variable that we want the data to be inserted into. After executing, the `id` variable will then had the value returned by the `RETURNING id` part of the SQL query.

Aside 10.5. Scan and pointer variables

When we call `row.Scan(&id)` we are passing in a pointer to our `id` variable. What this does is tell the `Scan` method the memory address of our variable, allowing it to set a new value in that location in memory. This is similar to handing someone your phone and asking them to type in their phone number.

With dynamic programming languages like Python or JavaScript, it is possible to dynamically determine what to return from a function. For example, the following Python function will return an integer if `10` is passed in, otherwise it will return a string and an integer.

```
def demo(x):
  if x == 10:
    return 10*10
  return "Hello, " + x, 100

print(demo(10)) # prints 100
print(demo("jon")) # prints ('Hello, jon', 100)
```

With static languages like Go, every function needs a static return value. A function must return either an integer, or a string and an integer, and it cannot switch back and forth between the two.

```
func demo(x int) int { ... }
// or
func demo(x string) (string, int) { ... }
// but not both!
```

This restriction makes it easy to know exactly what data is being returned from a function at compile time, but it also limits our flexibility. When we call `row.Scan` it cannot just return data to us because the number of variables and the data types could be different for every SQL query. Meanwhile, a language like Python could do this.

To get around this limitation while still being type safe, functions like `Scan` allow us to pass in pointers to the variables we want to scan data into. This allows us to call `Scan(&id)` one time, and `Scan(&name, &id)` another time, all while using typed variables. This technique is commonly found in libraries where data needs to be decoded from one format and then read into Go variables. Another example is the `encoding/json` package which uses it to read data from JSON and move it into Go variables.

When using `QueryRow` it is expected that data will be scanned, so it simply returns a sql.Row. If we wanted, we could immediately check for an error by calling row.Err, but if we read the docs we will notice that any errors returned by `row.Err` will also be included when we call row.Scan. This enables us to immediately proceed with the line `err = row.Scan(&id)` which will give us any errors from the initial SQL query or any errors that occurred while attempting to scan the data into our Go variables.

Finally, our code prints out the new user's ID showing that it was correctly scanned.

If we run this code we should see output similar to that below. The ID may be different in your code.

```
Connected!
Tables created.
User created. id =  2
```

We can also open up Adminer or use `psql` via Docker to verify that users are being created and have the IDs being returned.

Email addresses do not currently need to be unique, so you should be able to execute this multiple times. We will address that issue later when done learning about SQL with Go.

Source Code

- Source Code - see the final source code for this lesson.
- Diff - see the changes made in this lesson.

10.8 Querying a Single Record

Now that we have a few records created, we are going to first comment out the code used to create new users. If at any time you need to create additional users, feel free to uncomment the code. Open `cmd/exp/exp.go` and comment the following.

```
// name := "New User"
// email := "new@calhoun.io"
// row := db.QueryRow(`
//   INSERT INTO users (name, email)
//   VALUES ($1, $2) RETURNING id;`, name, email)
// var id int
// err = row.Scan(&id)
// if err != nil {
//         panic(err)
// }
// fmt.Println("User created. id =", id)
```

Below the commented code we are going to add code that helps us see how to query for a single record in Go. To do this, we will use the `QueryRow` function we saw in previous lessons along with a `SELECT` SQL query. We will likely also want to use a `WHERE` clause to specify which particular record we are interested in, otherwise we may receive multiple records and `QueryRow` will only give us the first record found.

We are going use an SQL statement similar to the following to query for a user.

```
SELECT name, email
  FROM users
  WHERE id=1;
```

This SQL is stating that we want a user with an `id` that matches to exactly `1`. A user with the id of `12` will not match. This is a common way to lookup a specific record in a database, and since the `id` is a primary key, we know that no two records will have the same `id` and this will return at most one result.

Rather than using `SELECT *` and getting all of the data back, we are explicitly telling our database which fields we want and in what order we want them. This will make it easier for us to scan the data into Go variable, as we will know exactly what to expect and in what order without needing to reference the SQL table in our database.

Aside 10.6. Use an appropriate ID

We are going to learn how to query for a single record by querying for a user with a specific ID. I will be using the id `1` in the examples, but be sure to use an ID that you know exists in your database otherwise you will likely encounter errors as you run the code. This can be done using either Adminer or psql and a `SELECT * FROM users;` statement.

To use this SQL in our Go code, open **cmd/exp/exp.go** and add the following code at the end.

```
id := 1
row := db.QueryRow(`
  SELECT name, email
  FROM users
  WHERE id=$1;`, id)
var name, email string
err = row.Scan(&name, &email)
if err != nil {
  panic(err)
}
fmt.Printf("User information: name=%s, email=%s\n", name, email)
```

We are using the **$1** placeholder like before to make sure we don't have to worry about SQL injection attacks. In this case it is being replaced by a number so SQL injection is unlikely, but it is good to stay in the habit of always using the **database/sql** package to build your queries so that you don't leave yourself susceptible to an attack.

As mentioned earlier, we explicitly wrote **SELECT name, email** in our query, so when we call **Scan** we need to account for those two fields. We do this by declaring the two variables, and then passing pointers to them into the **Scan** function.

```
var name, email string
err = row.Scan(&name, &email)
```

If we had used **SELECT *** we wouldn't know exactly what data we are going to get from this query, nor would we know what order it would be in without first looking at the SQL table's structure. It is also possible that a change to the SQL table - like the addition of a new column of data - could break our code because our query would start returning more data than we pass pointers into **Scan** for. This is why it is a good idea to explicitly name the fields you want returned when writing select statements in your Go code.

If you get the error **`sql: no rows in result set`** it means that your query isn't returning any records. As a result, you will get an error when you try to call the **`Scan()`** method on the returned **`sql.Row`** because it isn't possible to scan a row that wasn't returned.

The "no rows ..." message comes from the ErrNoRows error variable that is exported from the **`database/sql`** package. When calling the **`QueryRow`** function, this error is used if no records are found that matched the query. Not finding any results is a very different error from having a database connection issue, so it is often worth looking at the error to determine how to proceed.

We will use this later in the course when doing things like looking up users via the email address. For now, update your code to add the following check to see this in action. Be sure to use an **`id`** that doesn't exist in your database so you can see the **`Error, no rows!`** message being output before the code panics.

```
id := 200 // Choose an ID NOT in your users table
row := db.QueryRow(`
SELECT name, email
FROM users
WHERE id=$1;`, id)
var name, email string
err = row.Scan(&name, &email)
if err == sql.ErrNoRows {
  fmt.Println("Error, no rows!")
}
if err != nil {
  panic(err)
}
fmt.Printf("User information: name=%s, email=%s\n", name, email)
```

Running this should give output similar to that below.

```
Connected!
Tables created.
Error, no rows!
panic: sql: no rows in result set
```

Source Code

- Source Code - see the final source code for this lesson.
- Diff - see the changes made in this lesson.

10.9 Creating Sample Orders

There are often cases where you need to query for multiple records in a database rather a single record. For instance, when a user logs into their account we may want to show their entire order history, not just the most recent order. Rendering this would require us to query for multiple orders associated with the user.

In the next lesson we are going to look at how to write queries for multiple records, but first we need to create some records to query. Our database currently has a users table and an orders table, so we will be creating some fake orders associated to a user so we have data to query.

Head to Adminer, or use **`docker exec`** to run **`psql`**, and jot down the ID of one of the users in your database. If a user doesn't exist, use code from the previous lessons to create one.

Head back over to **`cmd/exp/exp.go`** and add the user ID to the bottom of the **`main`** function.

```
userID := 1 // Pick an ID that exists in your DB
```

Next we are going to create fake orders for this user. We will be using techniques from previous lessons to insert each records, but we will be using fake values and a **`for`** loop to do this multiple times.

```
for i := 1; i <= 5; i++ {
  amount := i*100
  desc := fmt.Sprintf("Fake order #%d", i)
```

```
  _, err := db.Exec(`
    INSERT INTO orders(user_id, amount, description)
    VALUES($1, $2, $3)`, userID, amount, desc)
  if err != nil {
    panic(err)
  }
}
fmt.Println("Created fake orders.")
```

Next, comment out the code that was used in the previous lesson to query for a non-existing user.

```
// id := 2 // Choose an ID NOT in your users table
// row := db.QueryRow(`
// SELECT name, email
// FROM users
// WHERE id=$1;`, id)
// var name, email string
// err = row.Scan(&name, &email)
// if err == sql.ErrNoRows {
//         fmt.Println("Error, no rows!")
// }
// if err != nil {
//         panic(err)
// }
// fmt.Printf("User information: name=%s, email=%s\n", name, email)
```

Finally, proceed with running the updated code and check to verify that several orders were created. We can do this by using Adminer, or running the following SQL inside of psql.

```
SELECT * FROM orders;
```

If everything was successful, we should see at least five new orders all associated with the same user.

```
 id | user_id | amount |  description
----+---------+--------+---------------
  1 |       1 |    100 | Fake order #1
  2 |       1 |    200 | Fake order #2
  3 |       1 |    300 | Fake order #3
  4 |       1 |    400 | Fake order #4
  5 |       1 |    500 | Fake order #5
(5 rows)
```

We are now prepared to query for multiple records with our Go code in the next lesson!

Source Code

- Source Code - see the final source code for this lesson.
- Diff - see the changes made in this lesson.

10.10 Querying Multiple Records

The last bit of coding we will be doing while learning to use Postgres with Go is to query for multiple orders. Open **`cmd/exp/exp.go`** and we can first start by commenting out old code used to create fake orders. We should comment it out rather than deleting it so we can easily create new orders if we need as we continue learning.

```
// userID := 1
// for i := 1; i <= 5; i++ {
//         amount := i * 100
//         desc := fmt.Sprintf("Fake order #%d", i)
//         _, err := db.Exec(`
//     INSERT INTO orders(user_id, amount, description)
//     VALUES($1, $2, $3)`, userID, amount, desc)
//         if err != nil {
//                 panic(err)
//         }
// }
// fmt.Println("Created fake orders.")
```

Next, we can create an **Order** type that we will use to store information about each order.

```
type Order struct {
  ID int
  UserID int
  Amount int
  Description string
}
```

Once we have a type for our orders we can create a variable that is a slice of orders. This will give us a place to store the orders as we read them from the database.

```
var orders []Order
```

Next, add the code to query for multiple orders using **Query** and a **SELECT** statement to **exp.go** inside the **main** function. We will discuss what the code does shortly.

```
userID := 1 // Use the same ID you used in the previous lesson
rows, err := db.Query(`
  SELECT id, amount, description
  FROM orders
  WHERE user_id=$1`, userID)
if err != nil {
  panic(err)
}
defer rows.Close()
```

In our new code the **db.Query** function call is used to run our SQL query because we want to receive multiple records back from our database. While **Query** and **QueryRow** work similar in some ways, they are also unique in several ways which we need to discuss.

The first difference is the return values. Query returns two things:

- A `*sql.Rows` (*Note: This is plural, not a singular row like* `QueryRow` *returns.*)
- An error

The error returned by `Query` can be used to verify that our SQL query ran without any issues. For example, if we mistyped the table name we might get an error like `pq: relation "uzers" does not exist`. We need to check this error before proceeding, as the `*sql.Rows` may be nil if we do have an error.

Aside 10.7. Why does Query return an error while QueryRow does not?

When we used `QueryRow` we didn't receive an error as a return value, and we didn't need to check for any errors until we called `Scan` on the row being returned.

```
// QueryRow doesn't return an error
row := db.QueryRow(`
  SELECT name, email
  FROM users
  WHERE id=$1;`, id)
var name, email string
err = row.Scan(&name, &email)
if err != nil {
  panic(err)
}
```

`Query` returns an error as a second return value, and we need to check it immediately before proceeding. Why do these two functions work so differently?

Truthfully, I don't know the entire reason for this. It is possible it was a design decision that would be changed if not for the Go 1.0 backwards compatibility promise. It is also possible that `Query` operates this way because creating the `*sql.Rows` was less performant and by returning an error a `nil` value can be

returned for the `Rows` in those cases. All I can say is that these two functions operate a bit differently, so it is worth noting those differences and keeping them in mind when writing our code.

Once we know there aren't any errors, we are ready to proceed with the results from our query. Unlike a single row from `QueryRow`, we could get back hundreds of rows of data with a call to `Query`. We might also stop reading data from the query before reaching the end of those results if we happen to find what we need. As a result, `*sql.Rows` has a `Close` method that allows us to close it and tell the garbage collector that we no longer need the `rows` variable, even if we didn't read all the SQL records from it. The simplest and safest way to ensure the `rows` variable is always closed is to use a defer statement immediately after creating it.

```
defer rows.Close()
```

While we could technically just write `rows.Close()` whenever we are done with the `rows` object, it is safer to use a defer statement immediately after verifying there are no errors so we don't forget about closing the rows. This makes it easier to accidentally forget to close the rows in the case of an error or any other branching logic, and it makes it easier for future developers to immediately see that you did remember to close the rows.

It is also important that the defer comes *after* we check for an error, as the `rows` object may be `nil` if there is an error and it calling `Close` on a nil value will result in a runtime error.

At this point we have written our query, checked for errors, and added some cleanup code to ensure the `rows` variable gets closed. We are now ready to iterate over the SQL records returned from our `SELECT` statement. This is done using a `for` loop and the `Next` method provided by `rows`. Add the following code to `exp.go` after the query.

```
for rows.Next() {
  var order Order
  order.UserID = userID
  err := rows.Scan(&order.ID, &order.Amount, &order.Description)
  if err != nil {
    panic(err)
  }
  orders = append(orders, order)
}
err = rows.Err()
if err != nil {
  panic(err)
}
```

There is a lot going on here, so we need to step through a few things. The first is the outer for loop and the subsequent error check.

```
for rows.Next() {
  // ...
}
err = rows.Err()
if err != nil {
  panic(err)
}
```

When we call **`rows.Next()`** it will attempt to prepare the next SQL record returned from our query, and it will then either return **`true`** or **`false`**. If it returns true, that means the next record from our SQL query was successfully prepared and we can read it from the **`rows`** object. If **`Next()`** returns false, it means that we either encountered an error, or that there are no more records left to read.

The designers of the **`database/sql`** package intentionally designed **`Next`** to only return a boolean so that it is easier to use in a **`for`** loop like we are doing, but this also means we need to check for errors immediately after the for loop. We do this with the **`rows.Err()`** method as we see in our code.

Another common question at this point is, "Aren't we skipping the first record if we call **`rows.Next()`** as part of the **`for`** loop?"

While it does look like we are skipping the first record, the `sql.Rows` object was designed to work this way. In fact, you cannot read from a `sql.Rows` object without first calling `rows.Next()`, so even if we didn't use a for loop we would need to call `rows.Next()` before trying to call `rows.Scan`.

Needing to call `rows.Next()` may feel odd, but consider the code required to write a for loop if that were not the case:

```
// This isn't correct. It is shown for illustrative purposes.
for {
  //
  // ... Read each row
  //
  if !rows.Next() {
    break
  }
}
```

It is much harder to read and manage code like this, as it is not clear how the for loop terminates without digging through the code in more detail looking for a `break` statement.

We are now prepared to examine the code inside of the for loop. This is the code that gets executed for every record returned by our SQL query.

```
var order Order
order.UserID = userID
err := rows.Scan(&order.ID, &order.Amount, &order.Description)
if err != nil {
  panic(err)
}
orders = append(orders, order)
```

Inside the for loop we first create a variable with the `Order` type. This gives us a place to store the data we are about to scan. We then proceed to read each record in a manner very similar to how we read a single record in previous lessons, calling `Scan` and checking for an error.

```
err := rows.Scan(&order.ID, &order.Amount, &order.Description)
if err != nil {
  panic(err)
}
```

Finally, we append the new order to the **orders** slice we created earlier. This places every order we read from the database into the **orders** slice so that we can use it at a later time in our code.

Finally, we can print out our orders to make sure everything is working as intended. For now, let's use a simple **Println** and let Go use its default slice formatting.

```
fmt.Println("Orders:", orders)
```

Putting this all together, below is the complete code we added to the **main** function in **exp.go**:

```
type Order struct {
  ID          int
  UserID      int
  Amount      int
  Description string
}
var orders []Order

userID := 1 // Use the same ID you used in the previous lesson
rows, err := db.Query(`
SELECT id, amount, description
FROM orders
WHERE user_id=$1`, userID)
if err != nil {
  panic(err)
}
defer rows.Close()
for rows.Next() {
  var order Order
  order.UserID = userID
  err := rows.Scan(&order.ID, &order.Amount, &order.Description)
  if err != nil {
```

```
    panic(err)
  }
  orders = append(orders, order)
}
err = rows.Err()
if err != nil {
  panic(err)
}
fmt.Println("Orders:", orders)
```

After running the code, we will see output similar to the output below.

```
Orders: [
  {1 1 100 Fake order #1}
  {2 1 200 Fake order #2}
  {3 1 300 Fake order #3}
  {4 1 400 Fake order #4}
  {5 1 500 Fake order #5}]
```

I have broken the output into a few lines to make it easier to read, but your code should output something similar all on a single line. The output we are looking at is the slice of orders we read from our SQL database. The first number in each object is the ID of the order, followed by the user ID, the amount, and finally the order description.

Source Code

- Source Code - see the final source code for this lesson.
- Diff - see the changes made in this lesson.

10.11 ORMs vs SQL

Note: You don't need to code anything from this lesson. All of the code is for illustrative purposes only.

At this point I suspect many people will want to ask questions about ORMs. Are they good or bad? What are the benefits? Any libraries you suggest? Why don't we use ORMs in this course?

There are a wide variety of ways to interact with databases in any programming language. Typically, these all fall somewhere on a spectrum between pure SQL and an ORM.

ORM stands for object-relational mapping and is the opposite of what we have been doing so far. Our Go code has been as close to the pure SQL side as we can get.

At this time, the SQL is our source of truth for the database. When we want to create a database table, we write a SQL statement. When we want to query code, we write a SQL statement. We use Go code to help execute the queries and retrieve data, but the Go code doesn't change much and the SQL is what really matters and, as a result, we need to have a reasonable understanding of SQL.

Using an ORM requires an opposite approach. Rather than using SQL to define our database tables, we write Go code to define the tables and that is eventually translated into SQL. When we want to write queries, we write Go code that ultimately gets translated into SQL queries, but we are focused on the Go code. We might not even know the exact SQL that will be executed and in many cases it might not even matter to us. Using this approach doesn't require as deep an understanding of SQL and can be easier to get started with when new to SQL or web development.

Let's look at an example. We currently declare our tables with the following SQL:

```
CREATE TABLE IF NOT EXISTS users (
  id SERIAL PRIMARY KEY,
  name TEXT,
  email TEXT UNIQUE NOT NULL
);

CREATE TABLE IF NOT EXISTS orders (
```

```
  id SERIAL PRIMARY KEY,
  user_id INT NOT NULL,
  amount INT,
  description TEXT
);
```

In an ORM like GORM we would instead declare Go models and use those as the source of truth that construct our SQL tables.

```
// User has many Orders, UserID is the foreign key
type User struct {
  ID     uint    `gorm:"primaryKey"`
  Name   string
  Email  string
  Orders []Order
}

type Order struct {
  ID          uint    `gorm:"primaryKey"`
  UserID      uint
  Amount      int
  Description string
}

// This would create both the users and the orders
// tables in our database similar to how our
// CREATE TABLE users ... SQL worked.
db.AutoMigrate(&User{}, &Order{})
```

For a Go developer, this Go code might look a lot easier to read and understand than the SQL. This trend carries on when we start to create new users, query for orders, and more. All of the SQL details are all handled by the ORM and we don't write much SQL.

```
// This creates a user in the database
user := User{
  Name: "Jon Calhoun",
  Email: "jon@calhoun.io",
}
db.Create(&user)

// Find a user with the ID 1
```

```
var user User
db.First(&user, "id = ?", 1)

// Query for all users
var users []User
db.Find(&users)

// Query for a user's orders
userID := 1
var orders []Orders
db.Find(&orders, "user_id = ?", userID)
```

Behind the scenes an ORM like GORM is translating this all into SQL and executing SQL queries, but from our end we look to be writing Go code!

There are many pros and cons to consider.

- **Pro**: It is easier for someone new to use an ORM. We just spent two sections with many lessons learning SQL which can be overwhelming when this is all new, and we only scratched the surface of SQL! With an ORM this learning can be delayed until it is needed rather than needing to be done upfront.

- **Con**: Each ORM is unique and needs to be learned. Once you know SQL, you can apply 99% of that knowledge anywhere, even if you end up writing code in Python or Rust. Once you learn an ORM you only know that specific ORM.

- **Pro**: In many ORMs you can still write SQL, so for really complex queries you can utilize pure SQL as needed.

- **Con**: It is best to learn and understand SQL eventually, otherwise you may end up with slow queries and performance issues. As a result, learning an ORM only delays the inevitable in many cases. Granted, this delay can be beneficial since it allows us to focus on learning fewer things at a time.

No matter what approach you opt for, there are going to be pros and cons. One isn't strictly better and both have their place. Regardless of which approach you use, it is advisable to spend some time learning SQL basics. You don't need to become a master, but a basic understanding can go a long way.

This course covers some of the basics, but I would definitely encourage you to continue learning more about SQL after completing the course.

In this course we are going to use the `database/sql` package. Not because I think it is the best option in the Go ecosystem - I actually think `GORM` is a better fit for this course - but because `database/sql` should be the most universal option with the least chance to break.

Anyone with Go has the `database/sql` library installed and can use it. All of our SQL will work years from now; an ORM on the other hand might update and introduce breaking changes.

Aside 10.8. Story time

The second point here - that ORMs can introduce breaking changes - isn't hypothetical. In the past this course used GORM and the library introduced a breaking change. It was very easy for students to fix, but it still caused issues and wasn't ideal. Pure SQL uses mostly the standard library, so the odds of this type of breaking change happening are much smaller.

As we progress through the course the SQL we have learned so far should cover most of our needs, and I'll try to explain what any future SQL we write is doing as a refresher, but if you are ever feeling overwhelmed remember than ORMs are also an option to consider.

10.12 Exercises

10.12.1 Ex1 - Create new SQL tables and Go code to work with them

If you are interested in extra practice using SQL with Go, take some time to create an entirely new set of SQL tables using the code we wrote in this section. A common way to do this is to pick an app you know well - like Twitter - and try to replicate a few of the relationships you imagine might exist there. For instance, we might have the following:

- A users table
- A tweets table, with a relationship to users
- A likes table with a relationship between both tweets and users

You could even expand this to have replies, which might be created using tweets with a parent tweet ID. This would have a value if a tweet is a reply or a message in a thread, but would be null if the tweet is not part of a thread or reply.

The goal here isn't to get everything perfect, but to experiment and a bit to both reinforce what you know and to push yourself beyond your comfort zone so you learn a bit more about SQL.

With all of these tables, try to write some Go code to perform inserts and other queries like we did in `cmd/exp/exp.go`.

10.13 Syncing the Book and Screencasts Source Code

In the screencasts we used the path `cmd/exp/main.go` for our experimental code, meanwhile we used `cmd/exp/exp.go` in the written version of the course. I've seen both used in practice and opted to use `exp.go` because running `go build` with this filename results in a binary named `exp` (at least on Mac OS), meanwhile using `main.go` builds a binary named `main` which is vague.

To sync the two up, this screencasts has us renaming the file to `exp.go`, but no changes are needed for anyone who has been following the written version of the course and already uses `exp.go`.

It is also possible that the source code for the written course doesn't have all the markdown files in the `notes` directory. This is intentional, as the notes didn't seem as necessary when the written course says the same thing but with more clarity.

Chapter 11

Securing Passwords

11.1 Steps for Securing Passwords

Now that we have spent some time learning about SQL we are eager and ready to start adding users to our application, but unfortunately we aren't quite ready to do that.

If we look at what information we need for a user, we will find that users are only composed of a few pieces of information:

- An ID
- An email address
- A password

Only three columns of data, simple, right?

Unfortunately, users are far from simple due to the password field. Unlike most data in our database, passwords cannot simply be stored as text in our database. If we were to do this, a data breach could result in every user's password being

leaked. Not only would this cause trouble with our own app, but many users share passwords between websites and this could lead to their accounts being compromised in other applications as well. As a result, our authentication setup is arguably the most important and sensitive part of our application.

While this might seem scary at first, the truth is that implementing a secure authentication service isn't incredibly challenging, but it does require us to follow a series of industry standards and to **not deviate from those standards at all**. In many cases, security issues stem from developers with good intentions attempting to do something custom and inadvertently introducing a security issue. We don't want to see our website on the front page of the New York Times with a "Data Breach" headline, so let's stick to the industry standards.

Throughout the rest of this section we will go into many of these in detail, but for now let's take a high level look at the steps to securely handling passwords:

1. Use HTTPs to secure our domain.
2. Store hashed passwords. Never store encrypted or plaintext passwords.
3. Add a salt to passwords before hashing.
4. Using time-constant functions during authentication.

The first one is pretty straightforward; when we deploy our application, we should get an SSL certificate and ensure that all traffic uses URLs with the `https` prefix. This ensures that all communications with our web server is encrypted, making it nearly impossible for someone to intercept the password when a user submits a form or does something similar.

We won't actually look at setting up an SSL certificate until later in the course when we deploy to production, but it is worth noting here as a requirement for securely handling passwords.

The next requirement will be covered in great detail, but it can never stated enough. NEVER under any circumstances store an encrypted or plaintext pass-

word. Always store a hashed value instead. If you are ever unsure of whether something is a hash, encryption, or something similar, the simple rule is that if you can calculate the password from whatever you are storing, it is NOT a hash. We will go into this in detail in the next lesson as we learn what hash functions are.

Third, passwords should always use a salt. This is done to prevent rainbow tables from being effective. We will explore this in detail in a future lesson.

The final one is a bit trickier to explain and understand, but hackers can actually use obscure information like how long it takes our server to check a password to try to determine which hashing function we are using. To prevent this, we will learn how to use time-constant functions when authenticating users.

11.2 Third Party Authentication Options

A fairly common question at this point is, "Why don't we just use another package or service that does all this for us?"

For instance, frameworks like Ruby on Rails have libraries like devise that provide almost all of the authentication out of the box, and paid services like Auth0 handle all of the password details for us.

While options like these can be useful, without truly understanding what goes into a secure authentication system it can be easy to unwittingly write code that causes a security issue. The only true way to avoid this from happening is education. Developers need to be aware of what security measures are necessary to properly store passwords so that they can avoid making design decisions that bypass them and make an app insecure.

Another reason for building our own authentication system is customization. In my experience, every site has custom requirements for their authentication system. Some sites require you to use two-factor authentication to log in; others require annual password resets without repeating any old passwords, and

others require specific characters and password lengths. While many of us may not agree with the usefulness of all of these random criteria, there may be a situation where we still need to add a specific password requirement.

Finally, using services like Auth0 add an additional cost to your bottom line. I'm not saying that they aren't worth that additional cost in many cases, but there are also situations where their pricing is pricing isn't feasible.

11.3 What is a Hash Function?

A hash function is a function that accepts data of any arbitrary size and converts it into data of a fixed size (source). Hash functions will produce the same results given the same inputs.

The act of applying a hash function is often called "hashing", and the values returned by a hash function are referred to as hash values or hashes.

The following function is an example of a hash function that uses the length of the string and a modulus (`%`) operator to calculate a fixed value between `0` and `4`.

```
// An example of a hash function, albeit a bad one
func hash(s string) int {
  return len(s) % 5
}
```

We can plug any string into this function and it will give us a number between `0` and `4`. No matter how often we call this function, each string will give us the same result.

It is also worth noting that two values can generate the same result. The following code produces `2` for both inputs "hi" and "Hello, world".

```
package main

import "fmt"

func main() {
        fmt.Println(hash("hi"))
        fmt.Println(hash("Hello, world"))
}

func hash(s string) int {
        return len(s) % 5
}
```

Run this on the Go Playground: https://go.dev/play/p/yclBda-YR_y

It is inevitable that two inputs might produce the same hash value, as we are returning data in a fixed size. With infinite inputs and a fixed number of outputs, collisions (two inputs having the same hash value) are impossible to avoid. In our example collisions are very likely, but in the hashing functions we will be looking at collisions are incredibly uncommon.

Another key point is that it is impossible to take a hash value and convert it back into the original string. Looking at our Go example, we cannot take the hash value `2` and determine what string was passed in. We can't even determine exactly how many characters were in the string, as it could have been 2, 7, 12, or any other number of characters that results in `2` when the `% 5` operation is applied.

Our hashing function is pretty bad for a number of reasons, so let's look at an example using HMAC.

```
package main

import (
        "crypto/hmac"
        "crypto/sha256"
        "encoding/hex"
        "fmt"
)
```

```
func main() {
        // While HMAC uses a secret key, this is not the same as encryption
        // and cannot be used to decrypt a password.
        secretKeyForHash := "secret-key"

        password := "this is a totally secret password nobody will guess"

        // Setup our hashing function
        h := hmac.New(sha256.New, []byte(secretKeyForHash))

        // Write data to our hashing function
        h.Write([]byte(password))

        // Get the resulting hash
        result := h.Sum(nil)

        // The resulting hash is binary, so encode it to hex so we can read it as letters and n
        fmt.Println(hex.EncodeToString(result))
}
```

Run this on the Go playground: https://go.dev/play/p/57Lh03BauIK

Unlike our initial **hash** function, **HMAC** requires a secret key that is used to hash data. While this key is necessary to generate the same results, it is important to note that even with the key we cannot convert a hash back into its original value. The key just provides us with a way to generate unique hashes that others cannot replicate with the same input unless they also have the same key.

The rest of the code is used to set up the hash, to write data to the hash function, to get the resulting hash value, and finally since the hash value is a byte slice that might not map to valid ASCII characters, we use the **encoding/hex** package to encode the binary data into a hex string.

There are a variety of hashing functions available to us, and each is designed for a different purpose. For instance, the **map** type in Go uses hashes behind the scenes to index each value stored in the map. More can be read about this in the article Go maps in action.

As we build our application we are going to focus on two hashing functions: bcrypt and HMAC. bcrypt is a hashing function designed explicitly for hashing passwords, so we will be using that for our authentication system, then later in

the course we will use HMAC to hash session tokens.

When setting up an authentication system, we should always use a password-specific hashing function. Not only will it simplify things for us, it will also ensure that we aren't susceptible to attacks we haven't even thought to consider.

11.4 Store Password Hashes, Not Encrypted or Plaintext Values

One of the most common mistakes made when building an authentication system is to improperly store passwords. For instance, inexperienced developers might opt for an encryption, or worse yet, encoding or a raw password, without realizing the mistake they are making.

If it is possible to determine the password with the value stored in a database, a mistake has been made. Web applications should not be able to decrypt a users password, period. It doesn't matter what the use case is or what awesome service is being worked on; if user passwords can be decrypted by a web app, an attacker could gain access to the password as well.

Aside 11.1. What about password managers?

When encrypting and decrypting data, a key is needed. Otherwise the process will fail.

Password managers are unique in how they encrypt/decrypt stored passwords because they do not store the key on their web server. Instead, the key is stored locally on your computer or phone, then when you want to decrypt the data the key is used. Even if a hacker gained access to a password management's database, without the keys the data would be useless.

Password management apps might use encryption for your stored passwords, but they use techniques like we will learn about for authenticating users as this is necessary to securely handle user authentication.

At this point things may seem confusing. If we can't figure out what a user's password actually is, how can we verify their password when they log in?

Rather than storing a user's password, we are instead going to store a hash of their password.

```
"some password" => hash function => "2b573d0f1a3d..."
                                          ^
                                  we will store a hash
                                  value in our DB
```

Note: The hash value is truncated to make it easier to read. This would normally be a much larger value.

Storing a hash value means that even if someone does get access to our database, they won't actually know what each user's password is. They will only know what the hash of each password is.

```
id | password_hash
---------------------
1  | 2b573d0f1a3d...
```

When we want to authenticate a user we won't actually know what their password is, but we will know what it should hash to. This means we can take the password they are attempting to log in with, hash it, then verify it matches the hash we have stored in our database. In the example above the user might type in **`some password`**, we would hash it, then if the resulting hash matched **`2b573d0f1a3d...`** we would know it was the correct password.

Using this approach we can effectively validate a user's password while limiting our knowledge of what the actual password is. The logic here is that if we do not know what the users password is and we cannot calculate it, then an attacker with access to our database also couldn't determine the password.

Earlier we mentioned that two inputs can hash to the same value, so is it possible for someone to enter the wrong password and our application to consider it correct because the hashes match?

While this is technically possible, the odds of it happening are **extremely** unlikely, and doing it intentionally would be just as hard as guessing the real password. Due to how unlikely this is, it isn't worth worrying about in practice. Put another way - it is worth the trade-off compared to storing plaintext passwords in our database which has much larger security risks.

Aside 11.2. Encryption functions are not hashing functions

At some point you may learn about encryption functions like AES, and at first glance they might appear to be a good fit. After all, the way we code AES looks somewhat similar to HMAC. I want to once again reiterate that encryption and hashing are two different things, and encryption is not appropriate for an securing passwords.

When you encrypt a password it is reversible. Yes, you do need the encryption key to decrypt a password, but if someone has hacked your server it isn't a stretch to imagine that they might be able to figure out your encryption key. Once they have that key they will be able to decrypt every password in the database. This is not possible with hashing functions, even ones that use a key.

It is my suggestion that you stick with `bcrypt`, or if you want to explore alternatives `scrypt` and `argon2` are also great choices, albeit a little harder to use with Go.

11.5 Salt Passwords

The goal when securing passwords is to do it in such a way that even if an attacker gets a copy of our database, they still won't be able to figure out what each user's password is. While we don't ever want to have a data breach and leak our database to hackers, designing our system this way will ensure that even if it does happen our users don't need to worry about accounts on other websites that share a password also being compromised.

As we discussed earlier, it isn't possible to take a hash value and reverse it into the original password. Knowing this, we might think we are covered just by hashing passwords, but unfortunately hackers are a pesky bunch and over time they have developed quite a few tricks and techniques. One of those techniques is using **rainbow tables**.

While it isn't possible to reverse a password hash into its original value, it is possible to take a list of common passwords and to hash each of them. This is called a rainbow table.

Once a rainbow table has been created, it is possible to take the list of generated hash values and to compare them with hash values in a stolen database. The goal here is to look for any passwords that match a value in the rainbow table, then once a match is found the attacker knows what common password was used to generate that hash value.

Let's look at an example to further clarify. Imagine that a hacker was able to get access to our database, as well as the information about the hashing function and any secrets we are using with it. They might end up with a list of hashed passwords like below:

```
id | password
-- | ------------
 1 | c66d8e7d45e7
 2 | eff5364b19f5
 3 | 93435c1a22a9
 4 | b78e5973fae1
 5 | 7c12e834ef44
```

```
 6 | 79e1f4ccf82e
 7 | 890b86bf0779
 8 | df5d6e16ac62
 9 | 30fa6a3d24fe
 ```

The attacker cannot determine any user's password directly from the hash, but since they also k
```

password abc123 secret none

```
Next, the hacker could use the information about our hashing function to run all of these throu
```

password | hash ——— | ————— password | 93435c1a22a9 abc123 | c66d8e7d45e7 secret | df5d6e16ac62 none | 79e1f4ccf82e

```
This is called a rainbow table, and with it a hacker could take the stolen database and compare
```

id | password | cracked_pw – | ————— | ———— 1 | c66d8e7d45e7 | abc123 2 | eff5364b19f5 | 3 | 93435c1a22a9 | password 4 | b78e5973fae1 | 5 | 7c12e834ef44 | 6 | 79e1f4ccf82e | none 7 | 890b86bf0779 | 8 | df5d6e16ac62 | secret 9 | 30fa6a3d24fe |

```
While it didn't tell the attacker every user's password, it did give them several matches and t

The first challenge for hackers using rainbow tables is that it takes a while to create each ra

That is where salting passwords becomes beneficial. Salting a password is the act of adding ran

For example, when a user signs up they might type in their password of `abc123`. At that time w
```

id | password_hash | salt – | ————— | ———— 1 | 64047ee6222f | ja08d
```

```
When the user signs in and gives us a password to authenticate, we know to add the `ja08d` salt

Each user should be given their own salt value. This value will be stored in the database, so i

\begin{aside}
\label{aside:bcrypt-has-salt-built-in}
\heading{bcrypt has salt built in}

We will be using `bcrypt` to hash our passwords, and because this algorithm was designed specif


\end{aside}

Another related topic is a **password pepper**. Like a password salt, a pepper is random value

In practice a pepper isn't necessary and the rest of our security measures are sufficient, so w

\begin{aside}
\label{aside:where-do-the-terms-salt-and-pepper-come-from}
\heading{Where do the terms salt and pepper come from?}

Salting is a term that has been around in cryptography for a long time, but unfortunately is is

Some say it was inspired by Roman soldiers who would salt the earth to make it less hospitable

Regardless, salting was the first term to be coined, and then the term pepper came about becaus


\end{aside}


## Learning bcrypt with a CLI

Now that we understand the importance of hashing our passwords, let's look at how to use `bcryp

\begin{aside}
\label{aside:warning-this-is-not-a-cli-tutorial}
\heading{Warning: This is not a CLI tutorial}

While we are building a fairly simple CLI in this lesson, this is far from a proper CLI tutoria

In short, there is no need to report bugs with this CLI. It serves its teaching purpose despite


\end{aside}

```bash
mkdir cmd/bcrypt
code cmd/bcrypt/bcrypt.go
```
```

```
In this command we will make a simplified CLI that allows us to do two things:

1. Hash a password
2. Compare a password with a hash to see if it is correct

To do this, we will use subcommands. For instance, we will use the following:

```bash
# Hash a password
go run cmd/bcrypt/bcrypt.go hash "some password here"

# Compare a hash and password
go run cmd/bcrypt/bcrypt.go compare "some password here" "some hash here"
```

To do this, we first need to know wether the user typed `hash` or `compare` as the first argume

```go
package main

import (
        "fmt"
        "os"
)

func main() {
        for i, arg := range os.Args {
                fmt.Println(i, arg)
        }
}
```

After running this, we will see an output similar to that below.

```bash
go run cmd/bcrypt/bcrypt.go hash "secret password"
```

This will result in the following output.
```

0 /var/folders/_5/866w6b. . . /T/go-build488928631/b001/exe/bcrypt 1 hash 2 secret password

```
When we call `go run` the Go tooling will create a binary behind the scenes then run it. That i

We can make this look less awkward if we build a binary and run it, but keep in mind how you ru
```

```
```bash
# build it
go build cmd/bcrypt/bcrypt.go

# run the binary. This may be bcrypt.exe in Windows.
./bcrypt hash "secret password"
```

This will provide the following output:
```

0 ./bcrypt 1 hash 2 secret password

```
Similarly, we can try the compare subcommand.

```bash
./bcrypt compare "secret password" "some hash value"
```

This will give us the following output:
```

0 ./bcrypt 1 hash 2 secret password 3 some hash value

```
Knowing that the second argument (index `1`) in `os.Args` is our command, we can use that and a

```go
package main

import (
        "fmt"
        "os"
)

func main() {
        switch os.Args[1] {
        case "hash":
                hash(os.Args[2])
        case "compare":
                compare(os.Args[2], os.Args[3])
        default:
    fmt.Printf("Invalid command: %v\n", os.Args[1])
        }
}

func hash(password string) {
```
```

```
        fmt.Printf("TODO: Hash the password %q\n", password)
}

func compare(password, hash string) {
        fmt.Printf("TODO: Compare the password %q with the hash %q\n", password, hash)
}
```

We are using `os.Args[1]` to access the second argument passed when our app was run. This tells

When a user opts to run the `hash` functionality, we take the third argument (index `2` with ze

We can try running both commands in our terminal.

```bash
go run cmd/bcrypt/bcrypt.go hash "secret password"
# Outputs:
# TODO: Hash the password "secret password"

go run cmd/bcrypt/bcrypt.go compare "secret password" "hash value"
# Outputs
# TODO: Compare the password "secret password" with the hash "hash value"
```

While our app is far from perfect, especially with the potential to panic, it is sufficient to

In the next few lessons we will see how to use bcrypt to implement the hashing and comparison f

Source Code

- [Source Code](https://github.com/calhounio/lenslocked/tree/bcrypt_cli) - see the final source

- [Diff](https://github.com/calhounio/lenslocked/compare/querying_multiple_records...bcrypt_cli

Hashing Passwords with bcrypt

We are now ready to add hashing functionality to our CLI. To do this, we first need to `go get`

```bash
go get golang.org/x/crypto/bcrypt
```

Bcrypt is a hashing package that is part of the Go project, but is maintained outside of the st

We can check out bcrypt's documentation at <https://pkg.go.dev/golang.org/x/crypto/bcrypt>

The primary functions we will be using in this package are:

- [GenerateFromPassword](https://pkg.go.dev/golang.org/x/crypto/bcrypt#GenerateFromPassword)
- [CompareHashAndPassword](https://pkg.go.dev/golang.org/x/crypto/bcrypt#CompareHashAndPassword

```
The first, `GenerateFromPassword`, is used to generate a password hash from an raw user passwor

The second, `CompareHashAndPassword`, is used to compare a user-provided password with a hash w

Our CLI app already has a `hash` function for us to work with, so we can open up `bcrypt.go` an

```go
func hash(password string) {
        hashedBytes, err := bcrypt.GenerateFromPassword(
                []byte(password), bcrypt.DefaultCost)
        if err != nil {
                fmt.Printf("error hashing: %v\n", err)
                return
        }
        hash := string(hashedBytes)
        fmt.Println(hash)
}
```

We want to use the `GenerateFromPassword` function to generate our hash, so we start by calling

- The password, as a byte slice
- The cost

Our password is a string, so we need to covnert it into a byte slice. We can do this with the f

```go
[]byte(password)
```

bcrypt's hash generating function also requires a cost parameter, but what is that? We didn't d

Password hashing functions often have a variable cost that is used to dictate how much work (an

The cost is a variable because it is meant to change over time as computers get faster. A good

With password hashing the race car is our CPU, and the number of laps needed to be completed is

Figuring out and keeping track of the best cost would be pretty tedious, so bcrypt provides us

The `GenerateFromPassword` function returns a byte slice and an error. If we receive an error w
```

[36 50 97 36 49 48 36 . . .]

```
Go's print functions treat a byte slice like any other slice and print out each individual byte

```go
```
```

```
string(hashedBytes)
```

```
\begin{aside}
\label{aside:not-all-bytes-make-valid-characters}
\heading{Not All Bytes Make Valid Characters}

Not all byte slices will can be converted into a valid string because not all bytes map to vali


\end{aside}

If we run the `hash` command we should now see a hash value being printed out.


```bash
go run cmd/bcrypt/bcrypt.go hash "secret password"
```

Although the output will differ between runs, it should be similar to the one below.
```

$2a$10$GIdKwhQ8jUnH8bNS/CDDM.yhb9tnPc6sZWI4NIWytGOvGwGAHIJLO

```
If we run our code multiple times we will get different output values each time. This occurs be

\begin{aside}
\label{aside:how-is-the-salt-stored-in-the-hash}
\heading{How is the Salt Stored in the Hash?}

This isn't exactly how bcrypt works, but imagine bcrypt generated the salt `ja08d` and was hash
```

alskdfjasd0f9j8

```
At this point bcrypt could append the hash to the result, and then return a value like below:
```

ja08d-alskdfjasd0f9j8

```
In the future when comparing passwords to a hash, bcrypt knows to read the salt from the start


```

````
\end{aside}

Our CLI now supports hashing passwords! In the next lesson we will look at how to compare a pas

**Source Code**

- [Source Code](https://github.com/calhounio/lenslocked/tree/cli_hash_func) - see the final sou

- [Diff](https://github.com/calhounio/lenslocked/compare/bcrypt_cli...cli_hash_func) - see the


## Comparing a Password with a bcrypt Hash

Our next goal is to update our code so we can compare a password with a hash value.

```bash
go run cmd/bcrypt/bcrypt.go compare \
  "secret password" \
  "some hash here"
```

\begin{aside}
\label{aside:use-a-backslash-for-multi-line-commands}
\heading{Use a Backslash for Multi-line Commands}

If we want to spread a bash command across multiple lines we can do so by adding a backslash (`


\end{aside}

We are going to be using the [CompareHashAndPassword](https://pkg.go.dev/golang.org/x/crypto/bc

```go
func compare(password, hash string) {
        err := bcrypt.CompareHashAndPassword([]byte(hash), []byte(password))
        if err != nil {
                fmt.Printf("Password is invalid: %v\n", password)
                return
        }
        fmt.Println("Password is correct!")
}
```

Like in the `hash` function, we need to convert our strings into byte slices for bcrypt. After

On the other hand, if no error is returned it means the password and the hash matched. In that

Using a hash calculated from an earlier run, we can test this by running the following code:

```bash
go run cmd/bcrypt/bcrypt.go compare \
````
````

```
  "secret password" \
  '$2a$10$GIdKwhQ8jUnH8bNS/CDDM.yhb9tnPc6sZWI4NIWytGOvGwGAHIJLO'
```

```
\begin{aside}
\label{aside:use-single-quotes-for-the-hash}
\heading{Use Single Quotes For the Hash}

Characters like `$2` can have a special meaning in bash strings with double quotes (`"`). To pr


\end{aside}

We can test the code out by generating a few hashes and trying to validate them with valid and

We won't be using this CLI moving forward, but it is worth keeping around to reference when we

**Source Code**

- [Source Code](https://github.com/calhounio/lenslocked/tree/cli_compare_func) - see the final

- [Diff](https://github.com/calhounio/lenslocked/compare/cli_hash_func...cli_compare_func) - se
```

Chapter 12

Adding Users to our App

12.1 Defining the User Model

Now that we have learned about SQL and all the steps required to use it with Go, and we have also learned how to secure passwords properly using bcrypt, we are going to start adding the user model to our application. We will start by keeping things simple and focusing on only the fields we need - a user's **ID**, their **email** address, and a **password hash**.

We need an ID to serve as a unique identifier for each user that doesn't change over time. While the email address needs to be unique for each user, it could potentially change in the future. A user's ID will not ever change. The need for an email address is pretty self explanatory. Users will use this to sign into their account, and we can use it to send them emails for things like resetting their password. Finally we need the password hash to authenticate a user and make sure others cannot access their account.

We have not launched our application, so at this stage it is okay to experiment a bit and not spend too much time designing all the database tables our application will need. After all, we can always delete our database and recreate it if we need to make a breaking change.

Once we deploy our application we will need to invest more time upfront in designing our database tables and our models since we won't want to delete our production database and all the user data stored in it, so any changes will need to be implemented in a way that doesn't destroy any data or break our application's functionality. This is typically called a database migration, which we will learn about later in the course.

We are ready to define our users, but we first need to decide what our source of truth is going to be. One option is to consider our Go code the source of truth, so anytime we want to make changes we would first update our Go code, then update the SQL table to reflect the updated Go code. This is what happens with many ORMs. The other option is to consider our SQL tabel the source of truth; in this scenario all models will first be defined as a SQL table, then we will write Go code to match the SQL table.

We will be using SQL as our source of truth, so we will start with the SQL code to create our users table.

Aside 12.1. Generating Go from SQL

In addition to ORMs that generate SQL for us, there are also ORMs that generate Go code given a SQL database. SQLBoiler is one example written in Go.

Let's create the `models` directory that will contain all of our model related code, then inside that directory add the `sql` directory that stores our SQL specific code related to models.

```
mkdir -p models/sql
code models/sql/users.sql
```

Inside the `users.sql` file we will define our users table using the SQL we have learned.

```
CREATE TABLE users (
  id SERIAL PRIMARY KEY,
  email TEXT UNIQUE NOT NULL,
  password_hash TEXT NOT NULL
);
```

Next we want to run this SQL on our database. While it is possible to run SQL files on a Postgres database, doing so with our Docker setup is a bit more complicated because the `users.sql` file isn't available inside our docker container. Later in the course we will be learning about migrations which will eliminate the need to do this, so we won't spend the time addressing it now. Instead, we will connect to our database with `psql`, open up `users.sql`, copy the contents of the file, and then paste the contents into our `psql` session.

First we will make sure we are starting with a clean slate. To do this, we will recreate our docker containers.

```
docker compose down
docker compose up -d
```

Next we connect to our database with `psql`.

```
docker exec -it lenslocked-db-1 /usr/bin/psql -U baloo -d lenslocked
```

Aside 12.2. Finding the correct container name

Just a quick reminder - you may need to use `docker ps` to find the correct container name when running `docker exec` commands.

Next we copy the contents of `users.sql` and paste it into our terminal where `psql` is running. Be sure to hit enter so the SQL is executed.

```
psql (14.2 (Debian 14.2-1.pgdg110+1))
Type "help" for help.

lenslocked=# CREATE TABLE users (
  id SERIAL PRIMARY KEY,
  email TEXT UNIQUE NOT NULL,
  password_hash TEXT NOT NULL
);
CREATE TABLE
lenslocked=#
```

Finally, we run a query to ensure the table was created correctly.

```
lenslocked=# SELECT * FROM users;
 id | email | password_hash
----+-------+---------------
(0 rows)
```

Let's define the user model in our Go code next. First, we create a file for our user-related code.

```
code models/user.go
```

Once we have a source file we need to declare the **`User`** type.

```
package models

type User struct {
        ID           int
        Email        string
        PasswordHash string
}
```

We now have a user model defined in both SQL and Go. We are ready to start writing the Go code required to interact with our database table and translate the results into our **`User`** type.

Aside 12.3. uint IDs

`int` is a data type that can store both positive and negative numbers. `uint` is similar, but only stores zero and positive numbers. As a result, its max value is twice as large as the max value of an `int` because it doesn't need to consider negative numbers.

When declaring an `ID` field, many codebases and ORMs will use `uint` because IDs will never be negative in the database. I am opting to use `int` because it is a bit simpler for learning, and in my opinion using `uint` is only a stopgap; if IDs start to exceed the bounds of an `int`, an application will quickly need to migrate to using something like UUID for IDs.

Source Code

- Source Code - see the final source code for this lesson.
- Diff - see the changes made in this lesson.

12.2 Creating the UserService

Now that we have a `users` table along with a `User` type, we want to add code to our application that allows us to create users and query them using our `models.User` type.

We will start by first considering what we need to create a new user in our database. For this we need three things:

- `*sql.DB` - a database connection

- `email` - the email address of the user
- `password` - the user's password (we will store a hash of it)

The email address and password will be provided to us by the user when they create their account, but the database connection is something we need to provide every time we interact with the database. There are two ways we can represent this in our Go code:

1. Accept an `*sql.DB` as an argument to each of our functions that interact with our database.
2. Create a type with `*sql.DB` as a field, then add methods to that type for interacting with our database.

The first looks like any other function, but we have the `*sql.DB` type as the first argument:

```
func CreateUser(db *sql.DB, email, password string) (*User, error) {
  // Create and return the user using `db`
}
```

The second option involves more upfront code, but we do not need to pass the `*sql.DB` into each individual method.

```
type UserService struct {
        DB *sql.DB
}

func (us *UserService) Create(email, password string) (*User, error) {
        // Create and return the user using `us.DB`
}
```

I prefer the latter in most cases because it allows us to setup the `UserService` once at the start of our application, and then from that point onwards our code doesn't have to pass around things like the `sql.DB` object.

Using a type with methods also allows us to write SQL agnostic code using interfaces.

```
package demo

type UserCreator interface {
  Create(email, password string) (*models.User, error)
}
```

In the code above we declare an interface for creating users, and as long as we are provided with a type that satisfies this interface our code will work. It doesn't matter if the code users SQL, MongoDB, or something else behind the scenes to our `demo` package.

Aside 12.4. Interfaces make testing easier

While it is extremely unlikely that we will migrate our app from one database to another, interfaces like this can be useful for testing. We still have a dependency on the `models` package to access the `models.User` type, but we can mock the UserService without needing an SQL database connection. This is a slightly more advanced topic that I don't suggest diving into until you have the basics of web development and Go under your belt, but it is something to consider for the future.

Using a `UserService` type with a `*sql.DB` field can also help encourage developers to write SQL-related code in the correct location - inside our `models` package as a method on the `UserService` type - rather than placing SQL queries anywhere and everywhere in our code that the `*sql.DB` object happens to be available. This is a mistake beginners make quite often that can lead to hard to maintain code.

Later in the course we will explore alternative ways to structure our code, at which time the benefits of this decision will become a bit more apparent. For now, trust that using types is a solid choice and let's code this all up.

Open `models/user.go` and add the following code.

```
type UserService struct {
        DB *sql.DB
}
```

This should also add the `"database/sql"` import.

Anytime we want to interact with the users table, it will likely be a new method added to this `UserService` type we created. In upcoming lessons we will add methods to the `UserService` like `Create` and `Authenticate` that allow us to interact with our database and perform the relevant actions.

Source Code

- Source Code - see the final source code for this lesson.
- Diff - see the changes made in this lesson.

12.3 Create User Method

Now that we have a `UserService` to attach methods to, we are going to create a method that allows us to create new users. In an ideal world we would accept a `*models.User` to our create function:

```
func (us *UserService) Create(user *User) error {
  // ...
}
```

This approach works well because we can use the same `User` type for creating, updating, and querying users. We are also using a pointer (`*User`) which allows us to set the ID and other fields once the database assigns them.

Unfortunately, our `models.User` type has a `PasswordHash` field, not a `Password` field. We could ask users to hash the password before calling `Create`, but that really feels like logic that should be part of our `UserService` code.

Another option is to document that the `PasswordHash` field should contain a plaintext password when calling `Create`. Our `UserService` could then hash the value before saving it in the database, but this could lead to bugs where someone doesn't read the docs and hashes the password on their own. Ideally, we want to make things intuitive and easy to use so we avoid bugs like this.

When our requirements for creating or updating a resource differ from what we store in our models, we have a few options. We will consider the following:

1. Create a new type for that specific action. A `NewUser` type in this case.
2. Write code that accepts the fields we need without a custom type.

The first option might look like the following code:

```
type NewUser struct {
  Email string
  Password string
}

func (us *UserService) Create(nu NewUser) (*User, error) {
  // TODO: Implement this
}
```

In this code we accept the `NewUser` type which has all the specific fields we need for creating a user.

The second option is similar, but rather than creating a new type we accept the fields we need as arguments to our function.

```
func (us *UserService) Create(email, password string) (*User, error) {
  // TODO: Implement this
}
```

In both cases it makes sense to return the `User` that was created so we can use it moving forward.

Either approach is valid, and which we should use depends. Here are a few questions to consider when deciding which is a better fit:

- *How many parameters does our function need?* Two. Email and password. If we we had many more parameters, using a `NewUser` type would be a better fit.
- *Are we likely to need additional parameters later?* Probably not, but if we felt we might need new fields a `NewUser` type might be a better choice.
- *Does the team have a personal preference?* If the team prefers to use one approach over another, it might be worth using that approach all the time for consistency.
- *Are we, or will we be, using one of these patterns in other code?* If we expect to be using `NewPhoto` and `NewGallery` types later to create those images, it might be worth using `NewUser` now to keep things consistent.

Regardless of which route we go with, we can always refactor our code in the future. This is more challenging with a library that third parties will consume, but given that this is a private package meant to interact with our wn database we should be able to make changes with minimal issues.

Moving forward with the course, we are going to start by using function parameters without a custom type, but we can change that up later if it becomes necessary.

Open up your `models/user.go` source file.

```
code models/user.go
```

Next, stub out the `Create` method using the approach we settled on.

```
func (us *UserService) Create(email, password string) (*User, error) {
        // TODO: Implement this
        return nil, nil
}
```

Aside 12.5. Why do we return a pointer?

In my experience, returning a pointer is best here for a couple reasons:

1. We can return `nil` when we want to instead of an empty user.
2. We will be using our model types (`User`, `Gallery`, etc) as pointers in future code and it is best to be consistent rather than mixing pointer and non-pointer use of a type.

Postgres is case-sensitive. This means is it considers the string `jon@calhoun.io` to be different than `JON@CALHOUN.IO`. We know that an both of those email addresses will email the same person, so we need to make sure this is reflected in our database. One way to handle this is to convert all email addresses to lowercase before storing them in our database. Add the following code to `Create` to do that.

```
email = strings.ToLower(email)
```

Next, we need to hash the password using the knowledge we gained about bcrypt. The following code does that and is very similar to the code in **bcrypt.go**, except the error handling has changed.

```
hashedBytes, err := bcrypt.GenerateFromPassword(
  []byte(password), bcrypt.DefaultCost)
if err != nil {
  return nil, fmt.Errorf("create user: %w", err)
}
passwordHash := string(hashedBytes)
```

Finally, we need to use the knowledge we gained from the SQL sections to write the user info to the DB, read the ID that was assigned, and return everything in a **User** object.

```
user := User{
  Email:        email,
  PasswordHash: passwordHash,
}
row := us.DB.QueryRow(`
  INSERT INTO users (email, password_hash)
  VALUES ($1, $2) RETURNING id`, email, passwordHash)
err = row.Scan(&user.ID)
if err != nil {
  return nil, fmt.Errorf("create user: %w", err)
}
return &user, nil
```

In this code we first create a **User** with the information we are writing to the DB. This is done so we can return exactly what was written to the database, so it should be done after doing things like converting the email address to lowercase.

Once we have a **User** ready we can use the **QueryRow** function to insert the record. This returns a **sql.Row** and we call **row.Scan** on it to scan any re-

turned data. The only data we are scanning is the ID, so we pass in a pointer to the `user.ID` field and scan it directly into our `user` struct's ID field.

We never want to ignore errors, otherwise we might not know when things are going wrong, so our next step is to check for errors returned by `Scan`. If one occurs, we wrap it with `fmt.Errorf` so we know that it occurred while creating a new users. This helps debugging later since it adds a bit more context to the error message. We also return `nil` as the `*User` return value to make it clear that no user was created.

If no errors occur, we return a pointer to the user and a `nil` error.

Putting it all together, `Create` should look like the code below.

```
func (us *UserService) Create(email, password string) (*User, error) {
        email = strings.ToLower(email)
        hashedBytes, err := bcrypt.GenerateFromPassword(
                []byte(password), bcrypt.DefaultCost)
        if err != nil {
                return nil, fmt.Errorf("create user: %w", err)
        }
        passwordHash := string(hashedBytes)

        user := User{
                Email:        email,
                PasswordHash: passwordHash,
        }
        row := us.DB.QueryRow(`
                INSERT INTO users (email, password_hash)
                VALUES ($1, $2) RETURNING id`, email, passwordHash)
        err = row.Scan(&user.ID)
        if err != nil {
                return nil, fmt.Errorf("create user: %w", err)
        }
        return &user, nil
}
```

We can test that this code is working with our experimental code. Open up `cmd/exp/exp.go` and comment out (or delete) all of the code in `main()`. Replace it with the following code:

```
// cmd/exp/exp.go

func main() {
  cfg := PostgresConfig{
    Host:     "localhost",
    Port:     "5432",
    User:     "baloo",
    Password: "junglebook",
    Database: "lenslocked",
    SSLMode:  "disable",
  }
  db, err := sql.Open("pgx", cfg.String())
  if err != nil {
    panic(err)
  }
  defer db.Close()

  err = db.Ping()
  if err != nil {
    panic(err)
  }
  fmt.Println("Connected!")

  us := models.UserService{
    DB: db,
  }
  user, err := us.Create("bob@bob.com", "bob123")
  if err != nil {
    panic(err)
  }
  fmt.Println(user)
}
```

Most of this code is setting up a database connection and instantiating the **`UserService`**. After that is all done, we create a user with the email **`bob@bob.com`** and the password **`bob123`** using the following code:

```
user, err := us.Create("bob@bob.com", "bob123")
if err != nil {
  panic(err)
}
```

Our error handling isn't stellar, but we can see that creating a user is now a much simpler process - a single function call!

Run the code to make sure it is working.

```
go run cmd/exp/exp.go
```

This should give us an output similar to that below:

```
&{1 bob@bob.com $2a$10$v2341SCqeiH4P5haiaQkfO0q5wcOqezO0Dn6t50k6Y/42rLsAWLp.}
```

The `1` is the user ID. This is followed by the user's email address, and finally their password hash. The ID and hash may be different in your case, but that is okay.

We can also open up `psql` to verify the user was created in our database.

```
docker exec -it lenslocked-db-1 /usr/bin/psql -U baloo -d lenslocked
```

```
SELECT * FROM users;
```

Which gives us output similar to that below.

```
 id |    email    |                          password_hash
----+-------------+--------------------------------------------------------------
  1 | bob@bob.com | $2a$10$v2341SCqeiH4P5haiaQkfO0q5wcOqezO0Dn6t50k6Y/42rLsAWLp.
(1 row)
```

We have successfully created a user with our new `UserService`! If we want to rerun our code we will need to use a new email address, as our database requires a unique email address for each user, but we can run the code a few times to see how it works and even see what the error messages look like.

Source Code

- Source Code - see the final source code for this lesson.
- Diff - see the changes made in this lesson.

12.4 Postgres Config for the Models Package

At the moment our `models` package appears to be database-agnostic, but in reality it isn't. We couldn't declare a `models.UserService` with a SQLite database connection, as our SQL queries currently use the Postgres syntax of `$1`, `$2`, and so on for variables.

```
INSERT INTO users (email, password_hash)
VALUES ($1, $2) RETURNING id
```

While we could try to update our code to work with any SQL database variant, it would take a good bit of effort to make this work, and we don't even know if we will ever need to use another SQL variant. If we were building a library that we expected many project to use, this might make sense, but given that our `models` package is intended to only be used within our application it makes less sense to make our code universal. Instead, a better choice is to embrace the fact that we are using Postgres and to make our package even easier to use for that database.

One way we can do this is by providing the `PostgresConfig` type that we previously declared in `exp.go`. This will make it easier for users of our `models` package to setup a database connection.

First, create a `postgres.go` source file for the config type.

```
code models/postgres.go
```

Next, add the code for the **PostgresConfig** type and its **String** method. The code for the type and the method are the same as the code in **exp.go**, so it can be copied from there.

```
package models

import "fmt"

type PostgresConfig struct {
        Host     string
        Port     string
        User     string
        Password string
        Database string
        SSLMode  string
}

func (cfg PostgresConfig) String() string {
        return fmt.Sprintf("host=%s port=%s user=%s password=%s dbname=%s sslmode=%s", cfg.Host
}
```

Since we are using Docker, we also know what the default development config looks like. We can use that information to provide a function that returns a **PostgresConfig** that works with our docker compose.

```
func DefaultPostgresConfig() PostgresConfig {
        return PostgresConfig{
                Host:     "localhost",
                Port:     "5432",
                User:     "baloo",
                Password: "junglebook",
                Database: "lenslocked",
                SSLMode:  "disable",
        }
}
```

Another portion of our code that could be simplified is opening the database connection. The **models** package is designed to work with the **github.com/jackc/pgx/**
driver, so we can import this driver automatically.

```
import (
        "fmt"

        _ "github.com/jackc/pgx/v4/stdlib"
)
```

While we are at it, we can also add a method that makes it easier to open a SQL database connection using just the **PostgresConfig**. That way users of our **models** package won't need to think about what database drive to import, or what the name of the driver is.

```
// Open will open a SQL connection with the provided
// Postgres database. Callers of Open need to ensure
// the connection is eventually closed via the
// db.Close() method.
func Open(config PostgresConfig) (*sql.DB, error) {
        db, err := sql.Open("pgx", config.String())
        if err != nil {
                return nil, fmt.Errorf("open: %w", err)
        }
        return db, nil
}
```

Inside the **Open** function we mostly reuses code from the **exp.go** source file, and anyone using it will still need to handle closing the database connection. The real benefit of the new **Open** function is that users of our **models** package can open a database connection using just a **PostgresConfig** without needing to think about database drivers.

Finally, let's head back to **cmd/exp/exp.go** and update it to use our updated **models** package.

```
package main

import (
        "fmt"

        "github.com/joncalhoun/lenslocked/models"
```

```
)

func main() {
        cfg := models.DefaultPostgresConfig()
        db, err := models.Open(cfg)
        if err != nil {
                panic(err)
        }
        defer db.Close()

        err = db.Ping()
        if err != nil {
                panic(err)
        }
        fmt.Println("Connected!")

        us := models.UserService{
                DB: db,
        }
        user, err := us.Create("bob4@bob.com", "bob123")
        if err != nil {
                panic(err)
        }
        fmt.Println(user)
}
```

With the changes we made our `exp.go` source file no longer needs to import a SQL driver, nor does it need to setup a `PostgresConfig`; we can instead use the default config.

All of these changes will make it easier for us as we integrate the UserService with our signup form over the next few lessons.

Source Code

- Source Code - see the final source code for this lesson.

- Diff - see the changes made in this lesson.

12.5 UserService in the Users Controller

Our next goal is to make the sign up page actually work. That is, we when a user visits <localhost:3000/signup> and submits the form, we want to process the inputs and attempt to create a new user.

To do this we will need to make our **`models.UserService`** available inside our users controller at **`controllers/users.go`**. Let's open that source file.

```
code controllers/users.go
```

Next, let's add a field to the **`controllers.Users`** type that is of the type **`*models.UserService`**.

```
type Users struct {
        Templates struct {
                New Template
        }
        UserService *models.UserService
}
```

Now we have access to the **`UserService`** field, but it isn't getting assigned to anything. To do that, we need to update code in our **`main()`** function.

```
code main.go
```

Sidenote: We will eventually move this into the **`cmd`** *directory like our other main packages.*

Find the section of code where we the users controller is instantiated and add the following code. If following along, it should be around line 25 in **`main.go`**

```
func main() {
  // ...

        // Setup a database connection
        cfg := models.DefaultPostgresConfig()
        db, err := models.Open(cfg)
        if err != nil {
                panic(err)
        }
        defer db.Close()

        // Setup our model services
        userService := models.UserService{
                DB: db,
        }

        // Setup our controllers
        usersC := controllers.Users{
                UserService: &userService,
        }
  // ... the rest of our routing is unchanged.
}
```

The first thing we are doing in this code is setting up a database connection. This is identical to the code in **`exp.go`** that we have already seen. We also use **`defer`** to close the connection when our **`main()`** function exits.

```
// Setup a database connection
cfg := models.DefaultPostgresConfig()
db, err := models.Open(cfg)
if err != nil {
  panic(err)
}
defer db.Close()
```

Once our database connection is setup we can use it to setup our services used to interact with models. For now the only one we have is the UserService, but later we will have a few services to work with.

```
// Setup our model services
userService := models.UserService{
  DB: db,
}
```

Finally, we update our previous instantiation of the `usersC` variable and assign the `UserService` field to the user service we setup.

```
usersC := controllers.Users{
  UserService: &userService,
}
```

We are now ready to use the UserService inside our users controller.

Source Code

- Source Code - see the final source code for this lesson.
- Diff - see the changes made in this lesson.

12.6 Create Users on Signup

Now that we have access to an instantiated UserService inside of our `controllers.Users` methods, we are ready to update the `Create` method to process a signup request. Open the users controller source file.

```
code controllers/users.go
```

Head to the `Create` method and update it with the following code.

```
func (u Users) Create(w http.ResponseWriter, r *http.Request) {
        email := r.FormValue("email")
        password := r.FormValue("password")
        user, err := u.UserService.Create(email, password)
        if err != nil {
                fmt.Println(err)
                http.Error(w, "Something went wrong.", http.StatusInternalServerError)
                return
        }
        fmt.Fprintf(w, "User created: %+v", user)
}
```

Aside 12.6. Don't output arbitrary errors

While we could call `http.Error` with the message from `err.Error()`, we should avoid doing this. The main reason is that we could unintentionally leak sensitive information. For instance, if the error message was about not being able to connect to a database, it might include the connection string used.

My general rule is to never print out an error message to end users unless it has been explicitly designed to be exposed to end users. Later in the course we will see how to do this with custom errors and interfaces.

The first few lines are getting the submitted form values and assigning them to variables. After that we call `UserService.Create` which will attempt to create a user in our database. From this point onwards we are checking for errors, and if non exist we print out a message stating that the user was created.

Our error handling isn't the best, and we don't sign a user in just yet, but this is a good start to the signup page.

Before proceeding, we should test that our code all works. Restart the server and head over to the signup page. Fill in a unique email address that isn't in the database as well as a password. Hit "Sign up" and verify that a user was created either with Adminer or with psql.

```
docker exec -it lenslocked-db-1 /usr/bin/psql -U baloo -d lenslocked
```

```
SELECT * FROM users;
```

```
 id |       email        |                          password_hash
----+--------------------+------------------------------------------------------------
  2 | test@calhoun.io    | $2a$10$ygwiWK3gwntEyGSNeIlx3uFDfdBNTwFMLOn/BHOKGdR6XD7umZj/.
  3 | bob@bob.com        | $2a$10$T8LO0zkhRe19mSR2ifSjo.NBCaI2v8OElK0eVWeWjpoLMOu5i9emm
(2 rows)
```

We can even use the bcrypt CLI to test our password was hashed correctly.

```
go run cmd/bcrypt/bcrypt.go compare \
  "bob123" \
  '$2a$10$T8LO0zkhRe19mSR2ifSjo.NBCaI2v8OElK0eVWeWjpoLMOu5i9emm'
```

We have now successfully learned how to connect all of the pieces of our application:

1. We create a controller type like **`controllers.Users`**.
2. Next we add fields to this type with views that we need to render pages.
3. We also add fields to the type for services we need access to from our **`models`** package.
4. When setting up our application, we make sure these fields are all assigned correctly.

Using this pattern helps us separate all of our logic making it easier to maintain. This is often called separation of concerns. If we ever find ourselves writing SQL inside of a controller, or adding lots of logic to a view, chances are we are

not properly separating our logic and it could lead to code that is challenging to maintain.

Over the next few sections we will learn how to properly create a user session and sign a user in after they create an account, but for now we will have to leave the `Create` method as-is.

Source Code

- Source Code - see the final source code for this lesson.
- Diff - see the changes made in this lesson.

12.7 Sign In View

We can't have an authentication system without a sign in page, so let's focus on getting that built. For now we will focus on rendering the form to sign in, then in the future we can handle processing the form and authenticating a user.

First, let's add a template field to the `controllers.Users` type so we can render the page.

```
code controllers/users.go
```

```
type Users struct {
        Templates struct {
                New    Template
                SignIn Template
        }
        UserService *models.UserService
}
```

Next, we need a method on the `Users` type that will handle rendering the page. We will the method `SignIn`, and the code for the method will be nearly

identical to the **New** method, so we can copy/paste that code as a starting point, then change the method name and the template being used.

```
func (u Users) SignIn(w http.ResponseWriter, r *http.Request) {
        var data struct {
                Email string
        }
        data.Email = r.FormValue("email")
        u.Templates.SignIn.Execute(w, data)
}
```

While we could have used a static controller to render this page, it is nice to have the ability to parse any URL query params and prefill them into the form like we learned to do with our sign up page, so we are opting for a custom **SignIn** method.

Now that our controller is prepared we are going to create the **gohtml** template file.

```
code templates/signin.gohtml
```

Once again our page will look fairly similar to the **signup.gohtml** template, so feel free to copy/paste that file as a starting point. From there we need to change:

- The **h1** tag's text to read "Welcome back!"
- The sign in **form** tag should have an **action** that directs us to the **"/signin"** path, as this is the path we will use to process the sign in form.
- The form **button**'s text to read "Sign in" instead of "Sign up"
- The footer of the form should read "Need an account?" instead of "Already have an account?", since we are looking at the sign in page and anyone who already has an account won't need linked anywhere. Make sure

to also update the path of the link so it directs the user to the **`/signup`** page if they need an account.

The final template code is shown below.

```
{{template "header" .}}
<div class="py-12 flex justify-center">
  <div class="px-8 py-8 bg-white rounded shadow">
    <h1 class="pt-4 pb-8 text-center text-3xl font-bold text-gray-900">
      Welcome back!
    </h1>
    <form action="/signin" method="post">
      <div class="py-2">
        <label for="email" class="text-sm font-semibold text-gray-800">
          Email Address
        </label>
        <input
          name="email"
          id="email"
          type="email"
          placeholder="Email address"
          required
          autocomplete="email"
          class="w-full px-3 py-2 border border-gray-300 placeholder-gray-500
            text-gray-800 rounded"
          value="{{.Email}}"
          {{if not .Email}}autofocus{{end}}
        />
      </div>
      <div class="py-2">
        <label for="password" class="text-sm font-semibold text-gray-800">
          Password
        </label>
        <input
          name="password"
          id="password"
          type="password"
          placeholder="Password"
          required
          class="w-full px-3 py-2 border border-gray-300 placeholder-gray-500
            text-gray-800 rounded"
          {{if .Email}}autofocus{{end}}
        />
      </div>
      <div class="py-4">
        <button class="w-full py-4 px-2 bg-indigo-600 hover:bg-indigo-700
          text-white rounded font-bold text-lg">
          Sign in
```

```
        </button>
      </div>
      <div class="py-2 w-full flex justify-between">
        <p class="text-xs text-gray-500">
          Need an account?
          <a href="/signup" class="underline">Sign up</a>
        </p>
        <p class="text-xs text-gray-500">
          <a href="/reset-pw" class="underline">Forgot your password?</a>
        </p>
      </div>
    </form>
  </div>
</div>
{{template "footer" .}}
```

While we are editing template files, we never correctly linked to the sign in page from our navbar, so we can open **tailwind.gohtml** and update that HTML as well.

```
<a href="/signin">Sign in</a>
```

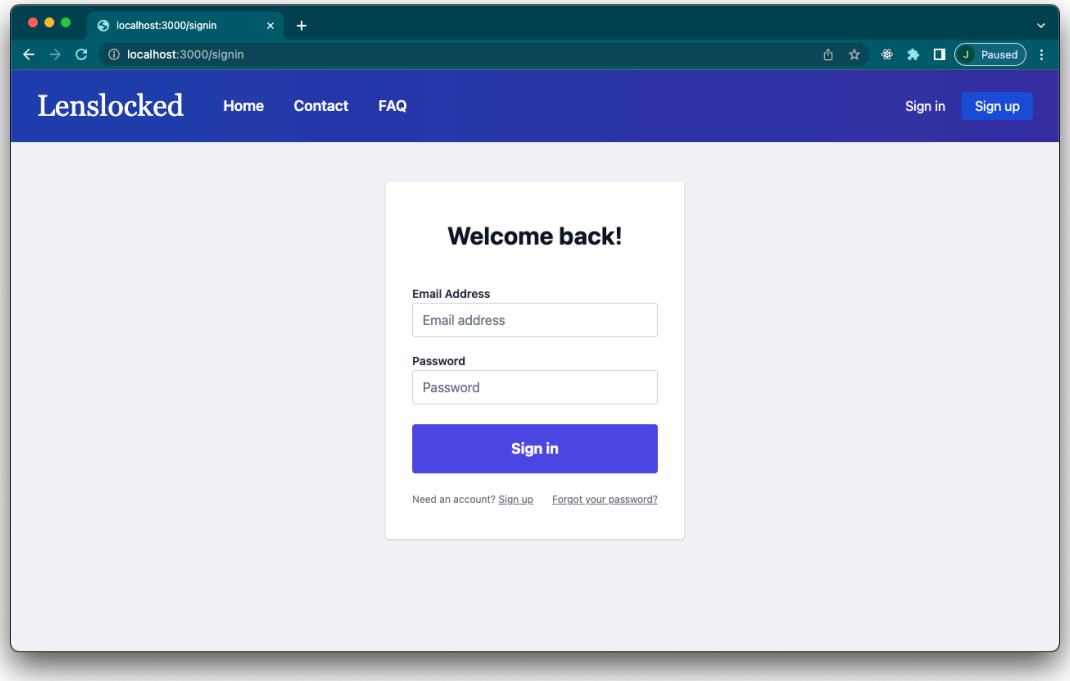
Next, we need to parse the template and set everything up. We do this inside **main.go** with the following code. Be sure to keep it next to the other line where **usersC** gets a template assigned so we can quickly find where all the user templates are setup in the future.

```
usersC.Templates.SignIn = views.Must(views.ParseFS(
  templates.FS, "signin.gohtml", "tailwind.gohtml"))
```

Lastly, we need to set the route for this page inside **main.go**. Once again, place this near the other routes for the **usersC** variable so we can easily find them all in the future.

```
r.Get("/signin", usersC.SignIn)
```

Restart the server, reload the page, and verify everything looks good. We should now be able to click the "Sign in" link on the navigation bar and be taken to the `/signin` path. When that page loads we should see a sign in page like the one below.



Submitting the form won't work yet, but we will address that in a future lesson.

Source Code

- Source Code - see the final source code for this lesson.
- Diff - see the changes made in this lesson.

12.8 Authenticate Users

If we want to log users in, we need a way to authenticate a user. This sounds like logic related to our models, so we will be adding it to the `UserService` inside of our `models` package.

```
code models/user.go
```

Similar to creating a user, we will need to know a users email address and password to authenticate them. We need the email address so we can lookup a user with the provided email address, and we need the password so we can compare it with the password hash to determine if it is correct.

When we successfully authenticate a user we will return the user object we retrieved from our database, but if there is an issue we want to return an error, so our method will also have two return values.

We will call our new method `Authenticate` which gives us the following starting point.

```
func (us UserService) Authenticate(email, password string) (*User, error) {
  // TODO: Complete this
}
```

The first thing we need to do inside the `Authenticate` method is to look up a user, but to do this we need to make sure we have a lowercase email address. Otherwise we might search for a user with the email `BOB@BOB.com` when it is saved in our database as `bob@bob.com`. While converting the email address to lowercase we can go ahead and create a variable of the `User` type and start assigning its fields.

```
email = strings.ToLower(email)
user := User{
  Email: email,
}
```

Now that we have the email address normalized, we can query for a user with that email address.

Aside 12.7. What is data normalization?

Normalizing data is the process of cleaning data so it is the same every time. For instance, when we convert the email address to lowercase we are normalizing it by converting every email address to a similar format.

Another common example of normalizing data might be stripping out any special characters from phone numbers. That way we can allow users to enter numbers like `+1(123)-456-7890` and `12223334444` and still compare them to see if they are equivalent.

A large part of the logic inside our models package will end up being normalization, as keeping our data in a consistent format makes everything easier when we need to use the data for queries or anything else.

If we find a user we want to retrieve their ID and password hash from the database, and if we don't find a user we need to return an error because we can't possibly authenticate a user who doesn't exist. Putting that all together gives us the following query and scan code.

```
row := us.DB.QueryRow(`
  SELECT id, password_hash
  FROM users WHERE email=$1`, email)
err := row.Scan(&user.ID, &user.PasswordHash)
```

```
if err != nil {
  return nil, fmt.Errorf("authenticate: %w", err)
}
```

Now that our `user` variable is populated with the correct information we can verify the password. We do this using the code we wrote when building the bcrypt CLI, and we return an error if the password isn't correct.

Long term we might want to return a specific errors for not finding a user and for passwords that cannot be validated, as this would allow us to provide the user with a more user-friendly error message, but for now any error will do.

```
err = bcrypt.CompareHashAndPassword([]byte(user.PasswordHash), []byte(password))
if err != nil {
  return nil, fmt.Errorf("authenticate: %w", err)
}
```

Finally, if there were no issues validating the user's password we can return the user and a `nil` error.

```
return &user, nil
```

Source Code

- Source Code - see the final source code for this lesson.
- Diff - see the changes made in this lesson.

12.9 Process Sign In Attempts

Our UserService is now able to authenticate a user, and we have a sign in page that renders the form. The next step we is to connect the two by creating an

HTTP handler function that will process the sign in form and use the UserService to authenticate the sign in attempt.

Open **controllers/users.go** and let's start by declaring a **ProcessSignIn** method that parses our incoming form values.

```
func (u Users) ProcessSignIn(w http.ResponseWriter, r *http.Request) {
        var data struct {
                Email    string
                Password string
        }
        data.Email = r.FormValue("email")
        data.Password = r.FormValue("password")
}
```

After we have the data from the form we want to validate it. To do this we call the **Authenticate** method on the **UserService**. We will also check for errors and print out a "User authenticated" message if the email and password are valid.

```
user, err := u.UserService.Authenticate(data.Email, data.Password)
if err != nil {
        fmt.Println(err)
        http.Error(w, "Something went wrong.", http.StatusInternalServerError)
        return
}
fmt.Fprintf(w, "User authenticated: %+v", user)
```

To test this we need to add it to our routes. Open **main.go** and add the following route inside the **main()** function.

```
r.Post("/signin", usersC.ProcessSignIn)
```

We can now restart our server and test out the sign in page. If we enter a valid username and password we should see a message saying we were authenticated. If we enter an invalid password or email address we should see an

internal server error, and our web application should print out a more detailed error in the terminal where it is running.

Our error handling is once again not ideal because we don't output very user-friendly error messages, and when a user signs in we don't remember who they are just yet, but we will improve those aspects of our sign in page as we progress through the course.

Source Code

- Source Code - see the final source code for this lesson.
- Diff - see the changes made in this lesson.

Chapter 13

Remembering Users with Cookies

13.1 Stateless Servers

There are a variety of ways to interact with other servers. Chances are you have seen or heard of some of the common protocols like HTTP, SSH, and FTP.

When we open our browser and visit https://www.google.com/ we are using HTTPS. This is basically the same as HTTP, but the `S` means it is using a secure connection.

If we open our terminal and type:

```
ssh root@ourserver.com
```

We are connecting to our server using the SSH protocol.

While all of these protocols are used to communicate with another server, how the communication happens, whether or not a connection stays open, and many

other details will vary between each protocol. For instance, when our web browser uses HTTP to fetch a website, it will send a request to a server, get a response back, and then the connection between the two is terminated. On the other hand, if we were to connect to a server with a protocol like SSH that connection would remain active until we close it.

A stateful protocol is a protocol where the server remembers who the user is and, most of the time, the connection to the server remains active between interactions. SSH is an example of a stateful connection. If we were to use SSH to connect to a server, it will look and feel like we are running a terminal session on our local computer because the server remembers what directory we are in, what commands we previously ran, and other aspects of our state.

A stateless protocol is one where the server doesn't retain the state of each client. When we connect to our Go web server using HTTP, it is a stateless protocol. As a result, our server does not need to know that the user was previously on the home page or any other information about what the user previously did to respond to a web request.

There are pros and cons to the approach used by each protocol. For instance, HTTP being a stateless protocol can make it easier to scale in comparison to a protocol like SSH because we can use multiple servers to respond to web requests.

Aside 13.1. HTTP/2 muddles this a bit

While HTTP is generally considered to be a stateless protocol, HTTP/2 does introduce some stateful features so deciding whether the protocol is stateless or stateful can be debated. The way an end user interacts with HTTP/2 still feels stateless, so for our purposes it can be considered a stateless protocol.

If HTTP is stateless, how can we remember who a user is after they log in?

With a stateful protocol we could keep an active connection and remember that the user has been authenticated, but with HTTP we need to instead include information about who the user is inside of every web request. The most common way to do this is through the use of cookies.

Cookies are data stored on our computer by our web browser. This happens when we make a web request and the server sends back a `set-cookie` header. This tells our browser that it should create a cookie to save some information. When we make future web requests to that server, our browser knows to include those cookies as headers in the web request. The end result is that servers responding to HTTP traffic do not need to manage state and can instead rely on the web request to contain cookies with that state.

Aside 13.2. Stateful vs stateless debate

As I mentioned earlier, whether or not HTTP is stateless can be debated, and cookies are another reason why many would argue it is not stateless. For our purposes we are considering a protocol to be stateless if the server does not need to retain and manage state information and can instead expect the information to be provided in a web request.

Cookies are stored on our computer, which also means that we can delete and edit those cookies. Doing so might log us out of a website or cause issues if we edit the cookie with an invalid value, but it is possible to alter our cookies.

To protect the data stored in cookies, there are a variety of rules that we can both set and customize when creating cookies. For instance, cookies will only be sent to the domain where they were created, and we can limit this even further by telling a browser to only include cookies when making requests to particular paths on that domain. As a result, we do not need to worry about cookies created by `calhoun.io` being sent to other servers and leaking user information. We will learn more about this throughout this section.

Aside 13.3. Tracking users with cookies

Over time companies have learned to abuse cookies to track users across multiple websites. The basic way this works is that a website like `abc.com` will add a "Login with Facebook" button to their website. When this button loads, it will make a request to `facebook.com` for JavaScript and other assets. While the request to `facebook.com` will only send the user's Facebook-related cookies, those requests can also contain information about what site the user is currently visiting. eg the JavaScript loaded for the login button could include this information. The end result is companies like Facebook are able to track what sites their users are visiting without them explicitly opting in to this type of behavior.

While cookies have received a fairly bad reputation due to the widespread use of tracking, most websites use cookies for simple, practical things like user authentication. This is how we will be using cookies - to remember who a user is after they have logged in. Before we can do this, there are a few security issues we will need to address:

- How do we validate information in cookies?
- How do we ensure cookie information isn't leaked?

In this section we will be exploring how to use cookies for authentication purposes, then in the next section we will learn about creating sessions and we will finalize our authentication system. It is not advisable to release an application into the wild before completing the next two sections, as it could lead to security issues.

13.2 Creating Cookies

Creating cookies in Go is done by using the `net/http` package. There are roughly two steps to creating a cookie:

1. Instantiate an http.Cookie, which is a type provided by the `net/http` package used to represent a cookie.
2. Calling the SetCookie function provided by the `net/http` package with both an `http.ResponseWriter` and an `http.Cookie`. This will add the `Set-Cookie` header to the response which is what tells the browser to create a cookie.

We will create a cookie in our code shortly, but for now the code below gives a basic example of what this might look like.

```
cookie := http.Cookie{
  Name:  "cookie key",
  Value: "cookie value",
}
http.SetCookie(w, &cookie)
```

When using `SetCookie`, there are two important details we need to make note of. The first is that **invalid cookies may be silently dropped** and the `SetCookie` function will not return an error. This happens because the `SetCookie` function only adds a `Set-Cookie` header to the response and does not validate the cookie data.

The second thing to note is that headers need to be set before writing to the `http.ResponseWriter`, otherwise they won't have any effect. The `SetCookie` function ultimately just creates a header, so it also needs to be called before writing to the response. This is noted in the `ResponseWriter` docs, although it might not be as obvious that it also applies to setting cookies.

> // Changing the header map after a call to WriteHeader (or // Write) has no effect unless the modified headers are // trailers.

In an ideal world any code that relies upon cookies being set properly will have a test to help catch these bugs, but for now we will be using our browser and manually verifying that things are working as intended.

Let's add a cookie to our application so we can see everything in action. Open **`controllers/users.go`** and add the following code to the **`ProcessSignIn`** method.

```
func (u Users) ProcessSignIn(w http.ResponseWriter, r *http.Request) {
        // ...

  // Add this code AFTER a user has been authenticated, but BEFORE any
  // writes to the ResponseWriter!
        cookie := http.Cookie{
                Name:  "email",
                Value: user.Email,
                Path:  "/",
        }
        http.SetCookie(w, &cookie)

  fmt.Fprintf(w, "User account authenticated: %v", user)
}
```

Warning: *This is not a secure way to handle authentication, as we will see later.*

Now if we were to restart our server and log in, it would create a cookie named **`email`** with a value equal to the email address we signed in with. In the next lesson we will see how to use Chrome to view the cookie and even edit it.

At their core cookies are just key-value pairs, but there are other options we can configure to help dictate when cookies are sent to our server, who has access to the cookie, and more. For instance, we want a cookie to be accessible from any page on our website, so we provided a **`Path`** value of **`/`** which maps to any path on our website.

```
cookie := http.Cookie{
  Name:  "email",
  Value: user.Email,
  Path:  "/",
}
```

On the other hand, if we only want the cookie available at specific pages we can use a value like `/app` to specify the path.

```
cookie := http.Cookie{
  Name:  "email",
  Value: user.Email,
  Path:  "/app",
}
```

With the `/app` value, the cookie would be available on paths like the ones below.

- `/app`
- `/app/`
- `/app/widgets/1`

The cookie will NOT be included for paths like the following:

- `/`
- `/application`
- `/docs/`

Conceptually it helps to think of the path as a directory, and the cookie will be available to any requests where the path matches that directory or a subdirectory. That is why the `/application` path does not have access to the cookie with a `Path: "/app"` attribute despite seeming to match the path prefix.

For our application cookies need to be accessed site-wide, so using the `/` path is a reasonable option to default to.

If we were to dive into the `http.Cookie` type we would find several other options in addition to the `Path`.

```
// A Cookie represents an HTTP cookie as sent in the Set-Cookie header of an
// HTTP response or the Cookie header of an HTTP request.
//
// See https://tools.ietf.org/html/rfc6265 for details.
type Cookie struct {
        Name  string
        Value string

        Path       string    // optional
        Domain     string    // optional
        Expires    time.Time // optional
        RawExpires string    // for reading cookies only

        // MaxAge=0 means no 'Max-Age' attribute specified.
        // MaxAge<0 means delete cookie now, equivalently 'Max-Age: 0'
        // MaxAge>0 means Max-Age attribute present and given in seconds
        MaxAge   int
        Secure   bool
        HttpOnly bool
        SameSite SameSite
        Raw      string
        Unparsed []string // Raw text of unparsed attribute-value pairs
}
```

We will examine some of these as we progress through the course, but for the curious readers these are all standard for cookies across the web and are not specific to Go, so resources like the Mozilla docs are a great place to learn more about what each option does.

In the next few lessons we will examine the cookie our sign in page now produces, but please keep in mind that the cookie we created is not secure and

should not be used as a way to authenticate users. This will become very clear in the next lesson when we see how easy it is for a Chrome extension to edit the email address stored in our cookie. As we progress through the course we will learn how to secure our cookies and build a secure authentication system, but for now the cookie we have created is to be used to explore how cookies work.

Source Code

- Source Code - see the final source code for this lesson.
- Diff - see the changes made in this lesson.

13.3 Viewing Cookies with Chrome

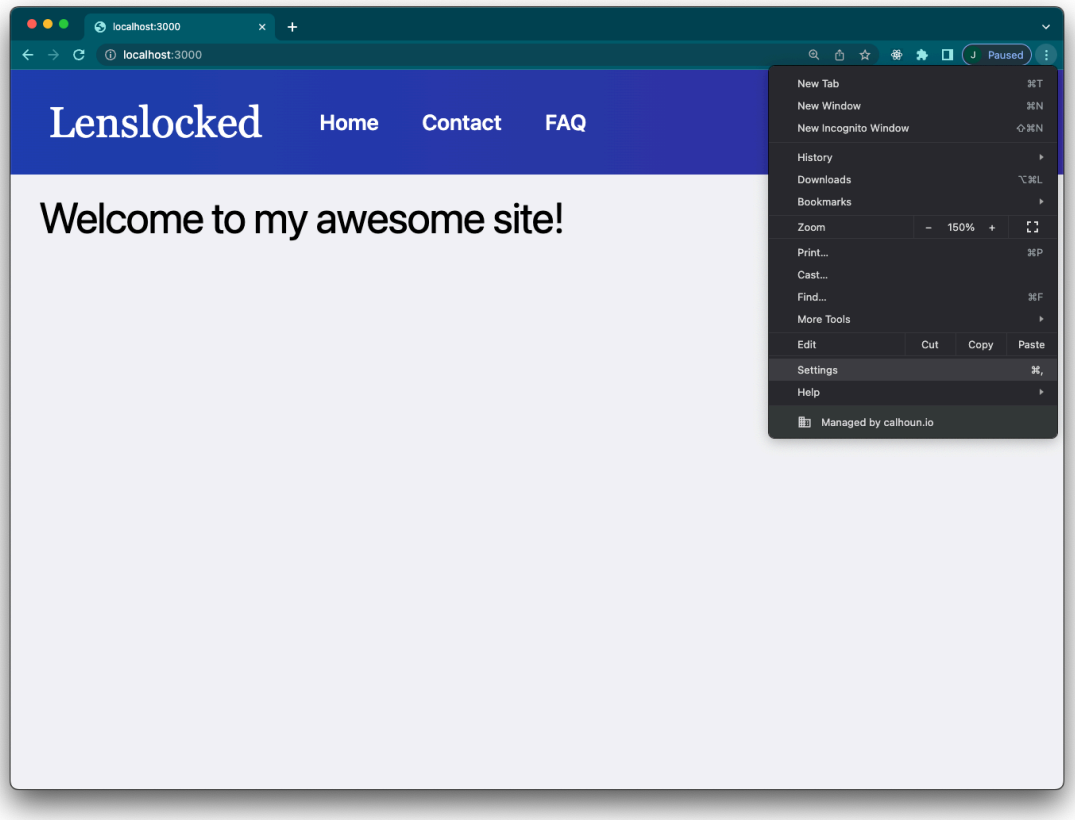
We need a cookie to proceed with this lesson, so be sure to restart the server and then sign in using the updated sign in page. With our new cookie code in place, this will result in a cookie being set that we can then proceed to examine in this lesson.

Aside 13.4. Not using Chrome?

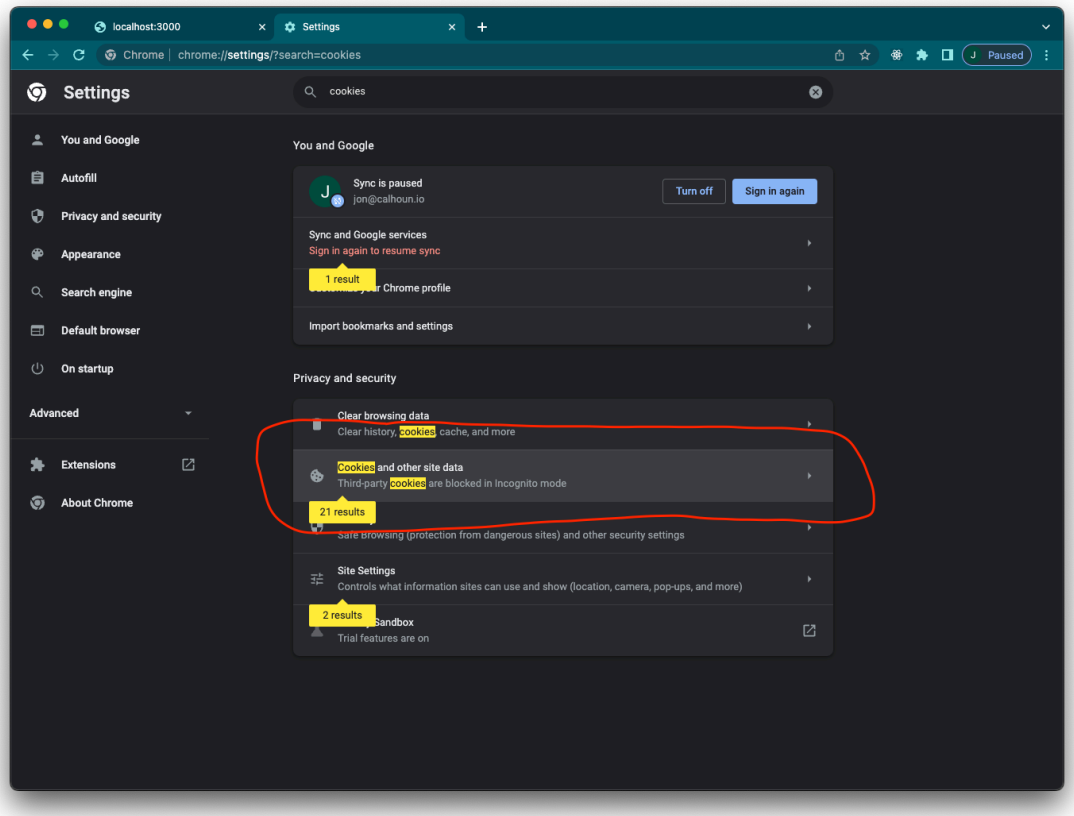
The following lesson will walk through looking at a cookie inside of Google Chrome. If you are using another browser, or a different version of Chrome, the steps might be different, but most browsers should have a way to view cookies. Feel free to use whatever browser you prefer.

The main goals of this lessons are to verify that we have set a cookie, and to see that those cookies can be updated freely by end users of our application. As a result, it isn't absolutely necessary to follow along with your own browser, but it is highly recommend to try so you can verify that your cookie was properly set.

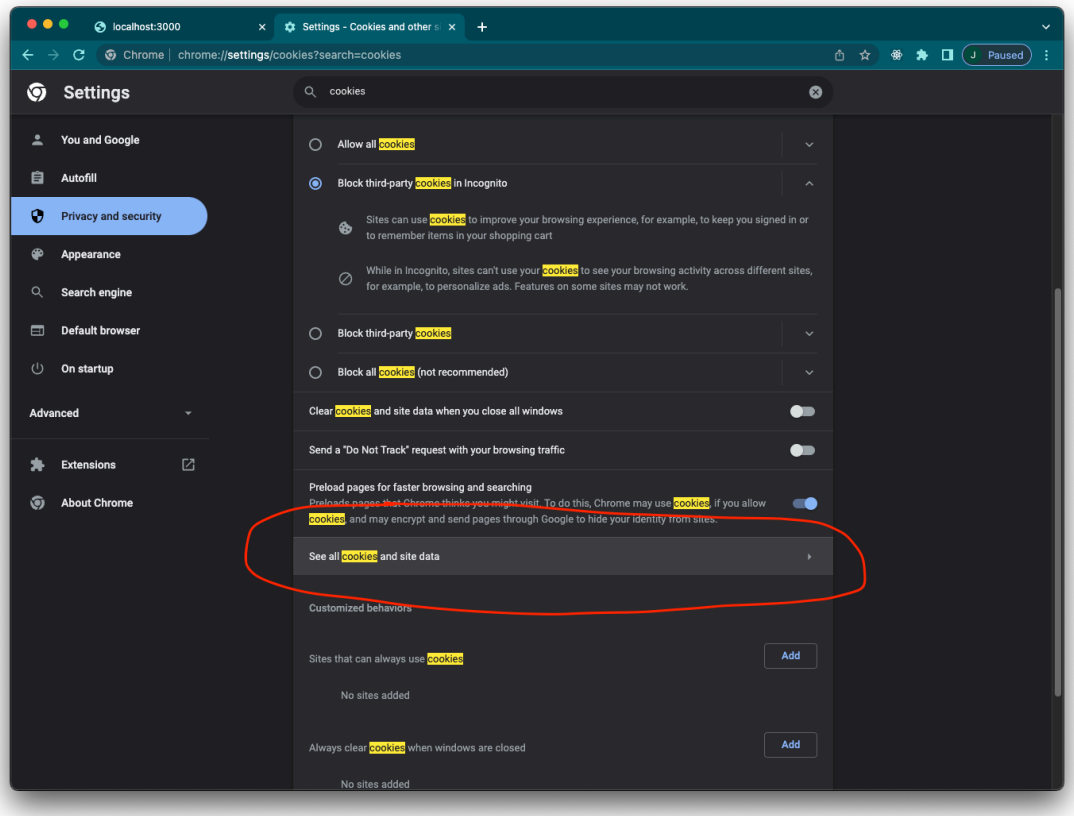
The first thing we want to do in Chrome is to open the settings. We can do this by clicking the three dots on the top right hand side of the browser window, then selecting the settings option.



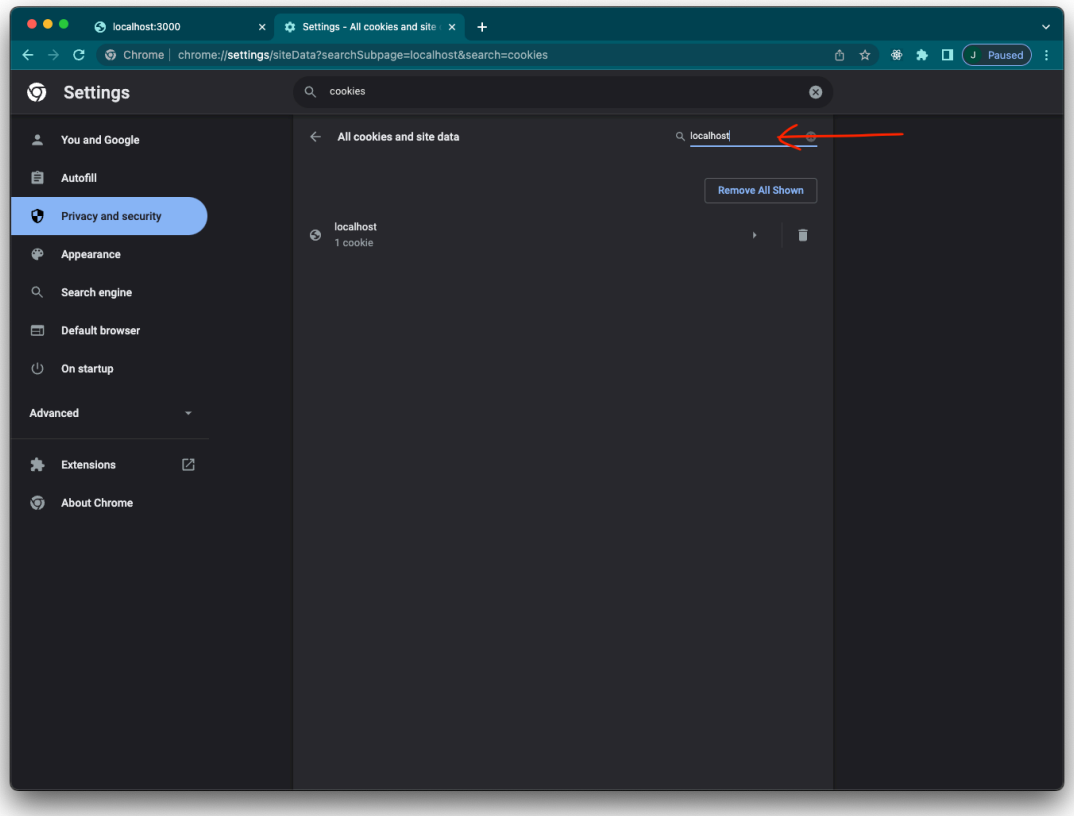
Selecting the settings option will open a new tab with Chrome settings. From here the easiest way to proceed is to type “cookies” into the settings search bar.



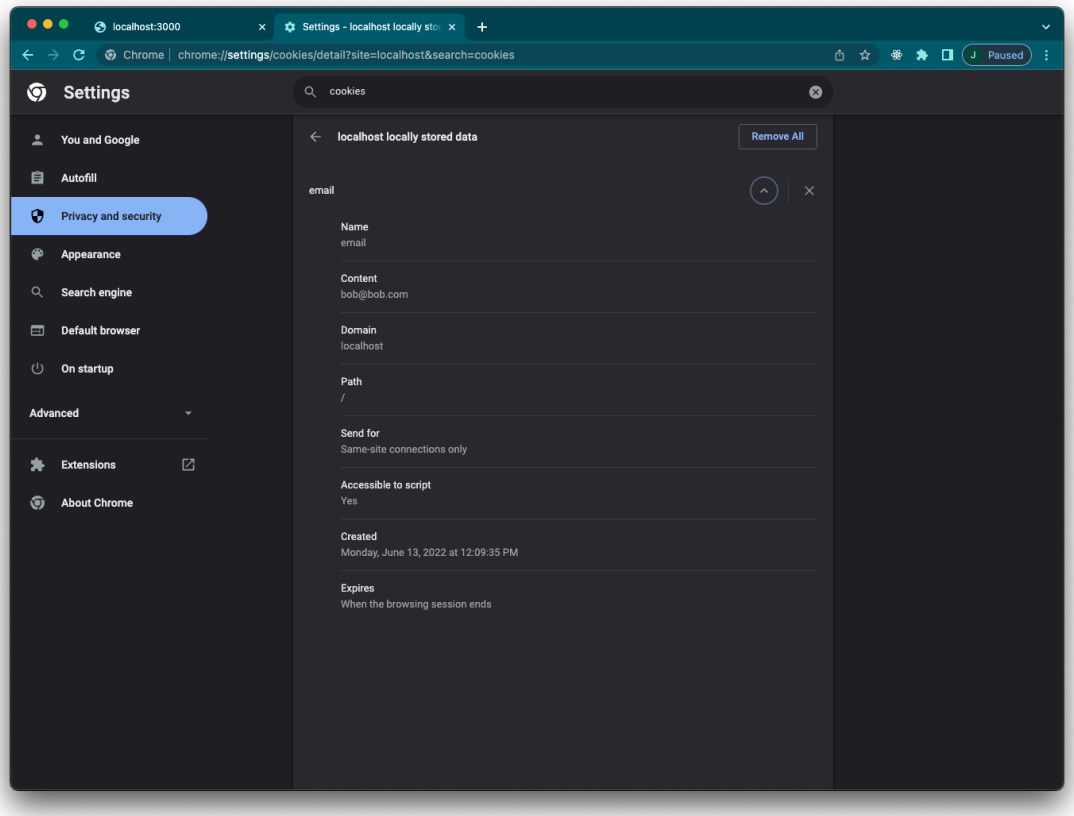
Next, select the “Cookies and other site data” option. Once on this page scroll down to the “See all cookies and site data” option.



At this point we should see a list of all cookies stored in our browser. We can also search for specific cookies using the smaller search bar at the top right. Use that search bar to find cookies that match the “localhost” query.

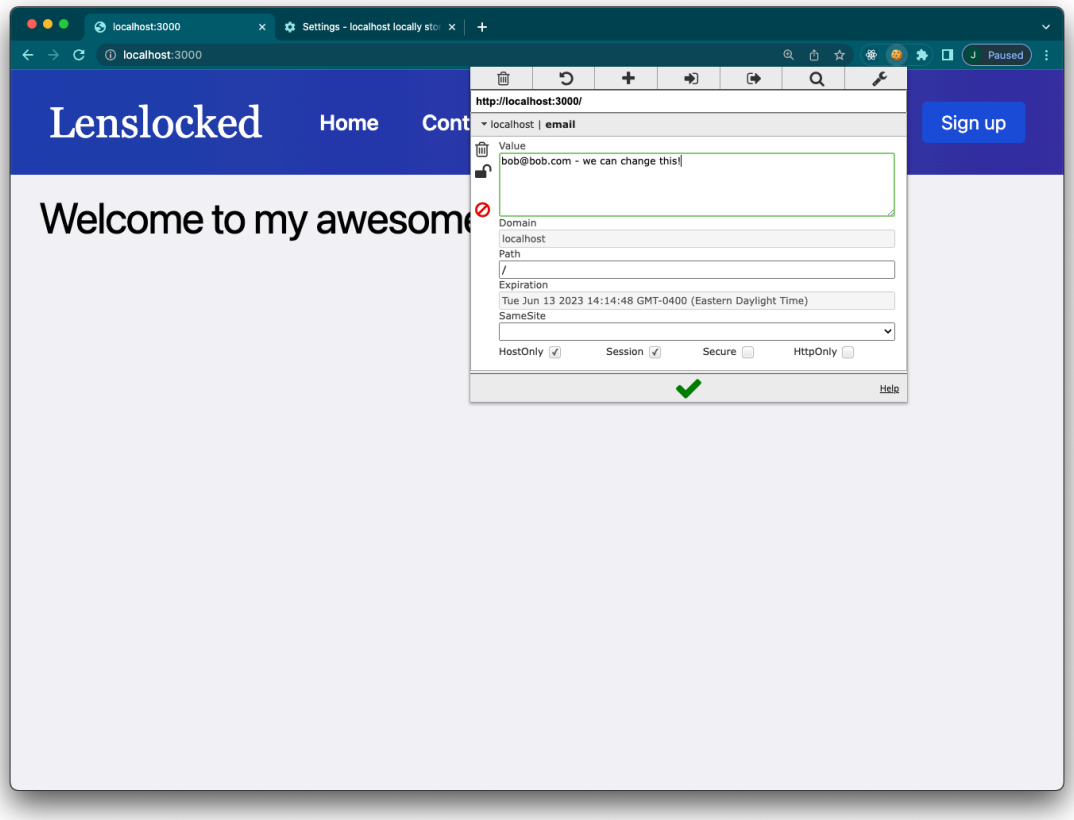


From here we can select the correct domain, `localhost`, and then select the cookie we want for that domain - `email`. We can then expand that cookie to see the information stored.



With this we can view all the information our browser stores about the cookie, as well as its contents. On top of the information that we set, we can also see information we didn't set. These are all default values that we can customize if we need to.

Chrome doesn't provide us with a way to edit cookies natively, but we can also install an extension that lets us do this. One such extension is the EditThisCookie extension. With it installed we can visit our website, then open the extension and not only see cookies related to the site, but we can also edit the cookie data.



With an extension like EditThisCookie we could theoretically change the value of the `email` cookie to instead be another user's email address. For this reason, our current cookie isn't a secure way to authenticate users, as it is far too easy for someone to impersonate another user.

In the next lesson we will see how to view cookies via Go code, but after that we will spend some time learning how to secure our cookies from both edits as well as other security concerns.

13.4 Viewing Cookies with Go

Now that we have seen a cookie stored locally on our computer, let's look at how we can read a cookie when processing a web request. To do this we will add a new HTTP handler to our application that reads the `email` cookie we created.

Our HTTP handler function is going to be in the users controller, and it is going to show us who the current user is. The handler will work by reading the `email` cookie and printing out its value. While this isn't very secure, it is a good starting point for learning how to read cookies with Go code.

Open the users controller source file.

```
code controllers/users.go
```

Add the following method to our `Users` type.

```
func (u Users) CurrentUser(w http.ResponseWriter, r *http.Request) {
        email, err := r.Cookie("email")
        if err != nil {
                fmt.Fprint(w, "The email cookie could not be read.")
                return
        }
        fmt.Fprintf(w, "Email cookie: %s\n", email.Value)
}
```

As we learned earlier, cookies are simply headers included in a web request, so it makes sense that the `http.Request` would have a way for us to access cookies. In this code we use the Cookie method provided by `http.Request` to look up the cookie. We need to provide a name of the cookie we want, which will match the `Name` field used when we created the `http.Cookie`.

Before using the cookie we need to make sure there weren't any errors. For instance, the http.ErrNoCookie error is returned when no cookies could be found

with the provided name. If an error occurred, we print out a response stating that the cookie could not be read and use **`return`** to exit our handler function.

If there are no errors we can use the cookie we retrieved from the request. The **`email`** variable has the type **`http.Cookie`**, so calling **`email.Value`** will give us the contents of the cookie and we can write that out to the response writer using **`fmt.Fprintf`**.

We also need to tell our router about the new handler. We can do this by opening **`main.go`** and adding the following to our router.

```
r.Get("/users/me", usersC.CurrentUser)
```

Typically the path for a resource would be something like **`/users/:id`**, but in this case we don't necessarily know the user ID, but we do know that we want information about the user account we are signed into. The path **`/users/me`** is a somewhat common pattern used for this information. It is more commonly seen in APIs that power a JavaScript frontend, but it also suits our needs quite well.

With the route in place we can restart our web server and navigate to the <localhost:3000//users/me> page to view the current user. Once there we should see information similar to that below.

```
Email cookie: jon@calhoun.io
```

We have heard on numerous occasions that cookies are headers, so let's do one final thing by proceeding - print out the headers included in the **`http.Request`** to see if we can find our cookie header. Add the following to the end of the **`CurrentUser`** method we just created.

```
fmt.Fprintf(w, "Headers: %+v\n", r.Header)
```

Restart the web server and navigate back to the <localhost:3000/users/me> page. We will now see quite a few headers being printed out, but what we are looking for is a header that looks it could contain our cookie information.

```
Headers: map[... Cookie:[email=jon@calhoun.io] ...]
```

The headers are stored as a map of keys and values, and our cookies are stored under the `Cookie` key. If we had multiple cookies set we would see them all here in the headers.

We won't be including this exact code when we ship our application to production, but we are going to keep it around for now as a way of verifying that our cookie has the value we expect.

Source Code

- Source Code - see the final source code for this lesson.
- Diff - see the changes made in this lesson.

13.5 Securing Cookies from XSS

Earlier in the course we learned about XSS and how it can be used to execute arbitrary JavaScript if not prevented. Now that we are introducing cookies, it is even more important to prevent XSS attacks, as cookies will be accessible to JavaScript by default. Rather than being annoyed with `alert()` messages, our users could have their login cookies stolen if we don't prevent an XSS attack.

We are using the **`html/template`** package to prevent cross-site scripting attacks, but with any package there is always the possibility of an unknown vulnerability. At the moment our cookies are accessible via JavaScript, and we can verify this with the following steps.

1. Open your browser and head to
2. Open your browsers developer tools
3. Run the following in the console where you can type in JavaScript:

```
console.log(document.cookie);
```

In many cases this is what we might want, but our application won't be using JavaScript and cookies to make web requests to our server. As a result, we do not need to give any JavaScript code access to our cookies.

We can disable JavaScript access to cookies by setting the **`HttpOnly`** field on an **`http.Cookie`**. First, open the users controller.

```
code controllers/users.go
```

Navigate to the **`ProcessSignIn`** function where we create our cookie and add the **`HttpOnly`** field. Set it to **`true`**. True means that this cookie should only be accessible via the browser making HTTP request, and not via JavaScript initiated requests. False is the default, and means JavaScript can access the cookie.

```
cookie := http.Cookie{
  Name:     "email",
  Value:    user.Email,
  Path:     "/",
  HttpOnly: true,
}
```

We can verify this is working by restarting our server, signing in again so we have a new cookie set with the correct settings, and finally examining the `document.cookie` in our developer tools console. We should no longer see the `email` cookie when accessing cookies via JavaScript.

We can also verify this by looking up the cookie in our browser like we saw earlier in the course. In Chrome there will be information similar to that listed below for the `email` cookie.

```
Accessible to script: No (HttpOnly)
```

While it may be tempting to skip the `HttpOnly` setting, I would advise against it. When a website gets hacked, it is rarely due to a single big mistake. Instead, it is typically the combination of several minor mistakes that all seem insignificant on their own, but when put together lead to the perfect storm. There are many examples of this on the internet.

Even if our app may use JavaScript to make web requests in the future, it is much safer to err on the side of caution and disable JavaScript access until it is actually needed. Another benefit of this approach is that we can make sure we only enable it on cookies necessary for our JavaScript to work at that time.

Source Code

- Source Code - see the final source code for this lesson.
- Diff - see the changes made in this lesson.

13.6 Cookie Theft

The next attack vector we are going to discuss is cookie theft. There are two potential ways to steal cookies, and we will discuss both briefly.

The first way to steal cookies is via packet sniffing. Packet sniffing occurs when an attacker manages to read web requests before they make their way to the intended server. For instance, if you are on a public wifi network, an attacker could be monitoring WiFi traffic trying to read it. One of the trickiest parts of this attack vector is that victims often are unaware of the packet sniffing even occurring because everything continues to operate normally on their end.

The second way to steal cookies is via physical access to a device. We saw that cookies are stored on a local computer and we can even view them in the Chrome settings. Likewise, an attacker with access to a machine could copy cookies from the machine.

Packet sniffing is preventable through the use of SSL/TLS. If we visit a website and it has `https` on the front of the URL, it is a secure website using TLS. On the other hand, websites with `http` are not secure from packet sniffing.

In the past using `https` was a bit more complicated and often cost additional money. These days it is relatively easy to secure a website using free services like Let's Encrypt and deploying with tools like Caddy Server. When we are getting ready to deploy, we will use Caddy to ensure our website is secure as well, but if ever considering other deployment options, it is important to make sure our web application is using an SSL certificate.

Aside 13.5. Firesheep

Eric Butler shed a lot more light on the issue of packet sniffing when he published a Firefox extension named Firesheep which demonstrates just how easy this attack is to administer. With the extension installed, anyone could connect to a busy WiFi network and the extension would sniff for anyone logging into sites that Firesheep recognized. Sites like Facebook, Twitter, and more. The person with Firesheep installed could then easily log into the service as that user.

The extension is likely useless at this time since every major website tends to use HTTPS, but before extensions like this it wasn't uncommon for them not to.

Physical theft is a bit trickier to prevent, but luckily it also isn't as easy for attackers since it requires physical proximity. One of the key ways to help mitigate this type of attack is ensuring that users have a way to invalidate all past login cookies. Our application will do this by default anytime the user logs in on a new machine, but if we were to change our system to allow multiple computers to be logged into the same account at the same time we would want to provide a way to invalidate old sessions. This would be similar to how TV streaming services give users the option to log out any device that is logged into their account.

There are other ways to help mitigate this type of attack such as using expirations on sessions, linking a user's IP address to a specific session, and more, but each of these can also have drawbacks. For instance, linking a user's IP to a session might make for a poor mobile experience if a user is frequently changing WiFi networks around their office, as they would need to sign in quite frequently.

In many cases, these extra measures aren't needed unless we are dealing with a very specific type of application, such as one that deals with a user's money, or if we are an email provider where compromising an account could lead to all that user's other accounts being compromised through password resets.

13.7 CSRF Attacks

When we make web requests to our web server, our browser automatically includes related cookies. That means when we submit a form to our bank telling it we want to transfer money, that form includes cookies with our authentication information. This is how the bank validates that we are who we say we are.

Cross-site request forgery, or CSRF, is an attack that occurs when a website creates a fake link, form, or even an image tag and tries to take advantage of our browser automatically including those cookies. Let's look at a few ways

this could happen.

The first is with a link. Imagine we were sitting at home and we received an email about a $500 charge from our bank that we did not make. Natural instinct is to immediately visit the website and see what is going on. What we might not realize is the email wasn't actually sent from our bank and is a fake.

When we click a link in the email it sends us to **yourbank.evilsite.com**. This isn't actually our bank's website, but it is designed to be close enough that we might not realize it while being so concerned about the $500 charge. This can be especially effective against the less tech-savy.

On the website we see a page that looks like our bank, and on it there is a link that says, "Dispute the charge". Our natural instinct is to click the link, but in reality that link might have HTML like the following:

```
<a href="https://yourbank.com/transfer?recipient=<attacker_account_number>&amount=500">Dispute
```

When we click this link, it will take us to our bank's real website at the **/transfer** path. This request will also include all of our cookies for our bank website, which potentially could include our authentication cookie.

While an attacker won't have access to our cookies or the response that our bank sends back to us, in many cases they may not need those details. Imagine if our bank initiated a transfer when users visited this link without any confirmation. By clicking the dispute link on the attackers website, we have inadvertently transferred money to their account!

Aside 13.6. GET request shouldn't perform actions

There are a number of reasons why actions should be reserved for POST, PUT, DELETE, and similar HTTP methods, but CSRF attacks are among the best rea-

sons why the GET method should never be used for actions that alter data. Otherwise CSRF attacks are nearly impossible to prevent, as we will see when we discuss prevention options.

Sneakier websites might not even provide a link for us to click. Instead, they will include image tags on their site that lead to your bank's domain.

```
<img src="https://yourbank.com/transfer?recipient=evil&amount=500"/>
```

When we visit the website, our browser will attempt to load the image which means it will send a web request to that URL and that web request will include our cookies. Yikes!

Now let's imagine that we are being smart and we don't perform actions on a GET request within our server. What does a CSRF attack look like for POST, PUT, or other HTTP methods?

Once again imagine that we somehow found our way to **`evilsite.com`**. On the website we instead see what looks like a button to dispute the charge, but in the HTML we may find something like the HTML below.

```
<form action="https://yourbank.com/transfer" method="POST">
  <input type="hidden" name="recipient" value="attacker@evilsite.com">
  <input type="hidden" name="amount" value=500>
  <button type="submit">Dispute the charge</button>
</form>
```

Oftentimes these forms will be styled to look even less like a form and more like a regular link, but for now we are just looking at plain HTML. Inside this HTML we have a form that once again gets sent to our bank's website, but this time the recipient and the amounts are included via hidden input fields.

Once again, **evilsite.com** doesn't need to be able to read the response of the form submission for it to work. When making fake forms like this, the attacker will never read the response from the server, but they can still cause damage simply by getting users to submit the form.

When we submit a form like this, even if it originates from **evilsite.com**, our browser will include our cookies for the website we are submitting the form to. Our browser will include our cookies, and it would look to our bank as if we submitting their money transfer form, not a fake from from an attacker's website. In short, without CSRF protection this could lead to real issues.

To prevent this type of attack, we will use what are known CSRF tokens. The basic idea behind a CSRF token is to first generate a random string for each client.

```
csrfToken := "some-random-string"
```

The CSRF token needs to be persisted between requests, so it is often stored in a cookie. This allows our server to read it in future requests.

Once a CSRF token has been set, our web application then uses it as a hidden field in all HTML forms.

```
<form ...>
  <input type="hidden" name="csrf" value="some-random-string">
  ...
</form>
```

When a form is ever submitted, the first thing our application will do is look at the **csrf** input value. If it matches the CSRF token we assigned to that user, we know it is a valid form that we created. If the CSRF tokens don't match, or if it isn't present in the form, we know it is an invalid form and we do not process it.

This technique works because external websites like `evilsite.com` cannot read our cookies for other websites, nor can they read the responses that those websites send to us. As a result, `evilsite.com` can cause us to submit a fake form, but they cannot read what the proper CSRF token should be and thus cannot fake it.

Aside 13.7. CSRF with JavaScript Frontends

When building a JavaScript based frontend, CSRF tokens are often included as a header rather than a hidden form value. This ultimately has the same effect, as the value can still be read and verified before parsing any POST, PUT, or DELETE web requests.

13.8 CSRF Middleware

We are going to implement CSRF protection using the gorilla/csrf package. `gorilla/csrf` is a small, well tested library that provides all the options we need.

All of the gorilla packages were archived on December, 2022. What this means is that `gorilla/csrf` and other packages will no longer be updated moving forward. In some cases this might mean that a library should no longer be used, but in the case of `gorilla/csrf`, the package has been stable for quite some time. If we look at the commit history, we will see that since 2019 there hasn't been a single code change to the package that would require a developer to update their dependencies.

Due to its stability, the `gorilla/csrf` library is still a good fit for our application. If that changes in the future we will update our application later in the course.

Aside 13.8. Why don't we write our own CSRF protection?

I believe it is important to understand how our application is kept secure; without understanding all the measures being taken, it is easy to inadvertently introduce vulnerabilities into an application. As a result, we tend to spend a large portion of this course learning about hashing passwords, securing cookies, and more. We also look at the code for each of these security measures to reinforce what is being learned.

On the other hand, I want this course to be realistic. I don't want to waste hours building our own CSRF library if we would almost never opt to go this route in a real application. It would be a distraction from learning the other important parts of web development, and it would result in a final codebase that doesn't truly illustrate a real-world application. With that in mind, I opted to use a third party library for CSRF protection.

There are many factors to be considered when deciding whether or not to use a third party library. Is the library well tested? Has it been around a while, and is it stable? Will we be locked into this library, or will it be fairly easy to replace if necessary?

`gorilla/csrf` is fairly minimal, stable, well tested, and is quite easy to replace with another library. If we had opted to use a third party library for authentication, it would have been quite challenging to replace it with something custom in the future, but replacing `gorilla/csrf` with something custom would be an easy task. As a result, it is a great fit for our needs and doesn't lock us into anything long term.

Before we can use the `gorilla/csrf` package, we need to `go get` it. This will download the source code as well as update our `go.mod` and `go.sum` files to mark a new dependency.

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

It is always a good idea to reference the docs for any new library we opt to use. For `gorilla/csrf`, the docs can be found at https://github.com/gorilla/csrf. The docs include example usage and more.

The CSRF library we are using is fairly small and only provides a few functions:

- `csrf.Protect`
- `csrf.Token`
- `csrf.TemplateField`

`csrf.Protect` is a function that will handle both setting CSRF tokens for each user, and validating CSRF tokens when necessary. This is achieved through the use of HTTP middleware, which we will learn about shortly.

`csrf.Token` is a function that accepts an `http.Request` as an argument and returns the CSRF token assigned to that request. We won't be using this, as the `TemplateField` function does this plus a bit extra for us.

`csrf.TemplateField` is similar to `csrf.Token`, except instead of returning the CSRF token, it returns an HTML `<input ...>` tag that we can insert into our HTML forms. The input HTML generated will be hidden, and it will have the CSRF token set as the value, making it very easy to add the CSRF token to our HTML forms.

We are going to look at how to use `csrf.Protect` shortly, but first we need to understand middleware.

13.8.1 What is Middleware?

Middleware is a function that accepts an HTTP handler function and returns a new HTTP handler with some functionality wrapped around it. For instance, imagine that we wanted to add a timer to our HTTP handler functions. We could use the following code to create a timer middleware:

```
func TimerMiddleware(h http.HandlerFunc) http.HandlerFunc {
  return func(w http.ResponseWriter, r *http.Request) {
    start := time.Now()
    h(w, r)
    fmt.Println("Request time:", time.Since(start))
  }
}
```

Any `http.HandlerFunc` can be passed into the `TimerMiddleware` function and it will return a new HandlerFunc that records the current time, calls the original handler function, then once the original handler has finished running it prints out how much time the request took using `time.Since`.

Middleware can be useful to wrap a variety of HTTP handlers with common functionality. In the case of our CSRF prevention, the middleware is going to both handle generating our unique CSRF token for each client and storing that in a cookie, as well as checking to make sure the CSRF field is provided when forms are submitted via POST, PUT, and other HTTP requests that need CSRF protection. In short, the CSRF middleware does 90% of our work for us for CSRF protection.

Aside 13.9. Using the Timer Middleware

If we wanted to test the `TimerMiddleware` code, we could add it to our application in `main.go`, then use it to wrap one of our HTTP handler functions.

```
r.Get("/signup", TimerMiddleware(usersC.New))
```

Now if we restart our server and visit the **`/signup`** page we will see a message similar to the one below in our terminal.

```
Request time: 196.573µs
```

If we visit other pages we won't see this timer, as we only wrapped the **`usersC.New`** handler with the timer middleware.

The concept of middleware can be challenging to grasp at first because it is often the first time a developer will see a function that returns a new function of the exact same type that was passed in as an argument. The code is typically easier to grasp if we start with strings.

```
func main() {
        output := Wrap("hello")
        fmt.Println(output)
}

func Wrap(input string) string {
        return "prefix-" + input + "-suffix"
}
```

Run this code on the Go Playground: https://go.dev/play/p/LkgFNwE0ONx

When reading this code, most developers will be able to deduce that the output is going to be:

```
prefix-hello-suffix
```

From here, let's try accepting a function instead of a string.

```
func main() {
        output := Wrap(Hello)
        fmt.Println(output)
}

func Hello() string {
        return "hello"
}

func Wrap(stringer func() string) string {
        return "prefix-" + stringer() + "-suffix"
}
```

If we run this code, we will still see the same output, but inside **`Wrap`** we need to call a function to get the **`hello`** string.

We can take this a step further and have our **`Wrap`** function return a new function that produces our final string.

```
func main() {
        output := Wrap(Hello)
        fmt.Println(output)
        fmt.Println(output())
}

func Hello() string {
        return "hello"
}

func Wrap(stringer func() string) func() string {
        return func() string {
                return "prefix-" + stringer() + "-suffix"
        }
}
```

If we simply print out the **`output`** variable like we did before we will get a memory address because a function is stored in the variable, but if we call **`output()`** we will get the string we expected.

```
prefix-hello-suffix
```

In this case there isn't a benefit to using functions like we are, but imagine a situation where the `Hello` function returned a different value over time. For instance, it might be using a `Name` variable that is altered as our program runs.

```
var (
  // Warning: global variables like this is often a bad idea,
  // but I am using one to simplify this demo code.
        Name = "Jon"
)

func main() {
        output := Wrap(Hello)
        fmt.Println(output())
        Name = "Bob"
        fmt.Println(output())
}

func Hello() string {
        return "hello, " + Name
}

func Wrap(stringer func() string) func() string {
        return func() string {
                return "prefix-" + stringer() + "-suffix"
        }
}
```

Run this on the Go Playground: https://go.dev/play/p/UpPt14FPYNh

Now when we run our code our `Wrap` function applies the same prefix and suffix, but the string it gets applied to isn't determined until we call the `output()` function.

This type of chaining occurs somewhat frequently in Go, especially with HTTP handlers. By returning an `http.HandlerFunc` from our `TimerMiddlerware`, we are returning something that can be used in our router or anywhere else an HTTP handler works. We could even wrap the `TimerMiddleware` with additional middleware.

Sidenote: Gophercises has an exercise on middleware if you want to check that out as an additional resource.

Middleware is a common technique because it enables reusable code that can be applied to any HTTP handler. For instance, we can have one function that handles CSRF protection, then apply it to all of our HTTP handlers to protect our entire website. Another use we will see in this course is using middleware to authenticate users and require them to be logged in before allowing them access to restricted pages.

13.8.2 Coding the CSRF Middleware

Now that we have learned about middleware, let's add CSRF protection using the `Protect` function. Open up `main.go` where we declare our routes.

```
code main.go
```

The middleware returned from the `csrf.Protect` function will handle:

- Generating a unique CSRF token for each client
- Storing the CSRF token in a secure cookie
- Reading and validating the CSRF field from incoming form submissions
- Rejecting requests if CSRF field is invalid or missing

In short, the CSRF middleware does most of our work for us.

`csrf.Protect` takes two arguments - an auth key, and options.

```
func Protect(authKey []byte, opts ...Option) func(http.Handler) http.Handler
```

The **authKey** should be 32 bytes, and it is used to digitally sign the CSRF cookie that is created so it cannot be tampered with. Alphanumeric characters (**A-Z**, **a-z**, and **0-9**) are 1 byte each, so any 32 character string using only these characters should work for the time being. We will be using the following hard-coded auth key as we learn.

```
gFvi45R4fy5xNBlnEeZtQbfAVCYEIAUX
```

We can also pass options into the **csrf.Protect** function call. By default, **gorilla/csrf** expects our app to be using **https** to secure our traffic, but we are currently running in development without a secure domain. To address this, we need to use the **csrf.Secure** option to disable the **https** requirement. When we deploy to production we will address this as well.

Putting this all together we get the following code, which can be added inside the **main()** function and will replace the existing call to **http.ListenAndServe**.

```
func main() {
        // ... unchanged
        csrfKey := "gFvi45R4fy5xNBlnEeZtQbfAVCYEIAUX"
        csrfMw := csrf.Protect(
                []byte(csrfKey),
                // TODO: Fix this before deploying
                csrf.Secure(false),
        )
        http.ListenAndServe(":3000", csrfMw(r))
}
```

The call to **csrf.Protect()** returns a function that accepts an HTTP handler as an argument and returns a new HTTP handler with CSRF protections applied. We assign this function to the **csrfMw** variable, then we call it passing our router **r** in as an argument. This will cause CSRF protection to kick in before our router has a chance to run any code, making sure all incoming requests

pass the CSRF check before we do anything else. If CSRF protection passes, the router code is then called.

This works because `gorilla/csrf` can check the request to see if the method is a `GET` or a `POST` and decide how to proceed. Once the CSRF checks are completed the router can also see if the HTTP method is a `GET` or a `POST` and route the request as needed.

Aside 13.10. Don't hard-code sensitive keys in source code

We are currently hard-coding our auth key into our source code. Generally speaking, this is a bad idea because anyone with access to our source code also has access to keys that we use in production. A better approach is to read these variables when our application starts, allowing us to use sensitive keys in production without giving everyone access to them.

We will address this in a future lesson, and I opt to hard-code the key for now because it is much easier to understand when learning.

13.8.3 Seeing the Changes

Once our changes are in place we can restart our server and navigate to any page. After that we can look at the cookies for `localhost` like we did earlier in the course. Look for a cookie named `_gorilla_csrf`. This is our CSRF cookie, though the value we see won't be a raw CSRF token since it is digitally signed.

Now if we head to the login form at <localhost:3000/signin> and attempt to sign in we will see that submitting our form no longer works. When we submit a form we get the response:

```
Forbidden - CSRF token not found in request
```

This error means our HTML form did not have a valid CSRF token in it. That means our CSRF protection is working, and until we add the CSRF token to our forms we won't be able to submit a form. We will get to this in the next lesson.

Source Code

- Source Code - see the final source code for this lesson.
- Diff - see the changes made in this lesson.

13.9 Providing CSRF to Templates via Data

The `csrf.Protect` middleware that we added caused all of our forms to stop working. We can render the forms, but when we submit them we see an error. This happens because we do not have a CSRF token included in the form. Let's look at how to add CSRF tokens to our HTML forms to fix this issue.

In this lesson we will be passing data into our template that can be used to add the CSRF token to the form. In a future lesson we will look at adding custom functions to our templates which will make using CSRF tokens easier, but is a bit more confusing to understand and set up correctly.

Open the users controller.

```
code controllers/users.go
```

Head to the `New` method and update the code with the following.

```
func (u Users) New(w http.ResponseWriter, r *http.Request) {
        var data struct {
                Email     string
                CSRFField template.HTML
        }
        data.Email = r.FormValue("email")
        data.CSRFField = csrf.TemplateField(r)
        u.Templates.New.Execute(w, data)
}
```

gorilla/csrf has the csrf.TemplateField function, which give us the HTML for a hidden **<input>** tag that has the CSRF token for the incoming request. We assign this to the **CSRFField** field of the **data** struct so that it will be available inside of our template.

We use the **template.HTML** type because we know the value returned from **csrf.TemplateField** will be an HTML tag, and because that is the type being returned from the **csrf.TemplateField** function.

Now that we have the **CSRFField** being passed into our template, we need to update our template to use it.

```
code templates/signup.gohtml
```

In the template we add the code to render the **CSRFField**.

```
<form ...>
  <!-- Add the following div -->
  <div class="hidden">
    {{.CSRFField}}
  </div>
  <!-- The rest of the form is unchanged. -->
</form>
```

The input generated by the **gorilla/csrf** package will be hidden already, but we can add a nice hidden div around it to organize and ensure it is always hidden.

Now we can start the server, head to the signup page, and view the source code. We will find an input like the following inside our new `<div>`.

```
<input type="hidden" name="gorilla.csrf.Token" value="A6YSbCQAh6U7Hinm5rThWORzr8bpuGxD1UX7EbOiW
```

The value will vary, but the type and name should be the same.

With these changes in place, we can submit the signup form and we will no longer see an error due to CSRF protection. Be sure to restart the server, navigate to the signup page, and test that the form works.

While this approach clearly works, the downside is that we would need to include a `CSRFField` field in the data passed into every template we render. Making this work would involve updating every HTTP Handler that renders a form. Another option is to add a new function to our templates, which we will learn about in the next lesson.

Source Code

- Source Code - see the final source code for this lesson.
- Diff - see the changes made in this lesson.

13.10 Custom Template Functions

Templates have access to a variety of functions that make it possible to add basic logic to our templates. For instance, the we used the following code when improving our forms.

```
{{if not .Email}}autofocus{{end}}
```

In this code we are using the `not` function, which takes in an argument (`.Email` in our case), and then returns the boolean negation. In other words, if the email address is provided, it returns false. If the boolean address is not provided, `not` returns true.

We can also add our own functions to templates using the Funcs function and a FuncMap as an argument. Behind the scenes, a `FuncMap` is a `map[string]any` where each key is a new function being added to our template, and each value is the function to be called if that function is used in the template.

Our goal in this section is to create our own `csrfField` function that will provide all of the HTML necessary to insert a hidden CSRF input field into any form. This will allow us to use the following in our templates:

```
<form ...>
        {{csrfField}}
        <!-- Other form stuff -->
</form>
```

Let's try adding the function to our templates.

```
code views/template.go
```

Aside 13.11. Removing Unused Code

We still have the `Parse` function in our code, but we are now using `ParseFS` instead. Delete the `Parse` function so we do not accidentally use it anymore.

Add the following code to `ParseFS` to add the new `csrfField` template function. This code has a bug in it which we will explore shortly, but it is still worth coding to see how the `Funcs` function works.

```
// Warning: This is buggy code, as we will see shortly.
func ParseFS(fs fs.FS, patterns ...string) (Template, error) {
        tpl, err := template.ParseFS(fs, patterns...)
        if err != nil {
                return Template{}, fmt.Errorf("parsing template: %w", err)
        }
  tpl = tpl.Funcs(
    template.FuncMap{
                        "csrfField": func() template.HTML {
                                return `<input type="hidden" />`
                        },
                },
  )
        return Template{
                htmlTpl: tpl,
        }, nil
}
```

Next, let's try to use the **`csrfField`** function in a template.

```
code templates/signup.gohtml
```

Update the code to no longer use the **`.CSRFField`** data, and to instead use the **`csrfField`** function.

```
<form ...>
        <div class="hidden">
                {{csrfField}}
        </div>
        <!-- ... -->
</form>
```

Next we need to restart the server to test things out, but we now have an error when starting up our sever!

```
parsing template: template: signup.gohtml:9: function "csrfField" not defined
```

When adding custom functions to a template, we need to parse them a bit differently. Currently we are parsing the template first, then adding our custom functions to the parsed template. We can see this in the **ParseFS** function.

```
// First we parse the template
tpl, err := template.ParseFS(fs, patterns...)
if err != nil {
        return Template{}, fmt.Errorf("parsing template: %w", err)
}
// Then we add our custom functions
tpl = tpl.Funcs(
        template.FuncMap{
                "csrfField": func() template.HTML {
                        return `<input type="hidden" />`
                },
        },
)
```

When we parse our signup template, **csrfField** has never been defined, but we are using it inside of the template. As a result, when we try to parse the template we get an error back stating that the template uses an undefined function. To fix this, we need to first add our custom functions, then parse the template source files (the **.gohtml** files).

Let's see the code for the changes, then discuss it.

```
code views/template.go
```

Update **ParseFS** with the following code.

```
func ParseFS(fs fs.FS, patterns ...string) (Template, error) {
        tpl := template.New(patterns[0])
        tpl = tpl.Funcs(
                template.FuncMap{
                        "csrfField": func() template.HTML {
                                return `<input type="hidden" />`
                        },
                },
        )
```

```
        tpl, err := tpl.ParseFS(fs, patterns...)
        if err != nil {
                return Template{}, fmt.Errorf("parsing template: %w", err)
        }
        return Template{
                htmlTpl: tpl,
        }, nil
}
```

We call `template.New()` first because we need to create an empty `template.Template` to start with. Our template needs a name, so we pass in `patterns[0]` as the name of the template. This will roughly mimic what `template.ParseFS` was doing when we were using it previously, keeping our code from breaking.

Next, we call `tpl.Funcs` and provide our custom function. For now the `csrfField` function isn't complete, but this will help us get headed in the right direction. When we call `tpl.Funcs` it returns a new template with the functions defined, so we assign it back to the original `tpl` variable.

Finally, we call ParseFS which will use our existing template and its functions to parse files like our `signup.gohtml`. It is important to note that this method is different from the ParseFS function we were using preivously. We are now using a method provided by the `template.Template` type. After the files are parsed via `ParseFS` a new template will be returned, or an error, so we assign those return values and check for errors.

With our code updated, we are now declaring the `csrfField` function before parsing any template files. As a result, our server should now compile and start up without any errors. We can even head to the sign up page and look for our hidden input.

```
<form ...>
        <div class="hidden">
                <input type="hidden">
        </div>
        <!-- ... -->
</form>
```

We can also remove all of the CSRF field code we had previously added to our users controller.

```
code controllers/users.go
```

Remove the CSRFField from the `New` function's `data`, and remove the line below that assigned it a value.

```
// Delete the CSRFField lines
func (u Users) New(w http.ResponseWriter, r *http.Request) {
        var data struct {
                Email     string
        }
        data.Email = r.FormValue("email")
        u.Templates.New.Execute(w, data)
}
```

Before moving on, restart the web server and verify that the signup page is rendering correctly. We won't be able to submit it any longer, as we haven't completely finished the CSRF template function, but the pages should render.

Our next step is to connect our custom `csrfField` function to the HTML provided by the `gorilla/csrf` package. We will complete this over the next few lessons.

Source Code

- Source Code - see the final source code for this lesson.
- Diff - see the changes made in this lesson.

13.11 Adding the HTTP Request to Execute

Inside `views/template.go` we are declaring the `csrfField` function for our templates.

```
func ParseFS(fs fs.FS, patterns ...string) (Template, error) {
        tpl := template.New(patterns[0])
        tpl = tpl.Funcs(
                template.FuncMap{
                        "csrfField": func() template.HTML {
                                return `<input type="hidden" />`
                        },
                },
        )
        // ...
}
```

At the moment this isn't using the correct HTML. We want to use the HTML generated by the csrf.TemplateField function provided by the **gorilla/csrf** package. Unfortunately, if we try to update our code at this time we will run into an error - **TemplateField** requires an ***http.Request** as an argument. We do not have access to an ***http.Request** when parsing our template because we parse the template when our server starts up, long before we handle any incoming web requests.

There are a few ways to address this issue:

1. We could delay parsing our templates until we have access to a request.
2. We could use a placeholder **csrfField** function when initially parsing our template, then replace the **csrfField** function with the correct one when we have access to the ***http.Request**.

The first option is the most direct. We could add the ***http.Request** as an argument to **ParseFS**, then wait to call that function until a web request has been made. While this is straightforward and works, it also has a downside - we won't know if a template is parsing correctly unless we make a web request to a page that uses that particular template. Tests can help, but if any page isn't tested it can still lead to an issue.

The second option takes a bit more effort to get right, but when setup correctly it ensures that all of our templates are parsed before our server starts

up. To implement this option, we will use a placeholder function in **ParseFS**, then later when we are ready to execute our template and have access to an ***http.Request** we will update the **csrfField** function just before executing the template. This also means that rather than adding an ***http.Request** as an argument to the **ParseFS** function, we will instead need it as an argument to the **Execute** method on our own **views.Template** type.

In this lesson we will focus on adding the ***http.Request** argument to our **template.Execute** method. In the next lesson we will utilize it to update the **csrfField** function prior to executing the template.

Open the source file with our **Template** type.

```
code views/template.go
```

Add the ***http.Request** argument to the **Execute** function.

```
func (t Template) Execute(w http.ResponseWriter, r *http.Request, data interface{}) {
        // ... unchanged
}
```

While our **views/template.go** source file doesn't have any bugs, our application code will no longer compile. We will instead see many errors like below.

```
./main.go:18:39: cannot use views.Must(views.ParseFS(templates.FS,
"home.gohtml", "tailwind.gohtml")) (value of type views.Template)
as type controllers.Template in argument to controllers.StaticHandler:
        views.Template does not implement controllers.Template (wrong type for
        Execute method)
                have Execute(w http.ResponseWriter, r *http.Request, data interface{})
                want Execute(w http.ResponseWriter, data interface{})
```

We changed the **Execute** method on our **views.Template** type, and it no longer matches the **controllers.Template** interface that we try to assign it

to in a few places. To fix this, we need to update the **controllers.Template** interface, then update our code that calls the **Execute** method.

```
code controllers/template.go
```

Update the interface to accept an ***http.Request**.

```
type Template interface {
        Execute(w http.ResponseWriter, r *http.Request, data interface{})
}
```

Now if we attempt to compile and run our code we get a new error.

```
controllers/static.go:13:29: not enough arguments in call to
static.Template.Execute
        have (http.ResponseWriter, nil)
        want (http.ResponseWriter, *http.Request, interface{})
```

The new error is telling us that we call the **Execute** method in our code, but we don't pass in the correct number of arguments. That makes sense since we never added the ***http.Request** argument to our function calls. Let's do that now.

```
code controllers/static.go
```

Update the **StaticHandler** and the **FAQ** function.

```
func StaticHandler(tpl Template) http.HandlerFunc {
        return func(w http.ResponseWriter, r *http.Request) {
                tpl.Execute(w, r, nil)
        }
}
```

```
func FAQ(tpl Template) http.HandlerFunc {
        // ... unchanged

        return func(w http.ResponseWriter, r *http.Request) {
                tpl.Execute(w, r, questions)
        }
}
```

Aside 13.12. Remove Unused Code

If your **static.go** source file still has the **Static** type with the **ServeHTTP** method, it can be deleted. We are no longer using the type.

Next we need to update our users controllers.

```
code controllers/users.go
```

Both the **New** and **SignIn** methods need updated.

```
func (u Users) New(w http.ResponseWriter, r *http.Request) {
        var data struct {
                Email string
        }
        data.Email = r.FormValue("email")
        u.Templates.New.Execute(w, r, data)
}
```

```
func (u Users) SignIn(w http.ResponseWriter, r *http.Request) {
        var data struct {
                Email string
        }
        data.Email = r.FormValue("email")
        u.Templates.SignIn.Execute(w, r, data)
}
```

With all those changes made, our code should now compile and run. We now have access to the `*http.Request` inside of our `Execute` method, and we will look into using it to set a correct `csrfField` template function.

Aside 13.13. Experience and the http.Request

In my experience, functions like our `template.Execute` almost always end up needing access to the `*http.Request`. Without prior experience that wouldn't be obvious, but with enough experience it becomes pretty clear. As a result, I would typically add the `*http.Request` argument when first defining the `Template` interface.

In the course I opted to leave the `*http.Request` argument out of our `Execute` method until we actually needed it. I did this because it felt more natural and easier to understand while learning, rather than preemptively adding it and leaving you to wonder, "How did he know we would need that?" Feel free to add it right away when building your own applications.

Source Code

- Source Code - see the final source code for this lesson.
- Diff - see the changes made in this lesson.

13.12 Request Specific CSRF Template Function

We have a placeholder `csrfField` function, and we have access to the `*http.Request` inside of our `Template.Execute` method. Now it is time to update the

Execute method to replace the **csrfField** function with the real one using **gorilla/csrf**.

Open the source file.

```
code views/template.go
```

Update **Execute** with the following code. We will discuss it shortly.

```
func (t Template) Execute(w http.ResponseWriter, r *http.Request, data interface{}) {
        tpl := t.htmlTpl
        tpl = tpl.Funcs(
                template.FuncMap{
                        "csrfField": func() template.HTML {
                                return csrf.TemplateField(r)
                        },
                },
        )
        w.Header().Set("Content-Type", "text/html; charset=utf-8")
        err := tpl.Execute(w, data)
        if err != nil {
                log.Printf("executing template: %v", err)
                http.Error(w, "There was an error executing the template.", http.StatusInternal
                return
        }
}
```

The first line we added is assigning the **t.htmlTpl** field to the **tpl** variable. We are doing this to make it a bit easier to work with the value in the rest of the code.

```
tpl := t.htmlTpl
```

Once we have the **tpl** variable, we call the **Funcs** method and pass in a new **template.FuncMap** with our real **csrfField** implementation. This uses the **gorilla/csrf** package to generate the HTML we need.

```
tpl = tpl.Funcs(
        template.FuncMap{
                "csrfField": func() template.HTML {
                        return csrf.TemplateField(r)
                },
        },
)
```

Finally, we update the **`Execute`** method call to be **`tpl.Execute`** rather than **`t.htmlTpl.Execute`**. For now this isn't 100% necessary, as both **`tpl`** and **`t.htmlTpl`** are pointers pointing to the same object, but it will break later in this lesson when we clone our template, so be sure to update the line.

```
w.Header().Set("Content-Type", "text/html; charset=utf-8")
err := tpl.Execute(w, data) // Update this line
// ...
```

Restart the server and reload the sign up page. We will now have a working **`<input>`** field with the correct CSRF token genrated by **`gorilla/csrf`**.

```
<input type="hidden" name="gorilla.csrf.Token" value="XlUQiV7MxJIBrLuFLdb0rivSDD5bUSoCUNk/CPUEg
```

Next, update the sign in form.

```
code templates/signin.gohtml
```

Find the HTML that starts the **`<form>`** tag and insert the same div we used on the sign up page.

```
<form ...>
        <div class="hidden">
                {{csrfField}}
        </div>
        <!-- ... Unchanged code -->
</form>
```

Once again, restart the server and reload the sign in page. Verify that it has the hidden `<input>` with the CSRF token. It is also worth verifying that both the sign in and sign up forms can be submitted without a CSRF error.

Our custom CSRF template function is working, but we have a subtle bug in our code. If we were to receive many web requests at roughly the same time, it is possible for our `csrfField` template function to generate the wrong CSRF token for one or more of those requests. This can occur because the `net/http` package turns every web request into its own goroutine, and in the `Template.Execute` method we are sharing the same HTML template across all of those requests. As a result, it is possible to set the `csrfField` for one user's web request, only to have it overwritten with another users `csrfField` function BEFORE we execute the first user's template and generate their HTML. As a result, both users could end up with the same CSRF token in their HTML, and it would only be valid for one of the users.

To address the issue, we are going to clone the template before adding any request-specific functions to it. This will ensure that this type of bug doesn't happen again, and it will help avoid any data leaks in our application. Cloning can be achieved using the `Clone` method provided by the `html/template` package.

```
code views/template.go
```

Use the `Clone` method to clone our template before making any changes to it. You will also need to update the line that calls `tpl.Execute`, as `err` is no longer a new variable so we cannot use `:=` on that line.

```
func (t Template) Execute(w http.ResponseWriter, r *http.Request, data interface{}) {
        tpl, err := t.htmlTpl.Clone()
        if err != nil {
                log.Printf("cloning template: %v", err)
                http.Error(w, "There was an error rendering the page.", http.StatusInternalServ
                return
        }
        tpl = tpl.Funcs(
```

```
                template.FuncMap{
                        "csrfField": func() template.HTML {
                                return csrf.TemplateField(r)
                        },
                },
        )
        w.Header().Set("Content-Type", "text/html; charset=utf-8")
        err = tpl.Execute(w, data)
        if err != nil {
                log.Printf("executing template: %v", err)
                http.Error(w, "There was an error executing the template.", http.StatusInternal
                return
        }
}
```

Restart the server and verify both the sign in and sign up pages are working.

Aside 13.14. Couldn't We Use a Mutex?

An alternative option to cloning our template is to use a `sync.Mutex`. This is a type provided by the `sync` package that allows us to lock and unlock access to data across goroutines. The idea is that we could lock our template so that only one web request could use it at a time, ensuring that we don't have race conditions.

Both approaches are valid, but I opted to use the cloning approach as it enables us to execute multiple templates at once on the off-chance template execution is slow due to a poor network connection or something similar.

To recap the process we are using:

1. In `ParseFS` we provide a placeholder function that doesn't really work, but is necessary to parse our template.
2. In `Execute` when we have access to the `*http.Request` we clone the template and add real function to the template.

3. Finally, we execute the cloned template that has request-specific functions.

While this technique can be a bit challenging to grasp at first, it is very powerful for adding custom functions to templates that use request-specific data. For instance, in the future this will enable us to easily determine if a user is logged in and dynamically render the correct buttons on the navigation bar without needing to pass data into our navbar templates.

Source Code

- Source Code - see the final source code for this lesson.
- Diff - see the changes made in this lesson.

13.13 Template Function Errors

We haven't seen it yet, but template function can return an error as their last argument. When an error is returned during template execution, the execution stops and the error is returned. We can see this with the following demo program.

```
tpl, err := template.New("demo").Funcs(
        template.FuncMap{
                "customMethod": func() (string, error) {
                        return "", errors.New("custom method error")
                },
        },
).Parse("{{customMethod}}")
if err != nil {
        fmt.Println("Parse error:", err)
}
err = tpl.Execute(os.Stdout, nil)
if err != nil {
        fmt.Println("Execute error:", err)
}
```

Run this on the Go Playground: https://go.dev/play/p/JfpAYY0wrb7

We can use this to make it clearer that a template function is a placeholder and need to be implemented at a later date. Let's do this for the **csrfField** function we declared.

```
code views/template.go
```

Update the **csrfField** function in **ParseFS** to always return an error. This will signal that the function needs to be implemented elsewhere.

```
func ParseFS(fs fs.FS, patterns ...string) (Template, error) {
        tpl := template.New(patterns[0])
        tpl = tpl.Funcs(
                template.FuncMap{
                        "csrfField": func() (template.HTML, error) {
                                return "", fmt.Errorf("csrfField not implemented")
                        },
                },
        )
        tpl, err := tpl.ParseFS(fs, patterns...)
        if err != nil {
                return Template{}, fmt.Errorf("parsing template: %w", err)
        }
        return Template{
                htmlTpl: tpl,
        }, nil
}
```

If we wanted to see what this error looks like, we could comment out the **csrfField** function portion of the **Execute** method.

```
func (t Template) Execute(w http.ResponseWriter, r *http.Request, data interface{}) {
        tpl := t.htmlTpl
        tpl = tpl.Funcs(
                template.FuncMap{
                        // "csrfField": func() template.HTML {
                        //         return csrf.TemplateField(r)
                        // },
                },
```

```
        )
        // ...
}
```

Now if we attempt to load the sign up page we will see a half-rendered page, and our terminal will have the following error in it:

```
executing template: template: signup.gohtml:9:10: executing "signup.gohtml" at <csrfField>: err
```

We will also see a message similar to the following:

```
http: superfluous response.WriteHeader call from github.com/joncalhoun/lenslocked/views.Templat
```

The first error is occurring because our template calls the **`csrfField`** function and it returns an error. It is what we expected when we don't implement the **`csrfField`** template function.

The second error is a bit more confusing. When we called **`tpl.Execute`** and passed in the **`http.ResponseWriter`**, it began executing the template and writing the results to the response writer. When the **`http.ResponseWriter`** has anything written to it, it first checks to see if a status code has been set. If one hasn't been set, the **`ResponseWriter.WriteHeader`** method is called and the default HTTP status code of 200 is used.

When our **`Execute`** method eventually encounters an error, we then call **`http.Error`** and pass in a the response writer, a message, and a new status code. **`http.Error`** then attempts to write the status code using **`WriteHeader`**, but the 200 status code has already been written, and it cannot be updated because in many cases it has already been sent to the user making the web request. This is what leads to the superfluous error message.

One way to address this is to buffer the results from the template execution. Head to the **`Execute`** method in our **`views/template.go`** source file and update it with the following code.

```
func (t Template) Execute(w http.ResponseWriter, r *http.Request, data interface{}) {
        // ... unchanged
        w.Header().Set("Content-Type", "text/html; charset=utf-8")
        var buf bytes.Buffer
        err := tpl.Execute(&buf, data)
        if err != nil {
                log.Printf("executing template: %v", err)
                http.Error(w, "There was an error executing the template.", http.StatusInternal
                return
        }
        io.Copy(w, &buf)
}
```

We are using the bytes.Buffer type to store the final results from the template execution, then once we know the entire template has been executed without any errors we then call **`io.Copy`** to copy the data from our buffer into the **`http.ResponseWriter`**.

If we were to restart our server and load the sign in page we would no longer see half the page rendered and would instead see an error message. The correct HTTP status code would also be set.

While this approach removes our superfluous warning message, it does require us to store the entire results of the template execution in memory before finally writing to the response writer. If we had instead opted to write directly to the response writer, in many cases our server could stream the response to the user making the web request, alleviating our need to store the entire results in memory all at once. All of our HTML pages are fairly small, so this won't have a major performance impact, but if we were writing very large HTML pages and handling many requests at once it might make sense to execute templates directly to the **`http.ResponseWriter`** and to allow invalid status codes to sneak through in order to optimize the experience for non-error use cases.

Before moving on, make sure the final **`csrfField`** template function is uncommented so that our sign in and sign up pages render correctly.

Source Code

- Source Code - see the final source code for this lesson.
- Diff - see the changes made in this lesson.

13.14 Securing Cookies from Tampering

The last thing we need to talk about is cookie tampering. Cookie tampering is the process of altering the data stored in a cookie, often with malicious intent. As we saw earlier, extensions like EditThisCookie make it very easy to alter the data stored in cookies. Attackers can also write programs that make web requests and send any arbitrary cookie the attacker wants along with the request. As a result, we cannot blindly trust the data stored in a cookie. We need a way to validate our cookies.

There are two common strategies for doing this:

1. Digitally signing data
2. Obfuscation

In this lesson we will look at how each technique works at a high level.

13.14.1 Digitally Signing Data

Imagine we wanted to write the following data to a cookie:

```
{
  "id": 123,
  "email": "jon@calhoun.io"
}
```

If we just wrote this data to the cookie, a user could edit it and pretend to be another user. One way to prevent this is to digitally sign the data. To sign the data, we can use a hashing function that requires a secret key that only our server knows. This way nobody else can fake the signature, because they don't know the secret key.

HMAC is an example of a hashing function for signing data. What makes the HMAC hashing function ideal is that it requires both a secret key and data to be hashed, and the output of the hashing function changes if either of these changes. A few examples of this are shown below. Note that changing either the secret key or the data being hashed will result in a different hash value.

```
HMAC("secret-key-1", `{ "id": 123 }`)
// 60fc0ff9d3f4132cbe7b9dd12ef51837d8e...

// Different key, different result
HMAC("secret-key-2", `{ "id": 123 }`)
// b91c4b45a02dc9f92725d93ec4a7bfd6a4e...

// Different value, different result
HMAC("secret-key-1", `{ "id": 333 }`)
// 78257ef3e303a46b934fccf7ed9c01d7f77...
```

In the first and second examples the data is the same, but the secret key changes so the resulting hash differs. The final example uses the same secret key as the first, but since ID changes it also gives us a different result.

Now imagine that our secret key is something silly like **`secret-key`**, and we are encoding the JSON we saw earlier. Our server might create a signature with code similar to the code below.

```
package main

import (
        "crypto/hmac"
        "crypto/sha256"
        "encoding/hex"
        "fmt"
)
```

```
func main() {
        secret := "secret-key"
        cookie := `{"id": 123, "email": "jon@calhoun.io"}`

        h := hmac.New(sha256.New, []byte(secret))
        h.Write([]byte(cookie))
        result := h.Sum(nil)
        fmt.Println(hex.EncodeToString(result))
}
```

You can run this code on the Go playground: https://go.dev/play/p/942QmEci-ZT

The output from this code is called a digital signature, and is shown below.

```
b0e789e017e1e62d44e3b0be9e5e7683afbfd7f2fd71f2508b12fe872d0fdc0d
```

Once our server has a signature, it can be added to the JSON that we plan to store in the cookie.

```
{
  "id": 123,
  "email": "jon@calhoun.io",
  "signature": "b0e789e017e1e62d44e3b0be9e5e7683afbfd7f2fd71f2508b12fe872d0fdc0d"
}
```

We would then write this JSON data to the cookie, and it will be included with all future web requests letting us know who the user is.

Later when a the user makes a web request we need to validate the data in the cookie. With digital signing, we do this by reading the signature, then hashing the rest fo the JSON to verify the hash we generate matches the signature. In our example above, we would read the signature:

```
b0e789e017e1e62d44e3b0be9e5e7683afbfd7f2fd71f2508b12fe872d0fdc0d
```

Then we would generate a signature for the remaining JSON data:

```
{
  "id": 123,
  "email": "jon@calhoun.io"
}

// Generated Signature:
// b0e789e017e1e62d44e3b0be9e5e7683afbfd7f2fd71f2508b12fe872d0fdc0d
```

If the two match, we know that the data is valid and we can trust it. On the other hand, if an attacker had altered any of the data the new signature would be different, and we would know to disregard the cookie as invalid.

By keeping the secret key on our server, any would-be attackers might be able to update the data that we stored in their cookies, but because they don't know our secret key, they won't be able to produce a valid hash to match the data.

Aside 13.15. JSON Web Tokens

If you have ever heard of JSON Web Tokens (JWT), it is a standard that utilizes digital signatures to create authentication tokens. The data stored in a JWT is visibly by anyone, but editing the data without the secret key can be detected as the signature cannot be properly updated.

JWTs are not the only way to digitally sign data, nor does it use any new techniques. It is a way of standardizing things so that libraries from different programming languages could all know how to process the same digitally signed data.

13.14.2 Obfuscation

Rather than signing data, another way to prevent cookie tampering is to obfuscate the data. Obfuscation is the process of making the data in a cookie unclear to attackers, making it impossible for them to determine how to generate valid data.

For instance, rather than storing a user's email address in the cookie, we could instead generate a random string for each user:

```
user           | random_string
-------------- | --------------
jon@calhoun.io | 7r1mpIjJcl
bob@bob.com    | UkpEUzFPJy
```

We could then store the random string inside of our cookie. Now when a user makes a web request, we can look up who the user is via our table, but an attacker wouldn't be able to generate a valid random string because they are just that - random strings. An attacker would need to guess random values hoping that one of them works.

This approach is often referred to as using **sessions**, and the random string is the `session token` or `session ID`. We will be using this approach to build out our authentication system, as it is less error-prone than digital signatures, and it is much easier to invalidate a user's session compared to using digitally signed cookies.

The next section of this course will be dedicated to learning more about sessions and how to implement them in Go.

Aside 13.16. Why aren't we using JWTs?

A common question at this point in the course is, "Why aren't we using JWTs?" This typically gets asked because the student has read about JWTs on some web-

site as the new way to handle authentication, and they don't want to learn outdated technology.

First, let me assure you that sessions are not an outdated technology. Not only are they used in many secure applications, but sessions are also needed as part of JWTs as a means of creating refresh tokens. In short, even if you opt to use JWTs in the future, you will still need to know how sessions work.

Without going deep down a rabbit hole, the short answer for why we don't use JWTs is that JWTs are more complex to implement and don't offer any major benefits to our application. There are some cases where JWTs can be beneficial, but those don't apply to our application. In addition to not providing any benefits, using JWTs could introduce additional security risks if not set up properly, so they aren't a great choice unless there is a strong need for them.

13.15 Cookie Exercises

13.15.1 Ex1 - Set and View Cookies in your Browser

Attempt to create various cookies with your Go code, then view the cookie values with both your browser and Go code.

As an additional step, try setting cookies with a specific domain or path and verifying that it works.

13.15.2 Ex2 - Explore the Cookie Docs

Head to the net/http docs and explore the various options for cookies. It might also be worth visiting the Mozilla docs to explore what some of these options

can do.

13.15.3 Ex3 - Redirect When a Cookie is Not Found

Our `CurrentUser` method in the users controller currently renders the current user via their cookie. Update the controller to redirect the user to the sign in page if the cookie is not found.

Note: We will do this later in the course if you have trouble with the exercise.

13.15.4 Ex4 - Attempt to Create Middleware

Try creating middleware of your own. For instance, you could create middleware that logs the IP address of every request, the path of every request, or both. We could even log both the path and the total time taken to render the page, which might be useful for tracking the performance of various pages.

Chapter 14

Sessions

14.1 Random Strings with crypto/rand

We previously learned that obfuscation can be used to prevent cookie tampering. To quickly recap, our server can generate a unique string for each user - called a session token or a session - and store it in a table along with a way to look up the user. In our case we will be using the user's ID. Below is an example of what this table might look like, but the session tokens are not truly random to make it easier to read.

```
user_id | session token
==================================
      1 | AUniqueRandomString123abc
      2 | AnotherString789xyz
```

When a user makes a web request, it will include the session token in it. In our application we will store these in cookies, but they could be provided in other ways. Once our sever reads the session token, it will look up the session token in our table of session tokens. If the app finds a token, it will also know the user ID associated with it. If no token is found, the session token is considered

invalid and the user can be redirected to the login page or given an unauthorized message of some sort.

This technique prevents cookie tampering from becoming an issue by making it virtually impossible for attackers to guess or predict a valid session token. Attackers can still change the values stored in their cookies, but without knowing what a valid session token is, they will have to guess random values with the odds stacked heavily against them.

We are going to make our session tokens hard to guess and predict by using randomness. This will prevent any situation where an attacker can determine a pattern and predict future session tokens.

In theory using random data to generate a sessions sounds easy, but in practice true randomness is rare. For instance, if a group of people are asked to pick a number between 1 and 10, 7 will be picked far more often than any other number, while 1 and 10 will be picked far less often than any other number. If we were generating session tokens randomly and introduced a bias like this, attackers could guess the more likely session tokens and greatly increase their odds of bypassing our security measures. In order for our sessions to be secure, we need to be as close to true randomness as possible. The challenge here is that computers are deterministic machines that produce the same output given the same input. While that might not always seem like the case when troubleshooting an issue, what is really happening in those cases is the inputs are changing every so slightly. For instance, another process running on a computer might impact the timing causing different results.

The **`math/rand`** package is a good example of perceived randomness. If we seed it with the current time, the outputs will appear to be random, but in reality the package isn't generating random numbers. We will explore this in a future lesson to help demonstrate why it isn't a suitable package for our needs, but the short version is that the **`math/rand`** package is a number generator that will always produce the same outputs given the same inputs, making it very predictable.

For our use case we need the crypto/rand package. According to the docs:

> Package rand implements a cryptographically secure random number generator.

In simpler terms, this means that the random values from **`crypto/rand`** are about as close to truly random as we can get without special hardware. This is achieved by using build-target specific implementations of the package. When we build our Go code for Windows, various versions of Linux, or even WASM, the Go code used for the **`crypto/rand`** package will be different and will utilize features provided by the environment or operating system for generating secure random values.

We can see this in the docs for the Reader variable that is exported from the package:

> On Linux, FreeBSD, Dragonfly and Solaris, Reader uses getrandom(2) if available, /dev/urandom otherwise. On OpenBSD and macOS, Reader uses getentropy(2). On other Unix-like systems, Reader reads from /dev/urandom. On Windows systems, Reader uses the RtlGenRandom API. On Wasm, Reader uses the Web Crypto API.

We will be focusing on the Read function exported by the **`crypto/rand`** package.

```
Read(p []byte) (n int, err error)
```

The **`Read`** function we see here is very common in Go; it is the same as the method required for a type to implement the **`io.Reader`** interface.

```
type Reader interface {
        Read(p []byte) (n int, err error)
}
```

In fact, the **Read** function utilizes the Reader variable that is exported from the **crypto/rand** package, and the **Reader** variable implements the **io.Reader** interface.eader‘

To use the **Read** function, we pass in a byte slice with a specific size, and **Read** function will read random data and write it into the provided slice. For instance, if we provide a slice where **len(slice)** is equal to **12**, we will have 12 bytes of random data read and placed into our slice. If our slice has data in it, it will be overwritten.

As with all **io.Reader**s, the return value **n** represents the number of bytes written, and should always be equal to size of the byte slice provided unless there is an error. The second return value is **err**, the error, and it will be set if something went wrong while reading.

Let’s try writing some code that uses the **Read** function from the **crypto/rand** package. We won’t be keeping this code, so Go Playground links will be provided to run the code, but feel free to code it locally to help with understanding it.

```
package main

import (
        "crypto/rand"
        "fmt"
)

func main() {
        n := 8
        b := make([]byte, n)
        nRead, err := rand.Read(b)
        if err != nil {
                panic(err)
        }
        if nRead < n {
```

```
                panic("didn't read enough random bytes")
        }
        fmt.Println(b)
}
```

This code can be run on the Go Playground.

In this code we first set `n := 8`, which will be how many bytes of random data we want. We then make a byte slice with `n` bytes. After that we pass the byte slice into `rand.Read`, which leads to the `crypto/rand` package writing 8 bytes of random data to our slice.

After that we check for errors and verify that we read the correct amount of data, and finally we print out the random data we read.

In many case we could use the byte slice with random data as-is. We plan to store our random values in a cookie, so we need our random data to be a string. If we try to naively convert our byte slice into a string we may end up with gibberish output, because the random data may have invalid characters in it. We can see this by adding the `Println` statement below to our code.

```
n := 32
b := make([]byte, n)
nRead, err := rand.Read(b)
if nRead < n {
  panic("didn't read enough random bytes")
}
if err != nil {
  panic(err)
}
// The conversion below does not work as expected.
fmt.Println(string(b))
```

This code can be run on the Go Playground.

Running this code is very likely to produce output with invalid characters.

```
w}IU'Sk~z]**h AQ^L
```

While all strings in Go are really byte slices behind the scenes, not every combination of bytes will produce a valid character. When we attempt to print out an invalid string, we will see a weird gibberish looking character. This occurs because strings use a character encoding (like UTF-8), and these encodings have less characters than there are possible byte combinations, so some will inevitably be invalid. What this means for us is that we cannot directly convert our byte slice into a string via Go's conversion (eg `string(byteSlice)`), as this might lead to silent errors when attempting to write our cookies. Instead, we need to encode the bytes using an encoding algorithm.

Earlier in the course we saw hex encoding when we were looking at how HMAC works. Hex works by converting every byte into two characters from the set of characters below.

```
0123456789ABCDEF
```

This works because there are 16 characters in this character set, so using two of them gives us 16^2, or 256, valid combinations. The values we can store in a byte also range from 0 to 255, giving us 256 possible values, so we can map every possible byte value into a two character combination.

Other encoding algorithms work in a similar way - every byte, or perhaps every few bytes, is encoded into a character in a predefined set of characters giving us an encoded result. This is very different from hashing though, as we can always reverse the encoding and determine what the original bytes were.

Base64 is another popular encoding scheme. It is quite popular in web applications, and Go provides the `encoding/base64` package in the standard library. We are going to use this package to encode our random data and produce a string.

```
n := 32
b := make([]byte, n)
nRead, err := rand.Read(b)
if nRead < n {
        panic("didn't read enough random bytes")
}
if err != nil {
        panic(err)
}
fmt.Println(base64.URLEncoding.EncodeToString(b))
```

Once again this code doesn't need to be created locally and can be run on the Go Playground.

Unlike hex encoding, the **encoding/base64** package provides a few ways to encode strings. These are each separated inside the package as variables. We will be opting to use the **URLEncoding** variant, but we could have chosen any. This one tends to be my default because I know values produced by it can be included in URLs without any special encoding.

The primary benefit to using base64 over hex is that base64 produces smaller strings that still contain the same amount of bytes. While this is unlikely to be an issue with our application, it is nice to reduce the total size of our cookies wherever possible.

14.2 Exploring math/rand

We previously mentioned the **math/rand**, and it was noted that it is not suitable for session tokens because it is not truly random. In this lesson we are going to explore the **math/rand** package to show how it works and demonstrate that it isn't a good fit for our case.

This lesson can be skipped. All of the code is done in the Go Playground, and no changes are made to our source code.

Let's first explore code that uses the **math/rand** package to generate random numbers using the Intn function.

```
package main

import (
        "fmt"
        "math/rand"
)

func main() {
        fmt.Println(rand.Intn(100))
        fmt.Println(rand.Intn(100))
        fmt.Println(rand.Intn(100))
}
```

This code can be run on the Go Playground.

Each call to **Intn(100)** generates a new random number between 0 and 99. Upon first running the code, it appears to be random, but if we were to run the code several times it starts to become clear that it is not random. We get the same output every time we run the code.

```
81
87
47
```

This occurs because the **math/rand** package isn't actually random. Instead, the **math/rand** uses a psuedo-random number generator, and every time we run our code that number generator starts over from the same place.

The **math/rand** number generator works by first starting with a seed, then that seed is used to generate a sequence of seemingly random values. By default, the number generator is seeded with the value **1**, which is why we see the same output every time we run the code. If we change the seed, we will start to see new numbers being generated.

```
rand.Seed(123)
fmt.Println(rand.Intn(100))
fmt.Println(rand.Intn(100))
fmt.Println(rand.Intn(100))
```

Running this code will produce the following output:

```
35
49
3
```

If we run our code with the same seed, we will always get the same output, but changing the seed will give us a different output. With this in mind, one way to give the appearance of randomness with **`math/rand`** is to use the current time as a seed.

```
rand.Seed(time.Now().UnixNano())
```

This code can be run on the Go Playground.

Using the current date and time as the seed gives the perception of randomness because every time we run the code we will have a new date and time, so the seed will differ and we will see different results. This doesn't work on the Go Playground because **`time.Now()`** is always the exact same date and time, but if we run the code locally we will see the random numbers generated change from run to run.

Even when using the current time as a seed, **`math/rand`** isn't suitable for session tokens. One concern is that someone might determine our seed and be able to predict future session tokens. They might attempt to do this by signing in and out of our application many times until they have enough valid session tokens to guess possible seed values. Even if not extremely likely, if an attacker were to pull this off they could then determine both past and future session tokens that our application is using.

`math/rand` is a good fit for cases where something like randomness is useful, but being truly random is not critically important. For example, there are many reasons to run tests in a Go project in a random order, but we don't need them to be truly random. In fact, having a way to recreate the random order that was used via a see is incredibly useful for tests so that we can recreate a failure scenario. In cases like this, `math/rand` is a fantastic fit.

As we progress through the course, take an extra moment to ensure that the `crypto/rand` package is imported whenever we work with random values. Otherwise it could open up an application to security issues.

14.3 Wrapping the crypto/rand Package

While learning how to use the `crypto/rand` package, we saw that it requires us to make a byte slice, call `rand.Read`, check for errors, and finally encode the results. While this isn't a ton of code, it is enough logic that we likely don't want to repeat it everywhere that we want a random value. Instead, we are going to create a new `rand` package of our own that wraps the `crypto/rand` package and exposes functions more tailored to our needs. Then later in our code when we want to use random values, we will import our own `rand` package rather than the wrapped `crypto/rand` package.

The primary benefit to wrapping a package is to tailor it for our needs. In the case of `crypto/rand`, the package only offers the `Read` function, but what we would really like are functions like `String(n)` and `SessionToken` that handle all the details for us. We are going to make that happen.

First, create a directory for the package, then open a source file named `rand.go`.

```
mkdir rand
code rand/rand.go
```

In this source file we will add a `Bytes` function that is very similar to the code

we saw in an earlier lesson.

```
package rand

import (
        "crypto/rand"
        "fmt"
)

func Bytes(n int) ([]byte, error) {
        b := make([]byte, n)
        nRead, err := rand.Read(b)
        if err != nil {
                return nil, fmt.Errorf("bytes: %w", err)
        }
        if nRead < n {
                return nil, fmt.Errorf("bytes: didn't read enough random bytes")
        }
        return b, nil
}
```

Bytes uses code nearly identical to the code we used earlier when calling **crypto/rand**'s **Read** function with a few notable differences. The fist is that we don't create a byte slice, one will be returned to us, and the second is that we pass in a value **n** which is the number of bytes of random data we would like in our byte slice. The main advantage to this function is that it is easier to use, but a potential downside to consider is that it creates a new byte slice every time we call it, so in circumstances different from our own it may be less efficient.

Using the **Bytes** function, we can create a function that produces a base64 encoded string with a given number (**n**) of bytes.

```
// String returns a random string using crypto/rand.
// n is the number of bytes being used to generate the random string.
func String(n int) (string, error) {
        b, err := Bytes(n)
        if err != nil {
                return "", fmt.Errorf("string: %w", err)
        }
        return base64.URLEncoding.EncodeToString(b), nil
}
```

Finally, we can add a helper function that returns a fixed-size session token for us. This is very specific to our application, but it can be handy to avoid any chance that someone will write code that generates session tokens with too few bytes.

```
const SessionTokenBytes = 32

func SessionToken() (string, error) {
        return String(SessionTokenBytes)
}
```

We set the number of bytes in a constant with the value of `32`. We will discuss why we use 32 bytes in the next lesson.

Source Code

- Source Code - see the final source code for this lesson.
- Diff - see the changes made in this lesson.

14.4 Why Do We Use 32 Bytes for Session Tokens?

To understand the reasoning behind why we use 32 bytes, we first need to learn a bit about data types. A byte is a data type that stores 8 bits of data. A bit of data is either a `0` or a `1`. This means a byte can have 256 possible values. Putting that all together, we have two values per bit, and eight bits of data, so we have `2^8`, or `256`, possible values that can be stored in a byte.

Now that we know how many possible values a byte has, lets explore how that affects an attackers ability to guess a value. Imagine that we generated session tokens with only one byte of random data. This would mean that we only have `256` possible session tokens that we can create, and the math here is `256^1 = 256`.

If we used two bytes, we could create 65536 tokens. The math here is `256^2 = 65536`. Adding only one more byte vastly increases the total number of session tokens we can create.

If we jump to 32 bytes we have `256^32`, or `1e77` possible tokens. If we were to write that out, that is a 1 followed by 77 zeros. That is a massive number! Do we really need that many possible session tokens?

To understand this better, let's first imagine a simple game and discuss the odds in the game. In our imaginary game, one person picks a number between 1 and 10, and the other person tries to guess the number. When playing this game, the person guessing has a 1/10 (10%) chance of guessing the number correctly the first time assuming the person picking actually picks a number at random. Given 3 guesses, the player has a 3/10 (30%) chance of guessing the number.

If we were generating session tokens for users, guessing a session token for a single user would work in a similar manner. If we have 256 possible session tokens, the odds of guessing a valid session token are 1/256. If we can guess multiple times, as computers are capable of doing, we would have `n/256` odds, where `n` is the number of guesses we take.

Our app will have many users though, so that changes the odds a bit. Going back to our game, imagine we have 3 people who are each picking a unique number between 1 and 10, and then someone gets to guess a number. If anyone picked that number, the guesser wins. Otherwise they guess again. What are the odds of winning that game in 1, 2, or 3 guesses?

With 1 guess, the odds of success are 30%. We know that 3 out of 10 numbers will be picked, so on the first guess the guesser has a 3 in 10 chance of guessing a picked number.

With 2 guesses the odds of success jump to 53%. That seems wild, but for the guesser to lose they would need to guess incorrectly twice. On the first guess they have a 7/10 chance of getting it incorrect, and on the second guess they have eliminated one invalid option while three are still going to lead to success, giving them a 6/9 chance of getting it incorrect. The guesser needs to get both

guesses incorrect, so the odds of failure are `7/10 * 6/9`, or `46.66%`. In any other scenario they guess correctly at least once, so in `53.33%` of cases they guess someones number within two guesses.

With 3 guesses the odds of success are 70%. Yikes! Those odds rose drastically for the guesser compared to the 30% they had when only one person picked a number.

Aside 14.1. Explaining the odds

With 2 guesses there are three possible outcomes:

1. The guesser guesses right on the first try. There is a 30% chance of this happening, as 3 of 10 numbers are picked.
2. The guesses guesses wrong on the first try (7/10), then correct on the second try (3/9). Multiplying these together, there is a 23.3% chance of this happening.
3. The guesser guesses wrong on first try (7/10), then wrong again on the second try (6/9). There is a 46.6% chance of this happening.

While the third case is by far the most likely case individually, we need to remember that either of the first two cases result in a win for the guesser, so we have to add these odds together giving us a `53.3%` chance of winning within two guesses.

The odds for three guesses work in a very similar manner, except they add up to a 70% chance of the guesser winning within three guesses.

Assuming we decided that we wanted the odds after three guesses to be around 30%, how could we adjust the four player game to better match these odds?

One option is to increase the range of numbers that people can pick from. Rather than asking people to pick a number between 1 and 10, we could instead ask them to pick a number between 1 and 28. Now the odds of the guesser winning in 3 picks is at roughly 30% again, but we almost needed to triple the range of the numbers to pick from.

Our app's session tokens are similar. The more users we have, the easier it is to guess a valid session token. Computers can also guess random session tokens very quickly. As we saw earlier, one of the ways to decrease the odds of someone guessing a value in a set of possible values is to increase the size of that set. In other words, if our session tokens are composed of 32 bytes and there are many possible values, guessing a session token that is in use becomes incredibly challenging, if not impossible. For this reason, we opt to use 32 bytes for our session tokens.

According to the OWASP Foundation, an organziation focused on security practices, a session should be at least 128 bits, or 16 bytes. We opt to use 32 as this is a bit better than the minimum, while not being so large that it hinders the performance of our application.

14.5 Defining the Sessions Table

14.6 Defining the Sessions Table

Now that we understand how to generate random strings and why we are using 32 bytes for our session tokens, we are ready to start building a `SessionService`. This will be similar to our `UserService`, and it will be responsible for managing everything related to creating, deleting, and otherwise interacting with sessions.

Before we can start writing the Go code for this, we first need to decide what the data should look like in our database. We will do this in SQL first, much

like we did for the users table.

```
code models/sql/sessions.sql
```

There are several options for how we structure sessions in our database. We could add a field to the users table that stores each users session, or we could create an entirely new table that has a relationship with users. Let's use the latter option.

```
CREATE TABLE sessions (
  id SERIAL PRIMARY KEY,
  user_id INT UNIQUE,
  token_hash TEXT UNIQUE NOT NULL
);
```

Save the **sessions.sql** file so we have a copy of the SQL we used, then let's discuss what the SQL is doing.

The first field we declare is the **id**. Every session will have an **id** that serves as its own unique identifier, and it is one way we will select a specific session in our database. We will see this field in most database tables.

user_id will be used to create a relationship between a session and a specific user. For instance, if I sign up and my user account gets the id **123**, then my session will have the **user_id** value of **123** so that we can tell which user the session is for.

There are a number of terms used to define this relationship, such as a **one-to-one** or **belongs-to**. For instance, it can be said that a user has one session in our database, or we could say that a session belongs to a user. Due to the **UNIQUE** constraint that we added, a user cannot currently have more than one session at a time. This constraint could be removed, allowing a user to by signed in from multiple devices at one time, but for now we will limit a user to a single session to simplify session management.

We will talk about SQL relationships and more advanced SQL topics later in the course. For now all we really need to know is that every session is associated with a user, and the `user_id` field is how we keep track of that association.

The last field we need is the `token_hash`. This is where we are going to store the session token value, or rather a hash of it. Similar to how we don't store raw passwords in our database in case of a security leak, we also won't be storing raw session tokens for similar reasons. Instead, we will hash the token prior to storing it in our database, then we will hash session tokens to validate them with what we store in our database. This ensures that even if our database is leaked, hackers cannot use existing sessions to impersonate our users because they still will not have access to the original session token. They will only have access to hashes of session tokens. This will be discussed more when we implement both the hashing and looking up sessions.

The `token_hash` needs to be unique so that we ensure no two users have the same session token, and it cannot be null as this wouldn't be a valid session token.

Now that we understand the SQL table we defined, let's add it to our database. Open either Adminer (via http://localhost:3333/) or `psql` (via `docker exec`) and execute the SQL we just wrote.

```
# CREATE TABLE sessions ( ... );
CREATE TABLE
```

We can also quickly validate that the table exists by executing a query on the table.

```
SELECT * FROM sessions;

 id | user_id | token_hash
----+---------+------------
(0 rows)
```

Aside 14.2. Migration Tools

In a future lesson we will look into migration tools that will allow us to add new SQL tables and manage how our database is structured, but for now we are manually managing our SQL database. As a result, tables need to be added manually via `psql` or Adminer.

With our database table created, we need to create a type in our Go code that matches the table.

```
code models/session.go
```

We will start with a one-to-one mapping, and later we can adjust the fields if it makes sense.

```
package models

type Session struct {
        ID        int
        UserID    int
        TokenHash string
}
```

Save the `session.go` source file and we are ready to start working on the session service.

Source Code

- Source Code - see the final source code for this lesson.
- Diff - see the changes made in this lesson.

14.7 Stubbing the SessionService

Rather than implementing the entire `SessionService` at once, we are going to focus on stubbing out two functions to start. The first will be a function used to create new sessions, and the second will be a function to look up a user via their session. The goal is to focus on how each function will be used by the rest of our code before spending time implementing, then once we feel we have a good design in place we can proceed with the implementation details.

We will start with the `session.go` source file.

```
code models/session.go
```

In this source file we are going to create a `SessionService` type that has a `*sql.DB` field, since we know we will need access to the database to store sessions. We don't know of any other fields we need at this moment, but we can add additional ones when we need them.

```
package models

import (
        "database/sql"
)

type Session struct {
        ID        int
        UserID    int
        TokenHash string
}

type SessionService struct {
        DB *sql.DB
}
```

The first method we are going to create for the `SessionService` type is `Create`. At this point we know that each session will have a user ID associated with it, so we will need that as an argument when creating a session. It

is also common to return the object being created, or an error, so we will want to consider returning a `*Session` and an `error` at a minimum. That gives us the following starting point.

```
// We aren't 100% sure what the arguments should be to this function yet.
func (ss *SessionService) Create(userID int, ???) (*Session, error) {
        return nil, nil
}
```

Every session also needs a session token. We have a few options for making this happen when we create a session:

1. Accept a hashed session token as an argument, and we can write this to our sessions table in the database.
2. Accept the original session token as an argument, and then have the `Create` method handle hashing it.
3. We could both create the session token and hash it inside `Create`, then we could return the original session token from the `Create` method.

In all three cases we will end up creating a session token, hashing it, and storing the hashed session token in the database. The main difference between all three approaches is where the logic is in our application, and who is responsible for it.

With the first approach, our `SessionService` isn't responsible for any of the session token logic, and it also isn't responsible for any hashing logic. We leave that all up to whatever code calls the `Create` method. This might make sense if our `SessionService` only handled writing and reading from the database, but in our case it likely makes more sense to handle additional business logic related to session creation.

The second option is a nice middle ground, where the hashing is handled by our `SessionService`, but the token creation is not. The biggest concern with this

approach is that it becomes hard to ensure that session tokens are being created securely in all cases. A better option would be to let the `SessionService` handle everything, ensuring that no session is added to our database without a secure session token associated with it. That means we will be using the third option, so our `Create` method will look like the code below.

```
func (ss *SessionService) Create(userID int) (*Session, error) {
        // TODO: Create the session token
        // TODO: Implement SessionService.Create
        return nil, nil
}
```

While this looks good at first, we still have an issue. Our `Create` method creates the raw session token, but `*Session` being returned only has the hashed session token. We need a way to return the original session token to whatever code calls the `Create` method. Once again we have options.

The first option is to add an additional return value. An example of this is below.

```
func (ss *SessionService) Create(userID int) (*Session, string, error) {
        return nil, nil
}
```

Another option is to add a `Token` field to the `Session` type, but this might mislead developers into thinking this value is always available when it isn't. When we look up a session in our database, we won't have access to the original session token at that time. If we opt to go with this approach we will want to document the behavior to avoid any confusion.

Both approaches are valid choices, and which gets used depends on a variety of factors including personal preference, keeping code consistent, and the number of values that need returned (we likely don't want 20 return values). We could even go with a hybrid approach of creating a new type like `NewSession` that we return, allowing us to avoid adding a `Token` field to the `Session` type.

In this course we will be going with the second option - adding the `Token` field to the `Session` type and documenting that it is only set at certain times. This will help us minimize the number of return values, and adding a new type like the `NewSession` type felt like overkill at the moment.

Open up the `session.go` source file in the models directory and update it with the following code.

```
package models

import (
        "database/sql"
)

type Session struct {
        ID     int
        UserID int
        // Token is only set when creating a new session. When looking up a session
        // this will be left empty, as we only store the hash of a session token
        // in our database and we cannot reverse it into a raw token.
        Token     string
        TokenHash string
}

type SessionService struct {
        DB *sql.DB
}

// Create will create a new session for the user provided. The session token
// will be returned as the Token field on the Session type, but only the hashed
// session token is stored in the database.
func (ss *SessionService) Create(userID int) (*Session, error) {
        // TODO: Create the session token
        // TODO: Implement SessionService.Create
        return nil, nil
}
```

Once a session is created we will need a way to query our `SessionService` to determine who the user is with that session. We will have access to the original session token via a user's cookie, so we can use that as an argument, but we need to determine what to return and where to put this new code. There are a few options to consider:

1. Return a `*Session` when we query via session token, then let the caller query for the user separately.

2. Query for the user directly from the `SessionService`.

3. Query for the user directly from the `UserService`.

Examples of all three are shown below.

```
// 1. Query a Session via raw token, then query the user separately via
// a function added to the UserService.
func (ss *SessionService) ViaToken(token string) (*Session, error)
func (us UserService) ViaID(id int) (*User, error)

// 2. Query a user via raw token using the SessionService
func (ss *SessionService) User(token string) (*User, error)

// 3. Query a user via raw token using the UserService
func (us *UserService) ViaSessionToken(token string) (*User, error)
```

All three of these approaches have a trade-off of some kind. With the first approach, we are guaranteed to make multiple database queries. In most cases this won't matter, but SQL databases are very powerful and if our application was under load we could optimize queries like this to be done in a single query using a `join` (we will learn about these later).

The second and third approach allow us to optimize the database query, but at the cost of intermingling responsibilities. In one case our `SessionService` needs to know about the users table and how construct a `User` object. In the other the `UserService` needs to know about the hashing logic used for session tokens to query the sessions table.

We are going to use the second option. Add the following code to the `session.go` source file.

```
func (ss *SessionService) User(token string) (*User, error) {
        // TODO: Implement SessionService.User
        return nil, nil
}
```

Aside 14.3. Design Decisions and Your Database

A major factor in our design decision here is the database we are using and the way we are structuring our application. We are using a SQL database where `joins` can help us optimize database queries across multiple tables, and we are using a monolith. That is, we are creating a single Go web server to handle all of our incoming traffic.

If using another database, or using microservices, the motivation for one design decision might no longer be valid, so it is important to keep that in mind and opt for the design that works best for each situation.

Source Code

- Source Code - see the final source code for this lesson.
- Diff - see the changes made in this lesson.

14.8 Sessions in the Users Controller

With the `Create` and `User` methods stubbed, we have two tasks left to put everything together:

1. Implement both the `Create` and `User` methods in the `SessionService`.

2. Update our users controller to use the stubbed methods.

Until we complete both of these, our code won't work, but each task is large enough that we want to break it up and focus on one at a time. We are going to start with the second option - updating our users controller - as this will allow us to verify that the functions we stubbed are going to fit our needs. Otherwise we might waste time implementing the **`Create`** or **`User`** method only to later realize it is hard to use. By updating our users controller first and leaving the methods stubbed out we get to see what it is like to use each method with our current design.

Open the users controller source file.

```
code controllers/users.go
```

We are going to update the code where a user creates an account, and the code where a user signs in. In both cases we are going to create a session using the **`SessionService`**, then store the session token in a cookie. Again, *this code will not work properly until we implement the* ***`SessionService`*** *functions*, but we can still use the **`SessionService`** to verify that our function definitions fit our needs.

To do all of this, we need to add the **`SessionService`** to the **`Users`** type so that we can access the method stubs.

```
type Users struct {
        Templates struct {
                New    Template
                SignIn Template
        }
        UserService    *models.UserService
        SessionService *models.SessionService
}
```

Next we can navigate to the **`Create`** handler function where we will use the session service to create a new session after a user account has been created.

Once a session is created we will create and set a cookie to store the session token. Finally, we will redirect the user to the **`/users/me`** path. Long term we will redirect the user to their dashboard, but short term the current user path will help us validate that our sessions are working as intended.

```
func (u Users) Create(w http.ResponseWriter, r *http.Request) {
        email := r.FormValue("email")
        password := r.FormValue("password")
        user, err := u.UserService.Create(email, password)
        if err != nil {
                fmt.Println(err)
                http.Error(w, "Something went wrong.", http.StatusInternalServerError)
                return
        }
        session, err := u.SessionService.Create(user.ID)
        if err != nil {
                fmt.Println(err)
                // TODO: Long term, we should show a warning about not being able to sign the u
                http.Redirect(w, r, "/signin", http.StatusFound)
                return
        }
        cookie := http.Cookie{
                Name:     "session",
                Value:    session.Token,
                Path:     "/",
                HttpOnly: true,
        }
        http.SetCookie(w, &cookie)
        http.Redirect(w, r, "/users/me", http.StatusFound)
}
```

We create a session after successfully creating a user account, so how we handle errors is a bit different. If there is an error creating a session, we are going to redirect the user to the sign in page. Long term we will include a message notifying the user that their account was created, but we were unable to sign them in. We do this to avoid a scenario where the user thinks there was an error creating their account and attempts to sign up again, only to be presented with a confusing error message that their email address is already in use.

If there isn't an error, we use the **`*Session`** returned by the session service to create the session cookie and we set it using **`http.SetCookie`**. Finally, we redirect the user to the **`/users/me`** page where the current user will be

displayed.

Aside 14.4. Redirect Status Code

When calling `http.Redirect` we can provide any HTTP status code, but many web browsers will only redirect a user to a new page when either the 301 Moved Permanenty or the 302 Found status code is used. As a result, it is best to only use these two status codes when redirecting users.

`301` should be used when a page has been permanently moved to a new path. For instance, if we had a blog post with the path `/all-about-dgos` (with a typo) and we later wanted to fix the path to `/all-about-dogs` (no typo), we might use the 301 status code to tell browsers that the blog post has permanently moved to the new path. This allows browsers and search engines to remember this as a permanent change.

In our case, the page has not permanently moved. Instead, we want to send the user to a new page after successfully processing the sign up form. In cases like this, the `302` status code is the best fit.

Next we have the `ProcessSignin` function. The code here will be similar to `Create`, except we won't redirect the user when we encounter an error creating the session because the sole purpose of the handler is to sign a user in and an error should be displayed when that fails. Long term we can update this code to re-render the sign in form and to also display the error message.

```
func (u Users) ProcessSignIn(w http.ResponseWriter, r *http.Request) {
        var data struct {
                Email    string
                Password string
        }
        data.Email = r.FormValue("email")
        data.Password = r.FormValue("password")
        user, err := u.UserService.Authenticate(data.Email, data.Password)
```

```
        if err != nil {
                fmt.Println(err)
                http.Error(w, "Something went wrong.", http.StatusInternalServerError)
                return
        }
        session, err := u.SessionService.Create(user.ID)
        if err != nil {
                fmt.Println(err)
                http.Error(w, "Something went wrong.", http.StatusInternalServerError)
                return
        }
        cookie := http.Cookie{
                Name:     "session",
                Value:    session.Token,
                Path:     "/",
                HttpOnly: true,
        }
        http.SetCookie(w, &cookie)
        http.Redirect(w, r, "/users/me", http.StatusFound)
}
```

Finally, we will update the **`CurrentUser`** function to use the new **`session`** cookie that we have used in the previous two handler functions.

```
func (u Users) CurrentUser(w http.ResponseWriter, r *http.Request) {
        tokenCookie, err := r.Cookie("session")
        if err != nil {
                fmt.Println(err)
                http.Redirect(w, r, "/signin", http.StatusFound)
                return
        }
        user, err := u.SessionService.User(tokenCookie.Value)
        if err != nil {
                fmt.Println(err)
                http.Redirect(w, r, "/signin", http.StatusFound)
                return
        }
        fmt.Fprintf(w, "Current user: %s\n", user.Email)
}
```

We start by retrieving the cookie value, much like we did with the **`email`** cookie. If we are unable to retrieve a user's session cookie, we log the error and redirect them to the sign in page where they can hopefully sign in and get a new cookie.

Once we have a cookie we look up the user via the session token stored in the cookie. Once again, if there is an error we will redirect the user to the sign in page since we have no way of verifying who the user is.

If we don't encounter any errors we can print out the current user's email address.

With all of the code updated, we now have fairly strong evidence that the method stubs we added to the `SessionService` work well for our needs, so we can proceed with implementing them whenever we are ready to do so. Please keep in mind that until we finish implementing the `SessionService` our sign up and sign in processes will not work.

Aside 14.5. Interfaces vs Stubs

In this lesson we used stubbed methods to validate that our design worked well for our needs. Another approach that can be taken in Go is to define an interface and to use it instead of method stubs. The end result ends up being roughly the same, but we wouldn't need to declare the `SessionService` upfront and our controller would be less tightly linked to the `models` package, as it would only require a type that implements the interface.

I find that either approach can work well when learning and first designing an application, so feel free to use whatever works best for you.

Source Code

- Source Code - see the final source code for this lesson.
- Diff - see the changes made in this lesson.

14.9 Cookie Helper Functions

Inside the users controller we create two cookies with the following code:

```
cookie := http.Cookie{
  Name:     "session",
  Value:    session.Token,
  Path:     "/",
  HttpOnly: true,
}
```

In this lesson we are going to clean this code up. We will adding a constant for the cookie name to avoid any bugs due to typos, and we will be consolidating the cookie creation to a single function to avoid any situations where we accidentally forget a security setting like `HttpOnly`.

Create a new file in the controllers package for our cookie-related code.

```
code controllers/cookie.go
```

Aside 14.6. Controllers vs a Separate Package

We will be adding the cookie-related code to our controllers package, but if your application were to grow large enough it may make sense to move this into its own `cookie` package.

Add a constant that stores the name of our session cookie.

```
package controllers

const (
        CookieSession = "session"
)
```

We only have one cookie at the moment, but moving forward we will add the prefix **`Cookie`** to all of our cookie name constants, similar to how the **`net/http`** package has several constants like **`http.StatusNotFound`** and **`http.StatusOK`**. Using a common prefix makes it much easier for developers to find the correct cookie name using their developer tools.

Next, let's add code to create a new cookie. This will help ensure that we are creating cookies with the same secure settings every time. The only variable parts of our cookies will be their name and their value.

```
func newCookie(name, value string) *http.Cookie {
        cookie := http.Cookie{
                Name:     name,
                Value:    value,
                Path:     "/",
                HttpOnly: true,
        }
        return &cookie
}
```

In most cases, we will want to set a cookie immediately after creating it. We can add a helper function to do this all as a single function call while we are at it.

```
func setCookie(w http.ResponseWriter, name, value string) {
        cookie := newCookie(name, value)
        http.SetCookie(w, cookie)
}
```

Finally, we update the code in the users controller.

```
code controllers/users.go
```

The **`Create`** method needs updated first.

```
func (u Users) Create(w http.ResponseWriter, r *http.Request) {
        email := r.FormValue("email")
        password := r.FormValue("password")
        user, err := u.UserService.Create(email, password)
        if err != nil {
                fmt.Println(err)
                http.Error(w, "Something went wrong.", http.StatusInternalServerError)
                return
        }
        session, err := u.SessionService.Create(user.ID)
        if err != nil {
                fmt.Println(err)
                // TODO: Long term, we should show a warning about not being able to sign the u
                http.Redirect(w, r, "/signin", http.StatusFound)
                return
        }
        setCookie(w, CookieSession, session.Token)
        http.Redirect(w, r, "/users/me", http.StatusFound)
}
```

Next, we update the **`ProcessSignIn`** function to use our new **`setCookie`** function.

```
func (u Users) ProcessSignIn(w http.ResponseWriter, r *http.Request) {
        var data struct {
                Email    string
                Password string
        }
        data.Email = r.FormValue("email")
        data.Password = r.FormValue("password")
        user, err := u.UserService.Authenticate(data.Email, data.Password)
        if err != nil {
                fmt.Println(err)
                http.Error(w, "Something went wrong.", http.StatusInternalServerError)
                return
        }
        session, err := u.SessionService.Create(user.ID)
        if err != nil {
                fmt.Println(err)
                http.Error(w, "Something went wrong.", http.StatusInternalServerError)
                return
        }
        setCookie(w, CookieSession, session.Token)
        http.Redirect(w, r, "/users/me", http.StatusFound)
}
```

That covers everything related to writing cookies, but we also read cookies in our code. Let's see if we can improve that process at all.

```
code controllers/cookie.go
```

Add the `readCookie` function.

```
func readCookie(r *http.Request, name string) (string, error) {
        c, err := r.Cookie(name)
        if err != nil {
                return "", fmt.Errorf("%s: %w", name, err)
        }
        return c.Value, nil
}
```

The `readCookie` function returns a cookie's value rather than the `*http.Cookie`, and it will wrap any errors with the name of the cookie we were trying to read. The former just makes it less code when we only care about the value of a cookie, and the latter can help with debugging. In both cases the changes are minor and this entire function could be skipped on future projects.

Since we added the `readCookie` function, let's update our users controller to utilize it.

```
code controllers/users.go
```

The `CurrentUser` function is where we read a session token cookie, so that is what we need to update. Notice that both the first line and the line that calls `SessionService.User` both get updated.

```
func (u Users) CurrentUser(w http.ResponseWriter, r *http.Request) {
        token, err := readCookie(r, CookieSession)
        if err != nil {
                fmt.Println(err)
```

```
                http.Redirect(w, r, "/signin", http.StatusFound)
                return
        }
        user, err := u.SessionService.User(token)
        if err != nil {
                fmt.Println(err)
                http.Redirect(w, r, "/signin", http.StatusFound)
                return
        }
        fmt.Fprintf(w, "Current user: %s\n", user.Email)
}
```

Source Code

- Source Code - see the final source code for this lesson.
- Diff - see the changes made in this lesson.

14.10 Create Session Tokens

We have the **`SessionService`** stubbed in the **`models/session.go`** source file. That is, our methods like **`Create`** and **`User`** don't actually do anything just yet.

```
// This method and the User method are stubbed and need to be completed.
func (ss *SessionService) Create(userID int) (*Session, error) {
  // TODO: Create the session token
        // TODO: Implement SessionService.Create
        return nil, nil
}
```

Inside of the users controller (**`controllers/users.go`**) we use the **`SessionService`** as if it were completed. This helped us verify that the methods we designed on the **`SessionService`** are what we need before implementing them.

With this all done, we are ready to start implementing the **Create** and **User** methods. We won't get this all completed in a single lesson, but we can work towards completing both over the next few lessons.

Open the session model and we will begin there.

```
code models/session.go
```

We are going to start by taking the code we wrote in the **rand** package and using it to setup a **Session** object. While doing this we are going to add a few mor TODOs so we don't forget what extra steps we still need to implement.

```
func (ss *SessionService) Create(userID int) (*Session, error) {
        token, err := rand.SessionToken()
        if err != nil {
                return nil, fmt.Errorf("create: %w", err)
        }
  // TODO: Hash the session token
        session := Session{
                UserID: userID,
    Token: token,
    // TODO: Set the TokenHash
        }
        // TODO: Store the session in our DB
        return &session, nil
}
```

Inside the new code we use the **rand** package to generate a session token. After that we check for an error, and then setup a **Session** object. At the moment we only know the **UserID** and **Token** values for this object, so we add a TODO comment reminding ourselves to add the **TokenHash** field in the future. Lastly, we update the return value to return the session we are constructing.

While the code here is incomplete, it is a step in the right direction. Unfortunately, if we try to run our application and sign in we will see an error message like the one below:

```
runtime error: invalid memory address or nil pointer dereference
```

This occurs because the **SessionService** field of our **Users** controller was never set. Our SessionService isn't 100% complete, but we can address this error and get some of our users controller functionality back to a working state.

Open the **main.go** source file.

```
code main.go
```

Inside the **main()** function, find the code where we declare the **userService** variable and add the code to declare a **sessionService** variable. After that we need to set this as a field in the **usersC** variable that we construct.

```
// in main() find where we declare the userService and add the sessionService code below.
userService := models.UserService{
        DB: db,
}
sessionService := models.SessionService{
        DB:      db,
}

usersC := controllers.Users{
        UserService:    &userService,
        SessionService: &sessionService,
}
```

Now when we try to create a new user our application will still ultimately run into an error, but our application should redirect us to the **/users/me** path before the error occurs, and if we check our cookies we will see that a session cookie is being set.

We are still seeing an error because our **CurrentUser** handler function calls the **SessionService.User** method, and we haven't implmented it yet. As a result, both the user and the error returned from the method are **nil**, and when we try to print out the current user's email address it leads to a nil pointer error.

```
func (u Users) CurrentUser(w http.ResponseWriter, r *http.Request) {
        // ...

  // This line return nil, nil because the User method isn't implemented
        user, err := u.SessionService.User(token)
        if err != nil {
                fmt.Println(err)
                http.Redirect(w, r, "/signin", http.StatusFound)
                return
        }
  // This line causes a panic because the `user` variable is nil
        fmt.Fprintf(w, "Current user: %s\n", user.Email)
}
```

We will address this in a future lesson.

Source Code

- Source Code - see the final source code for this lesson.
- Diff - see the changes made in this lesson.

14.11 Refactor the rand Package

Now that we have updated our **`SessionService`** to use the **`rand`** package, it is a good time to review our initial designs and make sure they still seem like the best way to proceed.

If we look at our **`models/session.go`** source file, we have the following code for generating a session token:

```
func (ss *SessionService) Create(userID int) (*Session, error) {
        token, err := rand.SessionToken()
        // ...
}
```

Inside **rand/rand.go** we have the following code used for the **SessionToken** function:

```
const SessionTokenBytes = 32

func SessionToken() (string, error) {
        return String(SessionTokenBytes)
}
```

When we were designing our **rand** package, we needed somewhere to store this logic, and adding it to **rand.go** felt like a good choice at the time. Now that we have a **SessionService**, and more notably, an entire source file dedicated to storing session-related code, that might be a better place to store some of this code. After all, the SessionService should probably be in charge of dictating how many bytes are used in a session token, not some other package. Let's refactor our code to make this happen.

Open up the session model source file.

```
code models/session.go
```

First, add a constant that states the minimum number of bytes we need for each session token. This will allow us to ensure all of our session tokens are secure, while also allowing the freedom of using more bytes in the future.

```
const (
        // The minimum number of bytes to be used for each session token.
        MinBytesPerToken = 32
)
```

Next, we are going to update the **SessionService** with a field dictating how many bytes we want to use for each session token.

```
type SessionService struct {
        DB      *sql.DB
        // BytesPerToken is used to determine how many bytes to use when generating
        // each session token. If this value is not set or is less than the
        // MinBytesPerToken const it will be ignored and MinBytesPerToken will be
        // used.
        BytesPerToken int
}
```

Finally, we need to update the **`Create`** method to use the new **`BytesPerToken`** field. While doing this, we also need to account for the field not being set, or being set with a value that is too low. In both of these cases, we will default to the minimum number of bytes that we set with a constant. Once we have a valid number of bytes we can call **`rand.String`** with that value.

```
func (ss *SessionService) Create(userID int) (*Session, error) {
        bytesPerToken := ss.BytesPerToken
        if bytesPerToken < MinBytesPerToken {
                bytesPerToken = MinBytesPerToken
        }
        token, err := rand.String(bytesPerToken)
        // ... the rest of the code is unchanged
}
```

While this does force us to write more code every time we want to generate a new session token, this logic is now contained inside the **`SessionService`** where it is easier to find and manage. We also have a lot more flexibility now, as we can use more bytes for session tokens if we want to.

Aside 14.7. Make the Zero Value Useful

One of the Go Proverbs is to "Make the zero value useful." Put simply, this refers to making a variable useful with no construction required on the users part.

An example of this is the **`strings.Builder`** type. We can simply declare a variable with this type and start using it.

```
var sb strings.Builder
// We can us sb now without any setup
```

Having a useful zero value isn't always possible, or even desirable. For instance, our SessionService needs a database connection, and while we could try to use some default values for a connection if one isn't provided, that could mask added costs and cause issues down the road.

That said, we can still apply proverbs where they make sense, and the **`BytesPerToken`** field is a good example of that. While the field can be set by the user, not setting it isn't necessary to use our SessionService type.

We are no longer using the **`SessionToken`** function and the **`SessionTokenBytes`** constant in the **`rand`** package, so we can remove those.

```
code rand/rand.go
```

Delete the following lines from the file.

```
// Delete these lines
const SessionTokenBytes = 32

func SessionToken() (string, error) {
        return String(SessionTokenBytes)
}
```

While the changes we made in this lesson may seem minor, refactors like this help ensure that our code remains easy to maintain as our application grows larger and more complex.

Aside 14.8. Why Are We Refactoring So Often?

In this course we end up refactoring code quite a bit rather than simply writing the final version the first time. I realize that this makes the course longer, and it can give the impression that the course wasn't planned very well. I opted to keep these refactors in the course because I feel it best demonstrates the true development process, and that in itself is valuable for anyone new to web development.

Without seeing these refactors, it is easy to assume that you are doing something wrong if you can't come up with the final version on your first try, but the reality is very few developers could do that. Even the developers that could come up with the final version on their first try often have years of experience to lean on, so in a way they have already made those refactors and have learned to do them in their head now.

It is a good idea to take note of parts of your application that could use some small improvements and refactoring them when time permits. Sometimes this won't be for a while, as we may have deadlines to meet, but this is also why small refactors can be so valuable - they are easy to fit into almost any deadline, and when combined can have a large overall effect.

Source Code

- Source Code - see the final source code for this lesson.
- Diff - see the changes made in this lesson.

14.12 Hash Session Tokens

Our SessionService can generate tokens, but we don't plan to store those directly in our database. Instead, we plan to store the hash of a token in our database. This is why we defined the sessions table in our database with a `token_hash` column.

```
CREATE TABLE sessions (
  id SERIAL PRIMARY KEY,
  user_id INT UNIQUE,
  token_hash TEXT UNIQUE NOT NULL
);
```

We do this to make our application more secure. If we store a session token in our database, an attacker who accesses our database will have access to valid session tokens and can impersonate users. If we only store a hash of the session token, attackers won't be able to reconstruct the original session token to impersonate them.

In order to validate a users session, we will: 1. Read the session token from the request cookies. 2. If a session token is present, we will hash it. If not, we know the user isn't logged in. 3. Once we have a hashed session token we will search our database for a session with the same token hash, which will help us determine who the current user is.

This all means that we need to determine which hashing algorithm we will be using to store our session tokens, then we can proceed with inserting sessions into our database.

While it might be tempting to use `bcrypt` like we did with our passwords, this approach isn't a good fit for session token hashes because it adds a unique salt to every hash. If we were to hash the same session token three times in a row using bcrypt, we would get a different result every time because the salt would be different. The only way to validate a bcrypt hash is to use have access to both the hashed value, and the value we want to test.

```
bcrypt.CompareHashAndPassword(hashedToken, plaintextToken)
```

Technically we could make this work by storing the user's ID in a cookie, and using that to look up their sessions. This would give us access to the hashed tokens, but it would be slow, add unnecessary complexity, and make it more challenging to support multiple sessions per user in the future. What is even worse, we would be doing all of this extra work to use a hashing function that isn't well suited to our needs. After all, our session tokens are:

1. Generated randomly.
2. Replaced every login.
3. Not reused across websites like passwords may be.

The end result is that **`bcrypt`** just doesn't make sense, and we will instead be exploring other hashing functions.

There area few options that we could explore that fit our needs, but **`crypto/sha256`** is probably the most common and best suited hashing algorithm for our needs.

Aside 14.9. Why not HMAC?

In a past version of the course we used **`crypto/hmac`** to hash our session tokens. HMAC works by using a secret key, which is similar to the salt we apply to our passwords before we salt them, except the same secret key is applied to all values that are being hashed. Once the secret key is applied, a hashing algorithm like SHA256 is used to do the actual hashing.

The primary benefit of HMAC is that if an attacker gets access to our database but does NOT have access to our secret key, they cannot even attempt rainbow

table attacks. While this sounds good, in reality our session tokens are randomly generated and replaced at login, so a rainbow table attack is virtually impossible to succeed.

On top of HMAC not really adding any extra security, chances are an attacker who has access to our database will also have access to our secret key, making the use of HMAC pointless.

As a result, HMAC isn't necessary and I opted not to use it in this version of the course to simplify things. If you want to see how HMAC works, you can do so in the v1 of the course.

Looking at the crypto/sha256 docs, there are a few functions and examples to chose from. We are going to use the **Sum256** function, which has the following example in the docs.

```
package main

import (
        "crypto/sha256"
        "fmt"
)

func main() {
        sum := sha256.Sum256([]byte("hello world\n"))
        fmt.Printf("%x", sum)
}
```

The example is pretty short, and the main takeaway is that we will need to call **Sum256** and pass in the data we want to be hashed. This argument needs to be a byte slice, and the **Sum256** function will return a byte array with the hash of the inputs. *It is worth noting that a byte array is different from a byte slice, as we will see shortly.*

Aside 14.10. Printf Hex Encoding

The `%x` part of the fmt.Printf function call is used to print out bytes using hex encoding. This can be handy if you have an array or slice of bytes and need to print them in a readable format, but don't actually intend to use that hex encoding format beyond printing out the bytes.

Our results are a byte slice, so we are going to convert that into a string before storing it in our database. We can do this using `base64`.

Open up the session model source file.

```
code models/session.go
```

Add the following method to the SessionService. It will handle taking in a session token as a string and returning the hash of that session token.

```
func (ss *SessionService) hash(token string) string {
        tokenHash := sha256.Sum256([]byte(token))
        // base64 encode the data into a string
        return base64.URLEncoding.EncodeToString(tokenHash[:])
}
```

As long as we consistently use this function to hash our tokens. The function is only two lines of code, but I find that separating important logic like this can be handy, especially if we want to unit test the code at some point.

Aside 14.11. Why Do We Use the [:] Syntax?

When we call `Sum256`, it doesn't return a byte slice and instead returns a fixed-size byte array. While similar, there are some differences between the two and the compiler treats them differently. As a result, if we tried to pass the `tokenHash` variable directly into the `EncodeToString` function, we would get an error about the type not matching.

To resolve this error, we use `tokenHash[:]`, which tells Go that we want to create a byte slice using all of the data inside the byte array. This is a shorthand for `tokenHash[0:len(tokenHash)]`. In other words, we want the byte slice to use the contents starting at index `0`, and and ending at the last index in the array.

We could also create a byte slice using a subset of the byte array with something like `tokenHash[5:10]`.

With our `hash` function created, we can update `Create` to utilize it.

```
func (ss *SessionService) Create(userID int) (*Session, error) {
        bytesPerToken := ss.BytesPerToken
        if bytesPerToken < MinBytesPerToken {
                bytesPerToken = MinBytesPerToken
        }
        token, err := rand.String(bytesPerToken)
        if err != nil {
                return nil, fmt.Errorf("create: %w", err)
        }
        session := Session{
                UserID:    userID,
                Token:     token,
                TokenHash: ss.hash(token),
        }
        // TODO: Store the session in our DB
        return &session, nil
}
```

Before moving on, it is worth noting that a lot of the logic we added to the SessionService is dedicated to generating session tokens. While we won't be doing this, another way to structure this code might be to create a `TokenManager`

type and isolate any code related to generating and hashing tokens to methods on that type. We could then add the `TokenManager` as a field on the `SessionService` type, where it could use the token manager.

```
// Do not use this code. It is for illustration purposes only
type TokenManager struct {}

// We could provide function(s) to handle all of our token-related logic.
func (tm TokenManager) New() (token, tokenHash string, err error) {
  // ...
}

func (tm TokenManager) Hash(token string) string {
        // ...
}
```

Using code like this could simplify our SessionService.Create function quite a bit, and it might make testing some of our logic a bit easier in isolation. In our case we won't be doing this as our session service doesn't ever get too complicated, so it is quite manageable with the token logic embedded inside of it. It is being pointed out as an alternative way to organize your code that once again will have its own pros and cons.

Source Code

- Source Code - see the final source code for this lesson.
- Diff - see the changes made in this lesson.

14.13 Insert Sessions into the Database

Now that we have the code to build a valid `models.Session` object. It is time to add the code to insert all of that data into our database. To do this, we will be working inside the `sessions.go` file inside the `models` package.

```
code models/session.go
```

The `id` will be set by the database, so we will need to return that value after inserting a record. The unhashed token won't be stored, so we can ignore that field. That leaves us with the following SQL to insert a session into our database table.

```
INSERT INTO sessions (user_id, token_hash) VALUES (1, 'xyz-456') RETURNING id;
```

Turning this into Go code we get the following.

```
row := ss.DB.QueryRow(`
  INSERT INTO sessions (user_id, token_hash)
  VALUES ($1, $2)
  RETURNING id;`, session.UserID, session.TokenHash)
err = row.Scan(&session.ID)
if err != nil {
  return nil, fmt.Errorf("create: %w", err)
}
```

Now we can put this all together and add it to the `SessionService.Create` method.

```
func (ss *SessionService) Create(userID int) (*Session, error) {
        bytesPerToken := ss.BytesPerToken
        if bytesPerToken < MinBytesPerToken {
                bytesPerToken = MinBytesPerToken
        }
        token, err := rand.String(bytesPerToken)
        if err != nil {
                return nil, fmt.Errorf("create: %w", err)
        }
        session := Session{
                UserID:    userID,
                Token:     token,
                TokenHash: ss.hash(token),
        }
```

```
        row := ss.DB.QueryRow(`
                INSERT INTO sessions (user_id, token_hash)
                VALUES ($1, $2)
                RETURNING id;`, session.UserID, session.TokenHash)
        err = row.Scan(&session.ID)
        if err != nil {
                return nil, fmt.Errorf("create: %w", err)
        }
        return &session, nil
}
```

Restart the server and verify this code is working. Head to the signup page and create a new account. We will still see an error in our browser and in the Go code, but we can still verify that everything is working by:

1. Checking our browser for a session cookie.
2. Checking the database for an entry in the session table.

The first can be done with an extension, or by looking into the browser settings for cookies.

The second can be achieved by using **`docker compose exec`** like we have in the past to run **`psql`**. Once we have connected to our database, we can run the following SQL.

```
SELECT * FROM sessions;
```

This query could also be performed from within Adminer.

It is worth noting that while our code works to create a session, it will result in an error if the same user signs in, as this will attempts to create a new session and our sessions must have a unique **`user_id`**. We will see how to address this in the next lesson.

Source Code

- Source Code - see the final source code for this lesson.
- Diff - see the changes made in this lesson.

14.14 Updating Existing Sessions

We defined our sessions table to have a unique `user_id` so that we only have one session per user.

```
CREATE TABLE sessions (
  id SERIAL PRIMARY KEY,
  user_id INT UNIQUE,
  token_hash TEXT UNIQUE NOT NULL
);
```

While this is how we wanted out data structured, our code doesn't currently take this into account. When a user signs in our code attempts to create a new session. This will ultimately lead to an error, as we will be attempting to create a second session with the same `user_id` column.

There are a few ways to approach this, but in each case we need to take into account the fact that when a user signs out, we will very likely be deleting their session from the database, so there will definitely be times where we want to create a new session when a user signs in, and at other times we will want to update an existing session object.

The mostly database agnostic approach to this might be to:

1. Query the database for a session via the user ID.
2. If a session is found, update it with a new session token hash.
3. If a session is not found, create a new session for that user like we have been.

Alternatively, we could just assume that a session exists for every user, which would give us the following process:

1. Attempt to update the user's session with a new session token hash.
2. If no records could be updated, we create a new session for that user.

At the moment, the primary benefit to this approach is that we avoid needing to write code that queries for a session via the user ID. In the future we will also see how we can improve our SQL query further, turning this entire process into a single PostgreSQL query.

The SQL we will be starting with to update a session is shown below.

```
UPDATE sessions SET token_hash='111' WHERE user_id=1 RETURNING id;
```

We will place this inside the `SessionService.Create` method so that we have a single method that both creates and updates existing sessions. While we technically aren't always creating a new session in our database, any code that calls this function will receive a new `models.Session` object that we create, so `Create` still feels like the correct name for our function.

```
code models/session.go
```

We need to add our update query just after we construct the `Session` object. If that query fails, we can check what the error was and determine how to proceed. If it was an `sql.ErrNoRows` error, we know that the record wasn't updated and we need to craete a new session. In that case we will run the `INSERT INTO...` query that we had before. Otherwise, we will return an error, as this suggests an error that we weren't expecting.

Putting that all together we get the code below.

```
func (ss *SessionService) Create(userID int) (*Session, error) {
        bytesPerToken := ss.BytesPerToken
        if bytesPerToken < MinBytesPerToken {
                bytesPerToken = MinBytesPerToken
        }
        token, err := rand.String(bytesPerToken)
        if err != nil {
                return nil, fmt.Errorf("create: %w", err)
        }
        session := Session{
                UserID:    userID,
                Token:     token,
                TokenHash: ss.hash(token),
        }
        row := ss.DB.QueryRow(`
                UPDATE sessions
                SET token_hash = $2
                WHERE user_id = $1
    RETURNING id;`, session.UserID, session.TokenHash)
        err = row.Scan(&session.ID)
        if err == sql.ErrNoRows {
                // If no session exists, we will get ErrNoRows. That means we need to
                // create a session object for that user.
                row = ss.DB.QueryRow(`
                        INSERT INTO sessions (user_id, token_hash)
                        VALUES ($1, $2)
                        RETURNING id;`, session.UserID, session.TokenHash)
                // The error will be overwritten with either a new error, or nil
                err = row.Scan(&session.ID)
        }
        // If the err was not sql.ErrNoRows, we need to check to see if it was any
        // other error. If it was sql.ErrNoRows it will be overwritten inside the if
        // block, and we still need to check for any errors.
        if err != nil {
                return nil, fmt.Errorf("create: %w", err)
        }
        return &session, nil
}
```

While this approach works very well, we will also explore some ways to improve and optimize the SQL using the PostgreSQL specific **`ON CONFLICT`** in the next section of the course where we talk about improving our SQL knowledge and skills. While this is specific to Postgres, almost every other database variant has something similar that could also be utilized.

Sidenote: Our application still will not work fully, as we haven't finished im-

plementing all of the handlers and functions needed for logging in.

Source Code

- Source Code - see the final source code for this lesson.
- Diff - see the changes made in this lesson.

14.15 Querying Users via Session Token

We are now read yto implement the `SessionService.User` function. We are going to do this in four steps:

1. We will first hash the session token passed into the function.
2. Next, we will query for a session with that token hash.
3. As long as a user is found, we will use the user ID from the result to query for a user with that ID.
4. Finally, we will return the user associated with the session token.

Open the session model to get started.

```
code models/session.go
```

Let's first document the steps we need to take in our code.

```
func (ss *SessionService) User(token string) (*User, error) {
        // TODO: Implement SessionService.User
        // 1. Hash the session token
        // 2. Query for the session with that hash
        // 3. Using the UserID from the session, we need to query for that user
        // 4. Return the user
        return nil, nil
}
```

Now we can incrementally handle each step, starting with hashing the token which is a function call to our **`SessionService.hash`** function.

```
// 1. Hash the session token
tokenHash := ss.hash(token)
```

Next, we need to query for the session with that token hash, which leads us to the following code.

```
// 2. Query for the session with that hash
var userID int
row := ss.DB.QueryRow(`
        SELECT user_id
        FROM sessions
        WHERE token_hash = $1;`, tokenHash);
err := row.Scan(&userID)
if err != nil {
        return nil, fmt.Errorf("user: %w", err)
}
```

Now that we have a user ID, we can use it to query for a user.

```
// 3. Using the UserID from the session, we need to query for that user
var user User
row = ss.DB.QueryRow(`
        SELECT email, password_hash
        FROM users WHERE id = $1;`, userID)
err = row.Scan(&user.Email, &user.PasswordHash)
if err != nil {
        return nil, fmt.Errorf("user: %w", err)
}
```

Next, we need to return the user we created.

```
// 4. Return the user
return &user, nil
```

Finally, we can clean up our code a bit by creating the `user` variable earlier in our code, and then scanning the user ID directly into the `user.ID` field. That gives us the following `User` method.

```
func (ss *SessionService) User(token string) (*User, error) {
        tokenHash := ss.hash(token)
        var user User
        row := ss.DB.QueryRow(`
                SELECT user_id
                FROM sessions
                WHERE token_hash = $1;`, tokenHash)
        err := row.Scan(&user.ID)
        if err != nil {
                return nil, fmt.Errorf("user: %w", err)
        }
        row = ss.DB.QueryRow(`
                SELECT email, password_hash
                FROM users WHERE id = $1;`, user.ID)
        err = row.Scan(&user.Email, &user.PasswordHash)
        if err != nil {
                return nil, fmt.Errorf("user: %w", err)
        }
        return &user, nil
}
```

To test this code we simply need to visit the `/users/me` page after signing up or signing in, as this will attempt to lookup a user via their session token. This will render the current user's email address.

Aside 14.12. Improved SQL

The SQL we are using here isn't optimized. In the following section we will look at how to use a `JOIN` to optimize the `User` function to work with a single SQL query.

Source Code

- Source Code - see the final source code for this lesson.
- Diff - see the changes made in this lesson.

14.16 Deleting Sessions

Our sessions now allow users to sign in and remain signed in, but users still don't have a way to sign out. To add this functionality, we are going to start by adding a method to `SessionService` type that handles deleting a session. After that we can look at how to set up our links and HTTP handlers.

When deleting a session, we need some way to tell our database which specific session to delete. That means we could use any of the following SQL queries to delete a session:

```
-- Any of these would delete a session
DELETE FROM sessions
WHERE token_hash = $1;

DELETE FROM sessions
WHERE user_id = $1;

DELETE FROM sessions
WHERE id = $1;
```

We know that our controllers have access to the session token via the user's cookie, so turning this into a hash and deleting based on the `token_hash` is likely the best choice for our application. Let's add that code to our SessionService.

```
code models/session.go
```

We will our method `Delete`, and it will be very similar to code we have written already.

```
func (ss *SessionService) Delete(token string) error {
        tokenHash := ss.hash(token)
        _, err := ss.DB.Exec(`
                DELETE FROM sessions
                WHERE token_hash = $1;`, tokenHash)
        if err != nil {
                return fmt.Errorf("delete: %w", err)
        }
        return nil
}
```

We are using `Exec` here instead of `QueryRow` because we don't care about any data being returned by the database. If we did care about getting any data from our query, we would need to use something like `QueryRow`.

That's it for this lesson! We will work on adding the handler that uses this in the next lesson, along with a link for users to click.

Source Code

- Source Code - see the final source code for this lesson.
- Diff - see the changes made in this lesson.

14.17 Sign Out Handler

When a user signs out, we want to attempt to do two things:

1. Delete or invalidate the session in the database.

2. Delete the user's session cookie.

If we only did one of these the user would still be signed out, but it is best to do both.

The first is by far the most important, as it is necessary to invalidate the session cookie entirely. Without deleting the session from the database, it is technically possible for a stolen cookie to continue to work until the user signs in again.

We will first open the users controller source file.

```
code controllers/users.go
```

Here we are going to add the **`ProcessSignOut`** function to the **`Users`** type. This function will be used to handle a web request to sign out a user, which means it will need to read their session from the cookie, then use the session service to delete the session associated with that token.

```
func (u Users) ProcessSignOut(w http.ResponseWriter, r *http.Request) {
        token, err := readCookie(r, CookieSession)
        if err != nil {
                http.Redirect(w, r, "/signin", http.StatusFound)
                return
        }
        err = u.SessionService.Delete(token)
        if err != nil {
                fmt.Println(err)
                http.Error(w, "Something went wrong.", http.StatusInternalServerError)
                return
        }
        // TODO: Delete the user's cookie
        http.Redirect(w, r, "/signin", http.StatusFound)
}
```

Next, we need to add the code used to delete a cookie. We have all of our existing cookie code in the **`cookie.go`** source file, so we will add the **`deleteCookie`** function there as well.

```
code controllers/cookie.go
```

We can overwrite a cookie by setting a new cookie with the same name, but different attributes. We can utilize this to delete a cookie by setting a new cookie with the same name, but with a **MaxAge** attribute set to a value less than zero. This will tell our browser that the cookie is no longer valid (it has gone past its max age) and it will be deleted.

In code this looks like the function below.

```
func deleteCookie(w http.ResponseWriter, name string) {
        cookie := newCookie(name, "")
        cookie.MaxAge = -1
        http.SetCookie(w, cookie)
}
```

We can now go back to the users controller to utilize this function.

```
code controllers/users.go
```

Once we have deleted teh session we want to delete the cookie by calling **deleteCookie**.

```
func (u Users) ProcessSignOut(w http.ResponseWriter, r *http.Request) {
        // ...
        deleteCookie(w, CookieSession)
        http.Redirect(w, r, "/signin", http.StatusFound)
}
```

Lastly, we need to add our new HTTP handler to our routes. We have these stored in the main function, so we open that source file.

```
code main.go
```

Head to the code where the rest of our routes are located and add the following route for handling sign outs.

```
r.Post("/signout", usersC.ProcessSignOut)
```

While it might seem more intuitive to use `r.Delete` and the delete HTTP method, in practice it is quite annoying to create links and forms that perform a `DELETE` without the use of JavaScript, so we will be using `POST`.

We do NOT want to use a `GET`, as these links are sometimes followed by browsers and JS libraries that try to cache pages to speed up the browsing experience. In general, any page that alters data shouldn't be accessible via a GET.

In the next lesson we will add a sign out link and test all of this code.

Source Code

- Source Code - see the final source code for this lesson.
- Diff - see the changes made in this lesson.

14.18 Sign Out Link

Our last step in allowing users to sign out is to provide them with a button or link to click that leads to them signing out. While it would be nice to use an `<a>` tag and create a normal link, we can't make an HTML link perform a `POST` or `DELETE`. Fortunately, we can use HTML forms to perform a `POST`, and then we can style the button to submit the form to look like a normal link.

Open **tailwind.gohtml** where we will be adding the sign out button next to the sign in button.

```
code templates/tailwind.gohtml
```

Find the section of code that has the "Sign in" and "Sign up" links. It should look similar to the code below.

```
<div class="space-x-4">
  <a href="/signin">Sign in</a>
  <a href="/signup" class="px-4 py-2 bg-blue-700 hover:bg-blue-600 rounded">
    Sign up
  </a>
</div>
```

Inside of the **div**, add the following HTML form that can be used for signing out. We won't be editing any of our existing links, and for now the sign out link will always appear even if a user isn't signed in. We can fix that bug later.

```
<div class="space-x-4">
  <form action="/signout" method="post" class="inline pr-4">
    <div class="hidden">
      {{csrfField}}
    </div>
    <button type="submit">Sign out</button>
  </form>
  <a href="/signin">Sign in</a>
  <a href="/signup" class="px-4 py-2 bg-blue-700 hover:bg-blue-600 rounded">
    Sign up
  </a>
</div>
```

As a quick recap, we are using an HTML form to perform a POST request because browsers don't support **DELETE** well, and **POST** requires a form. When we use an **<a>** link, these always result in a **GET** request. Because we need to use a form, it is important to remember to include the **{{csrfField}}**, otherwise our form won't work.

Another thing to note is that Tailwind strips all styling from things like form buttons, so we don't need to do anything extra to style our "Sign out" button like a regular link. If we were not using Tailwind, we would get what looks like a button and we would need to use CSS to style it to look like a regular link.

With our sign out link in place, we can verify that the sign out process works using the steps below.

1. Sign into an account (or sign up for a new one).
2. Verify that the account is signed in by visiting the `/users/me` and ensuring that it renders your email address correctly.
3. Navigate to the home page (or any other page with the navigation bar) and click the "Sign out" link.
4. Verify that we are redirected to the sign in page.
5. Head to `/users/me` and verify that we are redirected to the sign in page.
6. Use `psql` via `docker compose exec` (or Adminer) to verify that our session was deleted in the database.
7. In our browser, look at the cookies and verify that the session cookie was deleted.

If all of these steps succeeded we are done implementing the sign out process.

Source Code

- Source Code - see the final source code for this lesson.
- Diff - see the changes made in this lesson.

14.19 Session Exercises

14.19.1 Ex1 - Move the token management code to its own type

Try extracting the token creation logic into a new type. Have that type offer a function like:

```
func (tm TokenManager) New() (token, tokenHash string, err error) {
  // TODO: Implement this
}
```

Update the SessionService to use the TokenManager.

The goal here is to explore other ways of organizing code and learning what you prefer. Try to think about the pros and cons of each approach.

Does one take more code? Does one look easier to test individual units?

14.19.2 Ex2 - Add a template for the current user page

Update the CurrentUser handler in the users controller to use a template and render an HTML page that shows who the current user is, and also has a button to sign out.

14.19.3 Ex3 - Try using a JOIN in SQL

Try updating the `SessionService.User` method to use a JOIN to look up the user with a single query.

We will cover it in the next section, so this is just getting an early start on what we will be doing.

Chapter 15

Improved SQL

15.1 SQL Relationships

Our application currently has one relationship defined in the `sessions` table.

```
CREATE TABLE sessions (
  id SERIAL PRIMARY KEY,
  user_id INT UNIQUE,
  token_hash TEXT UNIQUE NOT NULL
);
```

In this table we are saying that each session is related to a user when we add the `user_id` column.

Specifically, we are saying that each session *belongs to a user*. We could also say that a user *has one session*. This is singular because of the `UNIQUE` clause on the `user_id`.

Relationships can come in a variety of arrangements, and records can be associated with many other tables.

Imagine that we wanted to create an app with users, blog posts, and comments.

How might that data look?

We need the following data to render a blog post:

```
// Blog Post
{
  "id": 22,
  "title": "Demo Post",
  "markdown": "This is a demo blog post...",
  "comments": [
    {
      "author": {
        "id": 123,
        "email": "jon@calhoun.io",
        "username": "joncalhoun",
      },
      "markdown": "This was a great post!"
    }
  ]
}
```

With a struct shaped roughly like this JSON, we could render any blog post.

If we were using a document store like MongoDB, we might store out data in this format in our database. Pro: We could query any blog post by looking up a single document! Con: If we want to update a user's display name, we need to go through every post and update it. Result: Fast for reads, but a little slower and harder to manage updates.

With SQL relationships, we can take another approach. We may first start with a table for blog posts:

```
CREATE TABLE posts (
  id SERIAL PRIMARY KEY,
  title TEXT NOT NULL,
  markdown TEXT NOT NULL
);
```

Next is a table for users:

```
CREATE TABLE users (
  id SERIAL PRIMARY KEY,
  email TEXT UNIQUE NOT NULL,
  display_name TEXT NOT NULL
);
```

Finally, we need a way to comment on a blog post. We could do this with a **comments** table that defines both the comment text, as well as the necessary relationships.

```
CREATE TABLE comments (
  id SERIAL PRIMARY KEY,
  user_id INT,
  post_id INT,
  markdown TEXT NOT NULL
);
```

The relationships defined by the **comments** table are: - Comments belong to a user. - Comments belong to a post.

We could also say: - A user has many comments - A post has many comments

With these relationships we could query for a blog post, its comments, and the information for users who created those comments.

```
SELECT * FROM posts
  JOIN comments ON posts.id = comments.post_id
  JOIN users ON users.id = comments.user_id;
```

We will learn more about **JOIN** later, but the short version is that we are querying our database for all the information we need at once.

Pros: - Updates are faster/easier. Updating a user's display name doesn't require us to touch the posts table. - We can query for things we didn't plan for. For instance, we could query for all of the posts that a specific user commented

on. Cons: - This approach can be slightly less performant when reading data because we need to read from multiple DB tables. Result: - In practice SQL databases and relationships offer a solid middle ground for applications. It balances both write and read speeds, and we can optimize things later when/if the need ever arises. - It also gives us flexibility for new queries in the future.

As we saw earlier, our app has one relationship right now:

- A user has one session (or a session belongs to a user)

This is shown via the `user_id` in the sessions table, and we know it is a singular relationship because the `user_id` must be unique.

```
CREATE TABLE sessions (
  id SERIAL PRIMARY KEY,
  user_id INT UNIQUE,
  token_hash TEXT UNIQUE NOT NULL
);
```

As we build more complex apps, the number of relationships will expand. Eventually our users will have many galleries, and each gallery will have many photos.

When working with relationships, SQL provides a number of utilities to help us.

- We can use `JOIN` queries to write a single query that returns all of the related data we are interested in.
- We can use foreign keys to ensure that we don't have invalid relationship data.
- We can use indexes to speed up queries related to these relationships.

In this section we will be looking at some of those features before we move on with building our app.

15.2 Foreign Keys

Foreign keys are a way of telling a SQL database about a relationship and asking it to help us ensure the data is valid.

For instance, we could tell our database that the `user_id` in the session is a foreign key referencing a user's ID.

```
CREATE TABLE sessions (
  id SERIAL PRIMARY KEY,
  user_id INT UNIQUE,
  token_hash TEXT UNIQUE NOT NULL,
  FOREIGN KEY (user_id) REFERENCES users (id)
);

-- OR

CREATE TABLE sessions (
  id SERIAL PRIMARY KEY,
  user_id INT UNIQUE REFERENCES users (id),
  token_hash TEXT UNIQUE NOT NULL
);
```

Adding this foreign key tells our database that the `user_id` field is in reference to the `id` field in the `users` table. In other words, `user_id` in this table should match an `id` in our users table. We don't need the foreign key line in order to make this relationship work. *We saw this with our code - we can query for a user's session without any foreign keys.*

Adding the foreign key causes the database to do some extra checks for us. - If we add it, we cannot create a session with a user_id that does't match a valid ID in our users table. - If we try to delete a user while they have sessions, we will see an error because this would lead to invalid data. We would first need to delete those sessions.

```
DELETE FROM users WHERE id=1;
```

Gives us the output:

```
ERROR:  update or delete on table "users" violates foreign key constraint "sessions_user_id_fke
DETAIL:  Key (id)=(1) is still referenced from table "sessions".
```

```
INSERT INTO sessions (user_id, token_hash) VALUES (123, 'abc');
```

```
ERROR: TODO: Fill this in.
```

Foreign keys are a nice way to improve our data integrity.

15.2.1 Existing table

We saw how to do this when creating a table. If a table already exists, we can add a foreign key with the **`ALTER TABLE`** command.

```
ALTER TABLE sessions
  ADD CONSTRAINT sessions_user_id_fkey FOREIGN KEY (user_id) REFERENCES users (id);
```

`sessions_user_id_fkey` is a name I chose to represent the foreign key. - fkey is shorthand for “foreign key” - sessions is the table the key is applied to - user_id is the field the foreign key is based on Could be any name you want.

15.2.2 Updating our code

Open the **`sessions.sql`** file.

```
code models/sql/sessions.sql
```

Update it with the following SQL.

```
CREATE TABLE sessions (
  id SERIAL PRIMARY KEY,
  user_id INT UNIQUE REFERENCES users (id),
  token_hash TEXT UNIQUE NOT NULL
);
```

We are going to make more changes before proceeding, so no need to recreate the DB table just yet.

Source Code

- Source Code - see the final source code for this lesson.
- Diff - see the changes made in this lesson.

15.3 On Delete Cascade

In addition to adding a foreign key, we can also add some options about what to do when an object is deleted and it has foreign keys associated with it.

For instance, we can add the `ON DELETE CASCADE` option to our sessions.

```
CREATE TABLE sessions (
  id SERIAL PRIMARY KEY,
  user_id INT UNIQUE REFERENCES users (id) ON DELETE CASCADE,
  token_hash TEXT UNIQUE NOT NULL
);
```

This tells our database that is a user is deleted, it should also delete all of that user's sessions. This is handy for ensuring data integrity and avoiding foreign key errors, but be careful since it does result in permanently deleted data.

Now if we were to run the a query to delete a user, we wouldn't get an error even if they had a sessoin. Instead, that session would be deleted.

Open the `sessions.sql` file.

```
code models/sql/sessions.sql
```

Update it with the new sessions table that has a foreign key and the ON DELETE.

```
CREATE TABLE sessions (
  id SERIAL PRIMARY KEY,
  user_id INT UNIQUE REFERENCES users (id) ON DELETE CASCADE,
  token_hash TEXT UNIQUE NOT NULL
);
```

Source Code

- Source Code - see the final source code for this lesson.
- Diff - see the changes made in this lesson.

15.4 Inner Join

Now that we have our relationship defined with a foreign key, let's look at how to use a JOIN to look up related data in a single query.

15.4.1 Prep the DB

Run `docker compose down` to delete all of the data in the docker containers and stop them.

Next run `docker compose up -d` to run the containers in the background. This will also set the database back up, but it won't have any tables.

Open up `psql` or Adminer and copy/paste the `users.sql` and `sessions.sql` files into our database.

```
CREATE TABLE users (
  id SERIAL PRIMARY KEY,
  email TEXT UNIQUE NOT NULL,
  password_hash TEXT NOT NULL
);

CREATE TABLE sessions (
  id SERIAL PRIMARY KEY,
  user_id INT UNIQUE REFERENCES users (id) ON DELETE CASCADE,
  token_hash TEXT UNIQUE NOT NULL
);
```

Our DB should be reset and ready to run queries.

15.4.2 Create records to query

Start the application either using `modd` or `go run`.

```
go run main.go
```

Head to the signup page and create a few user accounts. Preferably at least three.

15.4.3 Using JOIN

Open `psql` or Adminer and run the following SQL query.

```
SELECT * FROM users
  JOIN sessions ON users.id = sessions.user_id;
```

This will return a table similar to the one below, but the exact value will vary depending on the user accounts created.

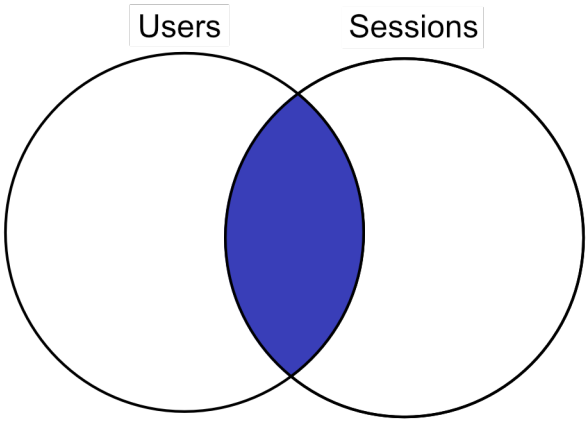
id	email	password_hash	id	user_id	token_hash
1	“bob@bob.com”	$2a$10...	6	1	66iB4lc...
2	“jon@calhoun.io”	$abc12...	10	2	hA41dP0...
4	“ex@ample.com”	$8d1ad...	13	4	ZQ12nm1...

When we run a join, we are telling our database to find all of the records that match our criteria of **`users.id`** being equal to the **`sessions.user_id`**.

If a user has no sessions, the user won’t show up with this query. Delete a session, then re-run the query. That user will no longer show up in the results.

Similarly, if a session points to an invalid user_id or had no user_id, it would not be present with this query. This isn’t possible with our foreign key, but if that foreign key weren’t present it would be worth knowing.

When we use **`JOIN`** we are using what is called an **`INNER JOIN`**. An inner join finds records where the two tables overlap.



Other types of JOINs exist and will be covered briefly in the next lesson.

15.5 Left, Right, and Full Outer Join

By default, a `JOIN` in SQL is really an `INNER JOIN`. This means that it only returns the records with an entry from both tables. In our case of users and sessions, it only returns users with a valid session associated with them.

There are times when we may want to include records even if a related record doesn't exist.

For instances, if we wanted all of our users, even if they don't have a valid session.

If we were to delete a session then run the following SQL, we would still see all the users in our results.

```
SELECT * FROM users
  LEFT JOIN sessions ON users.id = sessions.user_id;
```

In this case the table we are querying is the **users** table, and we are joining it with the **sessions** table.

A **LEFT JOIN** will include all items from the table we are querying, even if they do not have an associated record in the table we are joining.

Similarly, we can also perform a **RIGHT JOIN**. This will return all items from the joined table, even if they do not have an associated record in the first table.

Lastly, we can do a **FULL OUTER JOIN** which will return every record from both tables, regardless of whether there is a related record.

In all of these cases we can filter our results with **WHERE**. The type of join only dictates whether a record will be returned or not based on whether it has a record with the correct relationship.

In all of these instances, if the record returned doesn't have an associated record we will have null or empty values for those fields.

We will discuss why we are using each specific SQL query as we progress, so there is no need to master joins at this point. It is only essential to be aware that they exist and to have a broad understanding of what they allow us to do.

15.6 Using Join in the SessionService

Just like any other query, we can be more selective about the data we want, and we can add clauses to filter out data.

Let's start with a query to select users joined with their session.

```
SELECT * FROM users
  JOIN sessions ON users.id = sessions.user_id;
```

We can specify the fields we want just like we did before, but with one caveat; it is typically best to prefix the fields with the table name in case the two tables have overlapping field names.

```
SELECT users.id,
  users.email,
  users.password_hash,
  sessions.id
FROM users
  JOIN sessions ON users.id = sessions.user_id;
```

We can also rename fields as needed.

```
SELECT users.id,
  users.email,
  sessions.id as session_id
FROM users
  JOIN sessions ON users.id = sessions.user_id;
```

This won't change anything in our database, but the table returned by SQL will use this name.

```
 id |         email          | session_id
----+------------------------+------------
  2 | jon@calhoun.io         |          2
  4 | jim@dundermifflin.com  |          4
  5 | pam@dundermifflin.com  |          5
```

When placing the data into a Go struct, this typically doesn't matter, but it can make more advanced queries easier at times.

We can even add a **`WHERE`** clause to limit the results.

```
SELECT users.id,
  users.email,
  users.password_hash,
  sessions.id
FROM users
  JOIN sessions ON users.id = sessions.user_id
WHERE sessions.token_hash = $1;
```

Finally, we don't even need to return data from both tables. Sometimes we want to query for a specific item in a table, then retrieve a related record.

Let's look at this using our application code. Open the SessionService source file.

```
code models/session.go
```

If we look at the **`User`** method, we currently query for a session with a given token hash, then we use that session to query for the user. This requires us to perform two SQL queries.

We can instead use a JOIN to perform this lookup in a single query.

```
SELECT users.id,
  users.email,
  users.password_hash
FROM sessions
  JOIN users ON users.id = sessions.user_id
WHERE sessions.token_hash = $1;
```

We are using an inner join, so this query will not return any data unless we find a valid session with the required token hash.

If a session is found, information about the user associated with it will be returned.

Let's update our code to use this new query.

```
func (ss *SessionService) User(token string) (*User, error) {
        tokenHash := ss.hash(token)
        var user User
        row := ss.DB.QueryRow(`
                SELECT users.id,
      users.email,
      users.password_hash
    FROM sessions
      JOIN users ON users.id = sessions.user_id
    WHERE sessions.token_hash = $1;`, tokenHash)
        err := row.Scan(&user.ID, &user.Email, &user.PasswordHash)
        if err != nil {
                return nil, fmt.Errorf("user: %w", err)
        }
        return &user, nil
}
```

Source Code

- Source Code - see the final source code for this lesson.
- Diff - see the changes made in this lesson.

15.7 SQL Indexes

SQL indexes are used to help us speed up queries for specific data. They do this by storing that data in a sorted manner. When data is sorted, we can use much faster search algorithms. When data is not sorted, the only way to search is to look at every record one by one.

To visualize this, think of adding data to a table like adding an entry to a spreadsheet. By default the spreadsheet is sorted by the primary key - in our case the ID column.

A	B	C
id	**email**	**password_hash**
1	jon@calhoun.io	...
2	bob@bob.com	...
4	pam@dundermifflin.com	...
6	jim@dundermifflin.com	...
7	demo@user.com	...

Even though we aren't computers, this still helps us find a user with a specific ID much faster. What about searching for a user with a specific email address? Our example only has a few users, but imagine we had millions and the email addresses were in a random order.

Computers need to look at every record to check if a user has a specific email address. Adding an index can fix this, allowing SQL to query for users with a specific email much faster.

Going back to our spreadsheet analogy, adding an index is similar to copying the `id` and `email` columns from to a new spreadsheet, then sorting that data by the `email` column.

A	B	C
id	**email**	
2	bob@bob.com	
7	demo@user.com	
6	jim@dundermifflin.com	
1	jon@calhoun.io	
4	pam@dundermifflin.com	

While this isn't a perfect analogy to how SQL indexes work, it is close enough to help illustrate how they work.

Pro: We can now search by email address much faster, and then use the `id` to find the rest of that user's information in the original spreadsheet.

Con: If a user updates their email address, we need to update the spreadsheet that is sorted by emails.

SQL indexes can improve the speed of specific queries, but they can also increase the cost of updates.

If we also sorted users by their age, username, height, and favorite color we could end up updating quite a few spreadsheets if a user updates their information.

End result - we need to find a careful balance of indexing the important data, but not indexing too much.

A few examples of common things that may need indexed: - Any fields that are frequently used for querying a record. - Eg we query sessions via the token_hash every time we lookup who a user is via their cookie. - We also lookup users via their email every time they log in, so this might be another good candidate.

- Columns used frequently for joins.
  - sessions.user_id
  - many foreign keys will fall into this category, but some may be used infrequently enough to not index them.
- Columns with a `UNIQUE` or `PRIMARY KEY` constraint
  - Postgres automatically creates a unique index when these exist.

Another factor is the size of our database table. If we have a small table that we don't expect to grow, but we do plan to update it frequently and query infrequently, it might be worth experimenting with not using indexes to compare performance.

Common advice is to wait until we have clear evidence that we need an index, then add it.

15.8 Creating PostgreSQL Indexes

All of the fields we might consider adding indexes to at this time already have indexes due to **`UNIQUE`** or **`PRIMARY KEY`** constraints, but if we wanted to add one with PostgreSQL we would use the **`CREATE INDEX`** command.

```
CREATE INDEX sessions_token_hash_idx ON sessions(token_hash);
```

In some SQL variants you can create an index while creating a table, but this does NOT work with Postgres.

```
CREATE TABLE dogs (
  id SERIAL PRIMARY KEY,
  name TEXT,
  user_id INT REFERENCES users (id) ON DELETE CASCADE,
  -- Does NOT work with PostgreSQL
  INDEX dogs_user_id_idx (user_id)
);
```

Instead, we would need to create the table and then create the index.

```
CREATE TABLE dogs (
  id SERIAL PRIMARY KEY,
  name TEXT,
  user_id INT REFERENCES users (id) ON DELETE CASCADE
```

```
);

CREATE INDEX dogs_user_id ON dogs(user_id);
```

Indexes can be more complex and can cover multiple fields. This is useful for cases where we are commonly searching via multiple fields.

```
CREATE TABLE dogs (
  id SERIAL PRIMARY KEY,
  name TEXT,
  age INT,
  breed TEXT
);

CREATE INDEX dogs_breed_age_idx ON dogs(breed, age);
```

This might help if we were frequently looking for dogs with a specific breed and age range.

```
SELECT * FROM dogs WHERE breed = 'golden_retriever' AND age > 4;
```

15.9 On Conflict

In **`models/session.go`** we have quite a bit of logic to handle checking if a user has a session.

Alternatively, we can just write a database specific query that works even if a session exists.

For Postgres this requires the use of **`ON CONFLICT`**. For other DBs like MySQL this might be **`ON DUPLICATE KEY`**.

```
INSERT INTO sessions (user_id, token_hash) VALUES (1, 'xyz-456') ON CONFLICT (user_id) DO UPDAT
```

```
func (ss *SessionService) Create(userID int) (*Session, error) {
        bytesPerToken := ss.BytesPerToken
        if bytesPerToken < MinBytesPerToken {
                bytesPerToken = MinBytesPerToken
        }
        token, err := rand.String(bytesPerToken)
        if err != nil {
                return nil, fmt.Errorf("create: %w", err)
        }
        session := Session{
                UserID:    userID,
                Token:     token,
                TokenHash: ss.hash(token),
        }
        row := ss.DB.QueryRow(`
                INSERT INTO sessions (user_id, token_hash)
                VALUES ($1, $2) ON CONFLICT (user_id) DO
                UPDATE
                SET token_hash = $2
                RETURNING id;`, session.UserID, session.TokenHash)
        err = row.Scan(&session.ID)
        if err != nil {
                return nil, fmt.Errorf("create: %w", err)
        }
        return &session, nil
}
```

Source Code

- Source Code - see the final source code for this lesson.

- Diff - see the changes made in this lesson.

15.10 Improved SQL Exercises

15.10.1 Ex1 - Design SQL database tables and relationships for an app

Think about an application or website you frequently use. Now think about what we have learned about SQL databases, relationships, and data types. Try to design the database tables and relationships that you might need to replicate the functionality of that application.

Are there any cases where you aren't sure how the data would be structured? Try looking around online to see if you can get ideas on how to handle that particular case.

15.10.2 Ex2 - What are use cases for each type of JOIN?

We saw several different types of JOINs that can be used when querying a database. Try to think of a few situations where each might be useful.

Chapter 16

Schema Migrations

16.1 What Are Schema Migrations?

We currently have two `.sql` files: - `users.sql` - `sessions.sql`

These files define our database schema - it's structure - but we have to manage the DB schema manually. - Run SQL via `psql` - Drop tables and recreate if they change

This can be problematic: - Devs don't know if a change was made - We can't drop tables in production

Once we define a table and set it up in production, we cannot drop it without losing real user data. We need to alter tables without recreating the table from scratch. This means we may end up with quite a few SQL commands to alter our existing tables.

Schema migrations are used to manage database schemas. Various tools exist, but most work roughly the same way…

Each migration is an incremental step that makes a change to our DB. Eg: "Create a new table" or "Add a column to an existing table"

Migrations are applied in a specific order to get the DB schema into the right state.

Eg our current steps we manually take are: 1. Create the users table 2. Create the sessions table

We could turn each of these steps into a migration, then use schema migration tooling to manage those steps.

We could also add a new step later. Eg: Add the username field to the users table.

This step might have the following SQL associated with it:

```
-- Add username field to the users table
ALTER TABLE users ADD username TEXT;
```

Developers could then use our schema migration tooling to update their DB with these changes.

Schema migrations work best with a way to undo every migration. Eg in our new step to add the `username` column, our undo step might be to remove the column.

```
ALTER TABLE users DROP COLUMN username;
```

This gives us the ability to undo changes if we realize they were invalid. It also makes it easier to roll back a release to production if it causes issues.

The goal of this section is to update our application to use the `pressly/goose` schema migration tool. While we do this, we will also be learning: - How migration tools work behind the scenes - How to set up migrations for our app - How the `goose` migration tool works - How to add tools to our `go.mod` file - And more

By the end of the section we will have a reliable way of managing our DB schema. This will make it easier to: - collaborate with other developers - push DB changes to production - make changes to our DB tables without manually using `psql`

16.2 How Schema Migration Tools Work

Migration tools pretty much all require a list of migration steps. These are most commonly a series of files, and the steps are run using alphabetical ordering of the filenames, which is the order most computers show the files in by default. Below is an example listing of files.

```
001-create_users.sql
002-create_sessions.sql
...
```

Migration tools will have a version associated with each migration file. This is usually a number at the front of the filename, and in many cases that number will be a timestamp.

In the example above, our version numbers are `001`, and `002`.

16.2.1 Tracking versions

The version is used to track which migrations have been run and which have not been run. The rest of the filename is meant to help us as developers understand what each migration does.

One way migration tools track which migrations have been run is to create a table in the database, and insert an entry for each migration version that has been applied.

eg:

```
CREATE TABLE migrations (
  id SERIAL PRIMARY KEY,
  version TEXT UNIQUE
);
```

Whenever the tool runs a migration, it will add an entry to the table to track that is has been run.

Running `001-create_users.sql`:

```
INSERT INTO migrations (version) VALUES ('001');
```

This allows the migration tool to track which migrations have been applied on any DB and which need to be applied.

Not all tools track each individual migration. Some tools only track the most recent migration (eg `002`) that was run and assume any migration with a lower version number has been run.

16.2.2 Rolling back

If we need to undo a migration, the tool will run our undo steps and then remove the entry from the `migrations` table.

Some tools break the migration and the undo step into two files. Eg:

```
001-create_users.up.sql
001-create_users.down.sql
```

The `up` version is the migration (updating the DB) The `down` version is the rollback (downgrading the DB)

Other tools will put both of these in a single **sql** file and use some type of special syntax to separate them:

```
-- +goose Up
CREATE TABLE users (
  ...
);

-- +goose Down
DROP TABLE users;
```

Goose does the latter.

It is worth repeating that these details can change slightly from tool to tool, but the basic idea is generally the same.

16.3 Installing pressly/goose

A few migration tools exist in Go. Common ones: - pressly/goose - golang-migrate/migrate

We are going to use **pressly/goose**.

```
go install github.com/pressly/goose/v3/cmd/goose@v3
```

Goose can do a few things for us: 1. Create template migration files 2. Run migrations 3. Roll back migrations 4. Check the status of migrations 5. Embed migrations into our Go binary

Let's try a few of these out quickly.

```
mkdir migrations
cd migrations
goose create widgets sql
# Open the newly created file
code *_widgets.sql
```

Inside this SQL file there are some comments that goose uses to determine which parts are the migration (called **Up** in goose), and which parts are for rolling a migration (**Down** in goose).

The **StatementBegin** comments also help in more advanced queries. We don't need to worry about them aside form writing our SQL inside of the **Begin** and **End** section.

Update the file with the following SQL:

```
-- +goose Up
-- +goose StatementBegin
CREATE TABLE widgets (
  id SERIAL PRIMARY KEY,
  color TEXT
);
-- +goose StatementEnd

-- +goose Down
-- +goose StatementBegin
DROP TABLE widgets;
-- +goose StatementEnd
```

Now we can run goose to see the status of our database, but we need to tell it how to connect to our database. By default we are using the following connection string:

```
host=localhost port=5432 user=baloo password=junglebook dbname=lenslocked sslmode=disable
```

This can be found by going to **main.go** and updating it to print out the current postgres config:

```
func main() {
  // ...

  // Find this line
  cfg := models.DefaultPostgresConfig()
  // Add this line
        fmt.Println(cfg.String())

  //... unchanged from here onwards
}
```

That means we need to run the following with goose to see the status:

```
# The \ is used to break a command into several lines in bash
goose postgres \
  "host=localhost port=5432 user=baloo password=junglebook dbname=lenslocked sslmode=disable" \
  status
```

This should give us output similar to the output below.

```
Applied At          Migration
=======================================
Pending           -- 20221102180848_widgets.sql
```

For the rest of these commands keep the connection string and change the sub-command we are using with goose.

We can apply any migrations that haven't run yet:

```
goose postgres \
  "host=localhost port=5432 user=baloo password=junglebook dbname=lenslocked sslmode=disable" \
  up
```

We can also rollback migrations one at a time:

```
goose postgres \
  "host=localhost port=5432 user=baloo password=junglebook dbname=lenslocked sslmode=disable" \
  down
```

Now that we have seen some of what goose can do, let's try to rework our application to use it for managing our database.

Source Code

- Source Code - see the final source code for this lesson.
- Diff - see the changes made in this lesson.

16.4 Converting to Schema Migrations

16.5 Prepwork

Step 1: Delete demo widgets migration.

```
rm migrations/*.sql
```

Step 2: reset docker containers

```
docker compose down
docker compose up -d
```

16.6 Convert

Next we need to create the migration files.

```
# Navigate to the migrations directory if not there.
# Alternatively, the -dir flag could be used with goose.
cd migrations

# Create the migration file
goose create users sql

# Open it to add some SQL
code *_users.sql
```

Next, copy the code from the **users.sql** into the **Up** section.

```
-- +goose Up
-- +goose StatementBegin
CREATE TABLE users (
  id SERIAL PRIMARY KEY,
  email TEXT UNIQUE NOT NULL,
  password_hash TEXT NOT NULL
);
-- +goose StatementEnd
```

Next we need to write the SQL for rolling back the migration. In this case we are going to drop both tables.

```
-- +goose Down
-- +goose StatementBegin
DROP TABLE users;
-- +goose StatementEnd
```

Repeat with the **sessions.sql** file:

```
goose create sessions sql
code *_sessions.sql
```

```
-- +goose Up
-- +goose StatementBegin
CREATE TABLE sessions (
  id SERIAL PRIMARY KEY,
  user_id INT UNIQUE REFERENCES users (id) ON DELETE CASCADE,
  token_hash TEXT UNIQUE NOT NULL
);
-- +goose StatementEnd

-- +goose Down
-- +goose StatementBegin
DROP TABLE sessions;
-- +goose StatementEnd
```

Make sure the files are saved and run the migrations.

```
goose postgres \
  "host=localhost port=5432 user=baloo password=junglebook dbname=lenslocked sslmode=disable" \
  up
```

Start our application.

```
go run main.go
```

Create a user or two and make sure everything is working.

Source Code

- Source Code - see the final source code for this lesson.
- Diff - see the changes made in this lesson.

16.7 Schema Versioning Problem

By default, goose creates migrations with a timestamp as the version. This gives us a repeatable order to run and test migrations in. It also ensures new

migrations come after existing ones.

This tends to work well, but it can lead to the versioning problem. Let's look at an example so you understand what it is.

- On Jan 11, Jon starts work on a new feature and creates a migration **`2022-01-11_jon.sql`**
- On Jan 12, Bob starts work on a new feature and creates a migration **`2022-01-12_bob.sql`**
- On Jan 13, Bob finishes and his migration is merged and deployed.
- Now the DB thinks the schema version is **`2022-01-12`**

```
2022-01-08_some_migration.sql
2022-01-12_bob.sql <-- current schema version
```

- On Jan 14, Jon finishes, but his migration is BEHIND the current version of the DB schema.

```
2022-01-08_some_migration.sql
2022-01-11_jon.sql
2022-01-12_bob.sql <-- current schema version
```

Some migration tools will still run Jon's migration, but it will be run in a different order than our filenames suggest.

Other migration tools may ignore Jon's migration altogether, assuming it has been applied since it has an older version.

Even if the migration is run, this presents an issue where the order of migrations we run in development might differ from production.

If we reset our DB and run the migrations, Jon's migration will be run BEFORE Bob's, but that isn't what will happen in production.

We want to avoid this, so we need a solution.

There are a few ways to approach this.

The first is to just watch timestamps and make sure any merges have timestamps AFTER the last timestamp already merged. Eg Jon would have needed to rename his migration to something like `2022-01-14_jon.sql` giving us a new ordering:

```
2022-01-08_some_migration.sql
2022-01-12_bob.sql <-- current schema version
2022-01-14_jon.sql
```

Another similar option is to rename migrations just before merging to have a more obvious ordering. Eg prefixes like `0001`, `0002`, `0003`, ...

Even if Jon didn't know about Bob's migration, when he renames it he will have a list of migrations like below.

```
0001_some_migration.sql
2022-01-11_jon.sql
```

The timestamp will become the next number in the order, or `0002`.

```
0001_some_migration.sql
0002_jon.sql
```

Then Jon will try to merge and see the following files are about to be present in the main repository:

```
0001_some_migration.sql
0002_bob.sql
0002_jon.sql
```

Having two migrations with the `0002` prefix makes it a bit more obvious that we have an issue that needs addressed.

When Jon sees the two `0002` migrations, he will know that he needs to update his version number before merging.

The fix would be to move Jon's migration to the next version in the sequential list (`0003`), test that the migration works with the changes made by others, and merge the code.

Goose provides some tooling to help with this.

```
goose fix
```

This will rename all of our files with timestamps to fixed version numbers like `00001`, `00002`, and so on.

There are some details to keep in mind: - If a migration with a timestamp as the version has been run, it will need rolled back with `goose down` before running `goose fix`.

Aside 16.1. My Workflow with Goose

My typical workflow is to keep the timestamps while developing a feature, then when I am ready to submit changes I will:

1. `goose down` or `goose reset`

2. **`git stash`** to stash my current changes
3. **`git pull origin <main branch> -rebase`** to rebase my branch with any changes other devs have submitted.
4. **`git stash apply`** to apply the changes i stashed
5. **`goose fix`** to rename my migrations with new versions
6. **`goose up`** and verify migrations work with new versions
7. commit to main branch

Reviews may slow

Doing this will help ensure that you wait until the last possible minute to set a fixed version number and reduce the odds of conflicts.

Goose suggests a slightly different approach of hybrid versioning that can also be explored.

Let's fix the ordering for our migrations.

```
cd migrations
goose postgres \
  "host=localhost port=5432 user=baloo password=junglebook dbname=lenslocked sslmode=disable" \
  reset

goose fix

goose postgres \
  "host=localhost port=5432 user=baloo password=junglebook dbname=lenslocked sslmode=disable" \
  up

# Navigate back to the main directory
cd ..
```

Source Code

- Source Code - see the final source code for this lesson.
- Diff - see the changes made in this lesson.

16.8 Running Goose with Go

There are a number of reasons why we might want to run migrations via our Go code.

1. We might want to run migrations when the app starts, ensuring production is always on the most migration.
2. We might want to create a subcommand to run our migrations.
3. We may want to use migrations written in Go, which goose only supports if we build our own binary.

First we have to `go get` the library, since this is different from the `go install` we did earlier.

```
go get github.com/pressly/goose/v3
go mod tidy
```

Aside 16.2. Why do we need to use "go mod tidy"?

In a nutshell, running `go mody tidy` will tidy up our `go.mod` file. It does this by looking at our source code and the dependencies we have listed as imports. As a result, tidy can both add and remove entries from our `go.mod` file.

While coding the application for the course, I noticed I was getting an error when I tried to build the application after using `go get` to retreive the `pressly/goose` module. A quick fix to this issue was to run `go mod tidy`, which ensures the modules are all accurate.

Next, open the `postgres.go` source file inside the `models` package where we will add the code for migrating our database.

```
code models/postgres.go
```

Add the following below the `Open` function:

```
func Migrate(db *sql.DB, dir string) error {
        err := goose.SetDialect("postgres")
        if err != nil {
                return fmt.Errorf("migrate: %w", err)
        }
        err = goose.Up(db, dir)
        if err != nil {
                return fmt.Errorf("migrate: %w", err)
        }
        return nil
}
```

At this point we might find our code giving us an error about `goose`. This is because `go mod tidy` likely removed it from our `go.mod` file. Head to our imports in the `postgres.go` file and make sure we have an import for goose v3:

```
import (
        "database/sql"
        "fmt"

        _ "github.com/jackc/pgx/v4/stdlib"
        "github.com/pressly/goose/v3"
)
```

After we know the import is there, run tidy again and it will fix the `go.mod` file:

```
go mod tidy
```

From this point onwards we should get the correct version of goose when we let our editor's autocomplete import goose.

Now back to our code. In this code we first tell `goose` that we plan to use the Postgres dialect via `SetDialect`. After that we proceed to call the `Up` function which will run any missing migrations in the directory we provide using the `db` connection we pass in.

This is roughly equivalent to running the command:

```
goose postgres "connect string..." up
```

Aside 16.3. Goose uses global variables

The `pressly/goose` library uses global variables, which is why we can call a function like `SetDialect` and the changes from that function call are persisted later when we call the `Up` function. I suspect this design choice was made to make using `.go` migration files a bit easier to use (we will see this in the next lesson), but I can't say for certain.

One thing worth noting is that this approach could lead to bugs if other parts of our program relied on the `goose` library and expected some of these global variables to have different settings than we are choosing. Luckily that shouldn't be an issue with our app, but it is something to take note of if working on an app that uses multiple databases.

Let's update our code to use our new `Migrate` function.

Open `main.go`:

```
code main.go
```

Head down to the area where we declare the `sql.DB`, and add the following code right after the `defer db.Close()` line and before we setup our services.

```
// Find this code
cfg := models.DefaultPostgresConfig()
db, err := models.Open(cfg)
if err != nil {
  panic(err)
}
defer db.Close()

// Add the following code
err = models.Migrate(db, "migrations")
if err != nil {
  panic(err)
}
```

Our code now runs our migrations every time it starts up.

Source Code

- Source Code - see the final source code for this lesson.
- Diff - see the changes made in this lesson.

16.9 Embedding Migrations

At the moment our SQL migration files need to be present when we run our migrations via our Go code.

We can embed them into our Go binary, making it much easier to run migrations anywhere without also copying the `.sql` files around.

First we need a function that runs migrations with an embedded file system.

```
code models/postgres.go
```

We will be adding a new function named `MigrateFS`

```
func MigrateFS(db *sql.DB, migrationsFS fs.FS, dir string) error {
  // In case the dir is an empty string, they probably meant the current directory and goose wa
  if dir == "" {
    dir = "."
  }
        goose.SetBaseFS(migrationsFS)
        defer func() {
                // Ensure that we remove the FS on the off chance some other part of our app us
                goose.SetBaseFS(nil)
        }()
        return Migrate(db, dir)
}
```

We are able to use the existing `Migrate` function because goose uses global variables, so when we call `Migrate` it will use the base file system we just set. Then once our `MigrateFS` function is complete it will unset the base file system by passing in `nil` in the deferred code.

Next we need to use the `embed` package to embed our migration files. We will create a file in the `migrations` directory to store this.

```
code migrations/fs.go
```

Add the variable to store the embed.FS:

```
package migrations

import "embed"

//go:embed *.sql
var FS embed.FS
```

Finally, update our code in **main.go** to use the new code.

```
code main.go
```

```
func main() {
        // ...

        // Find this code
        cfg := models.DefaultPostgresConfig()
        db, err := models.Open(cfg)
        if err != nil {
                panic(err)
        }
        defer db.Close()

        // Change the following line of code
        err = models.MigrateFS(db, migrations.FS, ".")
        if err != nil {
                panic(err)
        }
        // ...
}
```

We no longer need the **"migrations"** directory variable because our embedding occurs within the **migrations** directory in **fs.go**. We can instead pass an empty directory string.

Restart your application and verify the code works.

Source Code

- Source Code - see the final source code for this lesson.
- Diff - see the changes made in this lesson.

16.10 Go Migration Files

Note: We won't be using any code from this section. It is only meant to illustrate how Go migration files work with Goose.

Goose also supports migration files written in Go. We can still access our database inside of our Go code, so this gives us the ability to use both Go and SQL at the same time. These can be handy in a variety of cases where that mixture is easier to use than pure SQL.

To create a `.go` migration, use the `goose` cli:

```
cd migrations

goose create widgets go

code *_widgets.go
```

Goose will create a Go source file similar to the one below.

```
package migrations

import (
        "database/sql"
        "github.com/pressly/goose/v3"
)

func init() {
        goose.AddMigration(upWidgets, downWidgets)
}

func upWidgets(tx *sql.Tx) error {
        // This code is executed when the migration is applied.
        return nil
}

func downWidgets(tx *sql.Tx) error {
        // This code is executed when the migration is rolled back.
        return nil
}
```

All we need to do is fill in the `upXxx` and `downXxx` functions with the logic we want to use for the migration and its rollback.

Migrations in Go files can only be run if we use the goose library and create a custom binary. If we attempt to use the `goose` CLI we will get an error with the message:

> Go functions must be registered and built into a custom binary (see https://github.com/pressly/goose/tree/master/examples/go-migrations)

This is because the CLI has no way of building the Go code that we are writing and running it.

In order to use Go migration files, we need to import our migration code and call functions like `goose.Up` in our code, much like we do in the `models/postgres.go` source file.

This works because of the `init` function inside of our generated `.go` file - as long as the `migrations` package is imported, it will register the migrations to `goose`.

This also means that we cannot roll a migration back with the CLI.

If we wanted to use Go migrations, we would either need to create a few custom commands that we can run, or we would need to look at wrapping the Goose CLI.

There is an example of the latter in the Goose docs, but we won't be covering that now since we won't be using Go migration files at this time.

Delete any files created from this lesson before proceeding, as we won't be using them.

16.11 Removing Old SQL Files

We no longer need the `models/sql/*.sql` files, so we can delete those:

```
rm -rf models/sql/
```

Source Code

- Source Code - see the final source code for this lesson.
- Diff - see the changes made in this lesson.

Chapter 17

Current User via Context

17.1 Using Context to Store Values

In the next few sections we are going to focus on completing our authentication system.

Most notably, we need a way to: 1. Access the user in various handlers so we can restrict access as needed. 2. Allow users to reset their password via a password reset form.

In this section we are going to focusing the first part - making it easier to work with the user in our http handlers, and even in our templates in order to render dynamic links (showing sign in vs sign out links).

Currently, we look up who a user is inside of each HTTP handler function by looking up their session cookie, then using a session service to determine if they have a valid session. We can see this code in the `Users.CurrentUser` function inside our controllers package.

```
func (u Users) CurrentUser(w http.ResponseWriter, r *http.Request) {
        token, err := readCookie(r, CookieSession)
        if err != nil {
```

```
                fmt.Println(err)
                http.Redirect(w, r, "/signin", http.StatusFound)
                return
        }
        user, err := u.SessionService.User(token)
        if err != nil {
                fmt.Println(err)
                http.Redirect(w, r, "/signin", http.StatusFound)
                return
        }
        fmt.Fprintf(w, "Current user: %s\n", user.Email)
}
```

We don't want to write this code in every handler that requires a user to be signed in. Instead, it would be nice to simplify this so our http handlers are easier to write. We also want to ensure that whatever process we use helps us avoid querying for the current users multiple times.

The most common way to achieve this is via Go's context.Context type.

Context can be used for a variety of things including setting timeouts and cancelling processes. Most of these use cases are more advanced, so we won't be diving into them. Instead, we are going to look at another way to use the `Context` type - storing request-scoped values.

A request-scoped value is a value that is specific to the specific query our code is responding to. For example, every web request we receive may be from a different user, so the current user is specific for each incoming web request. If our code used the IP address of a request for some reason, this might also make sense to store on in the context, as it is specific to that web request.

Something like a logger or a database connection typically isn't request-scoped. That is, these things tend to be the same regardless of who made the web request, so storing them in a context isn't a good idea.

We will elaborate more on this in a later lesson. For now, the key thing to know is that we will working towards writing code that looks up who a user is once, stores it in the context, then the rest of our app can use the user we looked up as needed.

Let's look at how to use the context package to store and retrieve values so we can slowly work towards our goal.

Contexts can store context-specific values using the WithValue function.

Values can be retrieved from a context with the **`Value`** method that is defined in the context.Context interface.

Let's look at an example of this. The entire example is available on the Go Playground, or you can code along in the **`exp.go`** source file we have for experimenting.

```
code cmd/exp/exp.go
```

To store a value we need a context. We can create an initial context with the Background function.

```
ctx := context.Background()
```

Next, we use the **`WithValue`** function to store a value. This requires an existing context to add the value to, a key used to store and lookup the value, and the value itself. It is worth noting that behind the scenes, functions like **`WithValue`** may return a new context that wraps the existing one (much like we can wrap errors), so we need to assign the result to a variable.

```
ctx = context.WithValue(ctx, "favorite-color", "blue")
```

Finally, we can retrieve a value from a context using the **`Value`** method. While this will give us back what we stored in the context, the **`Value`** method has a return type of any which means it can be any data type. *Sidenote: **`any`** is the same as **`interface{}`** if you happen to see that format, and in both cases it means we know nothing about the type of the data.*

```
value := ctx.Value("favorite-color")
```

The complete, runnable program is shown below.

```
package main

import (
        "context"
        "fmt"
)

func main() {
        ctx := context.Background()
        ctx = context.WithValue(ctx, "favorite-color", "blue")
        value := ctx.Value("favorite-color")

        fmt.Println(value)
}
```

While this code does show the absolute basics of using the context package, this isn't the best way to store and retrieve objects as we will see later in this section. For now the key takeaway is that we can store and retrieve values in a context, which means we can use this to store the current user and as long as the rest fo our code has access to the context it will be able to see who the current user is without looking them up again in the database.

Source Code

- Source Code - see the final source code for this lesson.
- Diff - see the changes made in this lesson.

17.2 Improved Context Keys

In our first example of using context values, we used a string as the key. Keys for context values can be of any type - int, string, or even a custom type.

Below is an example showing a custom type being used as the key.

```
type ctxKey string

const (
  favoriteColorKey ctxKey = "favorite-color"
)

func main() {
  ctx := context.Background()
  ctx = context.WithValue(ctx, favoriteColorKey, "blue")
  fmt.Println(ctx.Value(favoriteColorKey))
}
```

While any type can be used, general advice is to not use common types like string or int because these keys can be overwritten by any code, even code outside of your current package.

For example, we might set our "favorite-color" to blue, but some code later on uses that same key, our value can be overwritten.

```
// We set the color to blue
ctx = context.WithValue(ctx, "favorite-color", "blue")

// Code in another package could potentially overwrite our value if it happens
// to use the same key value.
ctx = context.WithValue(ctx, "favorite-color", "red")

// Oh no, this returns "red"!
value := ctx.Value("favorite-color")
```

In reality this doesn't happen frequently with a key like "favorite-color", but it can happen if we are using more common key names.

A common best-practice to avoid this is to use an unexported type for our context keys. By doing this, we avoid situations like the one we saw with the favorite color. Below is an example.

```
type ctxKey string

const (
        favoriteColorKey ctxKey = "favorite-color"
)

func main() {
        ctx := context.Background()
        // Our code uses our unexported `ctxKey` type. Even though the value still
  // appears to be a string with the contents "favorite-color", Go and the
  // context package treat this different from a string with the value
        // "favorite-color"
        ctx = context.WithValue(ctx, favoriteColorKey, "blue")

        // This key has a type of string, not ctxKey.
        ctx = context.WithValue(ctx, "favorite-color", "red")

        // Each key has a unique type, so the keys won't match and we will get
  // unique values for each key.
        value1 := ctx.Value(favoriteColorKey)
        value2 := ctx.Value("favorite-color")
        fmt.Println(value1)
        fmt.Println(value2)
}
```

Despite appearing to have the same key, the two can coexist without any issues.

Put more straightforward, keys in context values are a tuple consisting of both the value, and the type of the key.

By not exporting the key or the type used to create it from a package, we ensure that only code inside of that package can overwrite or access our context values.

Aside 17.1. Confused right now?

I am including this information because it is important in the long term, but if you don't fully understand how or why this unexported type works, don't worry about it. I do suggest that you follow along and use the same approach to avoid potential issues in your application, but you can very likely proceed and build your application without entirely understanding why or how the key type works.

In a later lesson we will utilize what we learned here as we write code to store and retrieve context values in our application.

Source Code

- Source Code - see the final source code for this lesson.
- Diff - see the changes made in this lesson.

17.3 Context Values with Types

When we retrieved a value from the context it had the type `any`. This is because any type can be stored in a context, so the context has to return data to us in that format.

Aside 17.2. Generics may affect this

It might not be entirely true that the context package HAS to return values with the type `any` now that Go has generics, but the context package hasn't been updated to account for this, so for now we assume it is true. If the context package is ever updated to include generics support, I'll update this lesson accordingly.

We want to store a `models.User` in our context, and ideally it will have that type when we retrieve it. Otherwise it isn't so useful, because we can't access any fields until we get it into the correct type.

Let's look at how we can convert a value back into it's proper type.

We do this using a type assertion in Go.

```
ctx := context.Background()
ctx = context.WithValue(ctx, favoriteColorKey, "blue")
anyValue := ctx.Value(favoriteColorKey)

// This .(string) format attempts to assert that anyValue has a type of string
// If it succeeds, ok will be true. Otherwise ok will be false.
stringValue, ok := anyValue.(string)
if !ok {
  // anyValue is not a string!
  fmt.Println(anyValue, " is not a string")
  return
}
// if we are here, stringValue has a type of string!
fmt.Println(stringValue, " is a string!")
```

Type assertions can be done without the second argument, but if the assertion fails it will result in a runtime error.

```
// If this fails, our code will panic with a runtime error
stringValue := anyValue.(string)
```

As long as we know the type that was stored in a context value, we can use this to convert it back into the correct type.

Source Code

- Source Code - see the final source code for this lesson.
- Diff - see the changes made in this lesson.

17.4 Storing Users as Context Values

Let's put this all together to make a working package that allows us to store and retrieve users from a context.

Create a new package named `context`, and a file in it named `users.go`

```
mkdir context
code context/users.go
```

The code:

```
package context

import (
  "context"
  ...
)

type key string

const (
  userKey key = "user"
)

func WithUser(ctx context.Context, user *models.User) context.Context {
  return context.WithValue(ctx, userKey, user)
}

func User(ctx context.Context) *models.User {
  val := ctx.Value(userKey)
  user, ok := val.(*models.User)
  if !ok {
    // The most likely case is that nothing was ever stored in the context,
    // so it doesn't have a type of *models.User. It is also possible that
    // other code in this package wrote an invalid value using the user key.
    return nil
  }
  return user
}
```

In exp.go:

```
package main

import (
  // TODO: Explain the stdctx rename here since we have two
  // context packages being imported
        stdctx "context"
        "fmt"
```

```
        "github.com/joncalhoun/lenslocked/context"
        "github.com/joncalhoun/lenslocked/models"
)

func main() {
        ctx := stdctx.Background()

        user := models.User{
                Email: "jon@calhoun.io",
        }
        ctx = context.WithUser(ctx, &user)

        retrievedUser := context.User(ctx)
        fmt.Println(retrievedUser.Email)
}
```

Source Code

- Source Code - see the final source code for this lesson.
- Diff - see the changes made in this lesson.

17.5 Reading Request Context Values

We have seen how to store and retrieve values from a context. We also wrote code to do this in a safer way, with both type safety and unexported keys. Now we need to determine what context we are going to use, and how we are going to access it inside of our HTTP handlers.

If we look at our existing handlers, there is no mention of a context being passed into the function.

```
// controllers/users.go
func (u Users) CurrentUser(w http.ResponseWriter, r *http.Request) {
        // ...
}
```

We also can't add it as a field to the `Users` type, as this type is created once when we start our application, while the context needs to be specific for each incoming web request.

Luckily, every `http.Request` has a context already associated with it.

We get access to the context by calling the Request.Context method.

We can also update the context on an request using the Request.WithContext function.

Using these two functions, we are going to update our code in two ways:

1. We are going to add middleware that looks up the current user and stores it in the request context.
2. We are going to update our http handler functions to only read the current user from the context, rather than trying to query the database for the user.

Doing this will avoid unnecessary duplicate queries to the database for a single user, and will prove to simplify the code in our controllers.

In this lesson we will focus updating the `CurrentUser` handler on the users controller to read from the context. In the next lesson we will write our middleware and set it up. Until we complete both steps, the code we write in this lesson won't work as expected, so keep that in mind.

Open up the users controller.

```
code controllers/users.go
```

Head to the `CurrentUser` function where we currently use the cookie and the session service to look up the current users.

Let's assume that we already had code in place to store the user in the request context and update this function to utilize that.

```
func (u Users) CurrentUser(w http.ResponseWriter, r *http.Request) {
        user := context.User(r.Context())
  if user == nil {
                http.Redirect(w, r, "/signin", http.StatusFound)
                return
        }
        fmt.Fprintf(w, "Current user: %s\n", user.Email)
}
```

This code is much simpler! It won't work just yet, because we never set a user in our context, but once we do that it be much easier to access the current user in our handlers.

Source Code

- Source Code - see the final source code for this lesson.
- Diff - see the changes made in this lesson.

17.6 Set the User via Middleware

Now that we are reading from the request context, we need to think about how we are going to set a user in the `http.Request` context. We can achieve this by writing middleware that runs both before and after our handlers run. We saw this earlier when we integrated CSRF protection and learned about middleware.

In our case, the middleware is going to: 1. Lookup the session token via the users cookies 2. Query for a valid session with the session service 3. Store the user associated with the session in the context 4. Continue with the HTTP handler that it wraps.

If at any point we cannot continue looking up the user - like if a session cookie isn't present - we will continue to the next handler without setting a user.

The code in our handler will look similar to the code that used to be in the `CurrentUser` handler, but not exactly the same. This is because the goal of this middleware isn't to require that a user be present; it is to look up a user if one can be found and to store it in the request context. Later we can write middleware that redirects incoming web requests if a user is not present, and we can apply this to specific endpoints that require a user to be logged in.

We could make an argument for this code being in either the `controllers` package, or in a new `middleware` package. In our particular app, our `controllers` package contains pretty much all code related http. It does NOT have HTML rendering - that is for the views package. It also doesn't interact with our data and datbase - that is the models packages. But anything related to cookies, http handlers, etc is in the controllers package. By that logic, middleware fits nicely.

It is worth noting that this doesn't have to be the case. If we wanted, we could: - Move the cookie-related logic into a `cookie` package. - Move the http handlers (eg `Users.CurrentUser`) to a `handler` package. - Create middleware in a `middleware` package.

While we are learning one way to structure our code, there is more than one valid way and it can depend on how much code you have. Our app isn't too big right now, so utilizing the single `controllers` package makes sense for now versus having a bunch of packages with one or two files.

Open the `users.go` file in the `controllers` package. This file has all of our user-related code, so we will add the user middleware to the same source file. Once again - you could break the HTTP handlers and the middleware up into two files if you wanted.

```
code controllers/users.go
```

In the new file, add a `UserMiddleware` type. We know from the old `CurrentUser` code that we will need a SessionService field as well so that we can query for a user via their session token.

```
type UserMiddleware struct {
        SessionService *models.SessionService
}
```

Next, we want to add a method that can be used to apply our middleware. That is, we need a method that accepts an http handler as an argument, and returns a new http handler.

The definition of our function will be:

```
func(http.Handler) http.Handler
```

We opt to use this format and return an **http.Handler** rather than an **http.HandlerFunc** because this is the pattern used elsewhere in our code right now. - The **chi** package uses it for middleware. - Our CSRF middleware uses this format.

It is a good idea to try to keep things consistent when we can.

Putting this all together, we can start our function with the following code.

```
func (umw UserMiddleware) SetUser(next http.Handler) http.Handler {
        return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
                // TODO: Add logic for the SetUser middleware, then eventually call next.ServeH
        }
}
```

We end up needing to use the **http.HandlerFunc** here because we want to return a function, but we need something that implements the **http.Handler** interface. Converting our function to an **http.HandlerFunc** causes it to implement the **http.Handler** interface. We learned about this earlier in the course in Section 2 if you want to go back and recap a bit.

Now we can fill in the logic

```
func (umw UserMiddleware) SetUser(next http.Handler) http.HandlerFunc {
        return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
                // First try to read the cookie. If we run into an error reading it,
                // proceed with the request. The goal of this middleware isn't to limit
                // access. It only sets the user in the context if it can.
                token, err := readCookie(r, CookieSession)
                if err != nil {
                        // Cannot lookup the user with no cookie, so proceed without a user bei
                        // set, then return.
                        next.ServeHTTP(w, r)
                        return
                }

                // If we have a token, try to lookup the user with that token.
                user, err := umw.SessionService.User(token)
                if err != nil {
                        // Invalid or expired token. In either case we can still proceed, we ju
                        // cannot set a user.
                        next.ServeHTTP(w, r)
                        return
                }

                // If we get to this point, we have a user that we can store in the context!
                // Get the context
                ctx := r.Context()
                // We need to derive a new context to store values in it. Be certain that
    // we import our own context package, and not the one from the standard
    // library.
                ctx = context.WithUser(ctx, user)
                // Next we need to get a request that uses our new context. This is done
    // in a way similar to how contexts work - we call a WithContext function
    // and it returns us a new request with the context set.
                r = r.WithContext(ctx)
                // Finally we call the handler that our middleware was applied to with the
    // updated request.
                next.ServeHTTP(w, r)
        })
}
```

Finally, we need to update our web server to use this middleware.

Open **`main.go`**.

```
code main.go
```

Near the bottom of the **`main()`** function, find the **`fmt.Println`** line and move

it just before the **`http.ListenAndServe`** function call. This is just cleanup - I like that println to occur just before we start the server.

```
func main() {
  r.NotFound(func(w http.ResponseWriter, r *http.Request) {
                http.Error(w, "Page not found", http.StatusNotFound)
        })
  // Find this line and move it down below.
        // fmt.Println("Starting the server on :3000...")

        csrfKey := "gFvi45R4fy5xNBlnEeZtQbfAVCYEIAUX"
        csrfMw := csrf.Protect(
                []byte(csrfKey),
                // TODO: Fix this before deploying
                csrf.Secure(false),
        )

        fmt.Println("Starting the server on :3000...")
        http.ListenAndServe(":3000", csrfMw(r))
}
```

Next, instantiate the middleware.

```
func main() {
  // ...
  r.NotFound(func(w http.ResponseWriter, r *http.Request) {
                http.Error(w, "Page not found", http.StatusNotFound)
        })

  umw := controllers.UserMiddleware{
    SessionService: &sessionService,
  }

  csrfKey := "gFvi45R4fy5xNBlnEeZtQbfAVCYEIAUX"
  csrfMw := csrf.Protect(
    []byte(csrfKey),
    // TODO: Fix this before deploying
    csrf.Secure(false),
  )
  handler = csrfMw(handler)

  fmt.Println("Starting the server on :3000...")
  http.ListenAndServe(":3000", csrfMw(r))
}
```

Finally, update the final line of code so that our middleware is applied to our router.

```
func main() {
  // ...

  http.ListenAndServe(":3000", csrfMw(umw.SetUser(r)))
}
```

In this code we are wrapping the router with the middleware that sets a user, then we wrap that whole http handler with the CSRF middleware. This means CSRF protection will run first, then our user lookup, and finally our router will decide what HTTP handler to use based on the path and HTTP method. The order of these can be important.

Long term we might look at other ways to apply middleware rather than chaining these functions, but for now this will work fine.

Let's verify that this is working.

1. Restart the server.
2. Sign in to a valid account.
3. Verify that the `/users/me` path works with the account you are signed into.

We now have an easy way to look up the current user in any of our handlers!

Source Code

- Source Code - see the final source code for this lesson.
- Diff - see the changes made in this lesson.

17.7 Requiring a User via Middleware

Some of our endpoints are going to require that a user is logged in to access them. - When seeing a list of our galleries, we need to be logged in so the server knows which galleries are ours. - When editing a gallery we need to be logged in so the server can determine if we have permission to edit the gallery.

We are going to create middleware that requires a user to be signed in, and otherwise redirects them to the sign in page. We will be adding this to the **`UserMiddleware`** in the controllers package.

```
code controllers/users.go
```

Add the following method to the user middleware. We are going to assume that the SetUser middleware has been run BEFORE our middleware, as we already run that middleware on all of our incoming requests.

```
func (umw UserMiddleware) RequireUser(next http.Handler) http.Handler {
        return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
                user := context.User(r.Context())
                if user == nil {
                        http.Redirect(w, r, "/signin", http.StatusFound)
                        return
                }
                next.ServeHTTP(w, r)
        })
}
```

Next we need to apply it to our handlers, which are setup in **`main.go`**.

```
code main.go
```

At this point we are going to rearrange our code a bit. Currently, we declare middleware at the end of the **`main()`** function. We also declare a few routes,

then setup our services and controllers, then declare more routes. We are going to place every step together so that we: 1. Setup a DB connection and handle migrations 2. Setup our services 3. Setup our middleware 4. Setup our controllers 5. Setup our router and routes

This will make it a bit easier to find what we need in the future. The code is shown below - remember that this is the same code we had, just rearranged a bit.

```
// TODO:...
```

Now that our main function is more organized, we are going to utilize the **`Router.Use`** method provided by Chi for applying middleware.

```
// Setup our router
r := chi.NewRouter()
// These middleware are used everywhere.
r.Use(csrfMw)
r.Use(umw.SetUser)
// Now we setup routes
r.Get("/", controllers.StaticHandler(views.Must(views.ParseFS(
  templates.FS,
  "home.gohtml", "tailwind.gohtml",
))))
// ...
```

Be sure to update the **`ListenAndServe`** line to just use the router. We are now applying middleware using **`r.Use`**, so we don't need to manually wrap the router like we were doing.

```
http.ListenAndServe(":3000", r)
```

Next, we add the new **`RequireUser`** middleware. For now it is only needed on the current user page, but in reality we know that any path with a **`/users/me`** prefix will need a current user. Eg if we had **`GET /users/me`** for showing a

user's info, and **`PUT /users/me`** for updating their info, both of these would require a user. We will use Chi's subrouter functionality to apply the router to all routes that match this prefix.

```
// This is the new code
r.Route("/users/me", func(r chi.Router) {
  r.Use(umw.RequireUser)
  r.Get("/", usersC.CurrentUser)
})
// This is the code we are replacing
// r.Get("/users/me", usersC.CurrentUser)
```

Now any routes defined inside the function we pass to **`r.Route`** will have a **`/users/me`** prefix automatically. This means we can say **`r.Get("/", ...)`** and it will match **`/users/me`** or **`/users/me/`**.

We won't use this code, but to quickly show a few examples:

```
r.Route("/users/me", func(r chi.Router) {
  // will be used on all routes with the /users/me prefix
  r.Use(mw)
  // Will match the path /users/me and
  // the path /users/me/. Trailing slashes
  // shouldn't matter with chi.
  r.Get("/", ...)

  // Will match the path /users/me/hello
  r.Get("/hello", ...)
})
```

As we build our application, we will start to use nested routes to make applying middleware easier to do.

For instance, we could use a prefix like **`/app`** to denote all paths that are used to interact with our web app and require a user to be signed in. Then our routes for the app might be like: - **`GET /app/galleries`** - **`DELETE /app/galleries/123`** - **`POST /app/images`**

Finally, update the **`CurrentUser`** handler to no longer check if a user is present.

```
code controllers/users.go
```

```
func (u Users) CurrentUser(w http.ResponseWriter, r *http.Request) {
        user := context.User(r.Context())
        fmt.Fprintf(w, "Current user: %s\n", user.Email)
}
```

We no longer need to worry about checking if the user is nil because this controller is expected to be behind the `RequireUser` middleware. It might be worth noting that in the documentation.

Source Code

- Source Code - see the final source code for this lesson.
- Diff - see the changes made in this lesson.

17.8 Accessing the Current User in Templates

We currently show a link to sign out regardless of whether a user is signed in. Many apps will show who is currently signed in via an avatar or email.

We are going to look at adding a way to determine if there is a current user, and who they are, to our templates.

We already know we can lookup a current user if we have access to the `http.Request` We also know that we can add template functions that have access to the `http.Request` - we did this with the `csrfField` template function at `views/template.`

```
// Stubbed function for parsing
func ParseFS(fs fs.FS, patterns ...string) (Template, error) {
        tpl := template.New(patterns[0])
```

```
        tpl = tpl.Funcs(
                template.FuncMap{
                        "csrfField": func() (template.HTML, error) {
                                return "", fmt.Errorf("csrfField not implemented")
                        },
                },
        )
  // ...
}

// Actual function implementation with the request
func (t Template) Execute(w http.ResponseWriter, r *http.Request, data interface{}) {
        tpl, err := t.htmlTpl.Clone()
        if err != nil {
                log.Printf("cloning template: %v", err)
                http.Error(w, "There was an error rendering the page.", http.StatusInternalServ
                return
        }
        tpl = tpl.Funcs(
                template.FuncMap{
                        "csrfField": func() template.HTML {
                                return csrf.TemplateField(r)
                        },
                },
        )
  // ...
}
```

Given that we have seen how this approach works, we won't review too much and will dive right into implementing it.

Aside 17.3. Passing the User via Data

When exploring how to provide the CSRF field, we also looked at how this could be passed in as data that we provide to our template. In the previous version of this course I used a custom data type to provide the current user to each template. The user was still determined via the http.Request, so it was otherwise a very similar approach. I'm not using that approach now because I felt it led to mistakes too easily. When writing Go templates, developers could too easily forget to pass the right data to nested templates, and this would lead to errors and bugs. By

providing this data via a template function, we ensure it is available in any template regardless of what data was passed into the template. There are pros and cons to both approaches, but given that we need to clone our template for every request anyway, the cons are pretty minimal. I am noting this for anyone who is wondering why this version of the course took another approach.

We will start by pretending we have a **currentUser** function in our templates We can then update our templates and figure out an ideal way for the function to work. A good way to do this is to write the code you wish you had, then see if it is possible to make that work in reality.

```
code templates/tailwind.gohtml
```

Find the ** ** block that contains the sign out and sign in links and update it with the following code.

```
<div>
  {{if currentUser}}
    <form action="/signout" method="post" class="inline pr-4">
      <div class="hidden">
        {{csrfField}}
      </div>
      <button type="submit">Sign out</button>
    </form>
  {{else}}
    <a class="pr-4" href="/signin">Sign in</a>
    <a class="px-4 py-2 bg-blue-700 hover:bg-blue-600 rounded" href="/signup">Sign up</a>
  {{end}}
</div>
```

With this code we are saying we want to show a sign out link if the current user exists, otherwise we want to show a sign in and sign up link.

Now our templates will start erroring when they compile. We need to add the **currentUser** function to our templates to fix this.

```
code views/template.go
```

Add the **currentUser** function in the ParseFS function so that we can parse templates without an error.

```
func ParseFS(fs fs.FS, patterns ...string) (Template, error) {
        tpl := template.New(patterns[0])
        tpl = tpl.Funcs(
                template.FuncMap{
                        "csrfField": func() (template.HTML, error) {
                                return "", fmt.Errorf("csrfField not implemented")
                        },
                        "currentUser": func() (*models.User, error) {
                                return nil, fmt.Errorf("currentUser not implemented")
                        },
                },
        )
  // ...
}
```

Head down to the **Template.Execute** method where we need to add the real implementation of the **currentUser** template function.

```
func (t Template) Execute(w http.ResponseWriter, r *http.Request, data interface{}) {
        // ...
        tpl = tpl.Funcs(
                template.FuncMap{
                        "csrfField": func() template.HTML {
                                return csrf.TemplateField(r)
                        },
                        "currentUser": func() *models.User {
                                return context.User(r)
                        },
                },
        )
  // ...
}
```

Restart and verify everything works.

Source Code

- Source Code - see the final source code for this lesson.
- Diff - see the changes made in this lesson.

17.9 Request-Scoped Values

The context package doesn't enforce any limitations on how it can be used. The docs state that it: > ... carries deadlines, cancellation signals, and other request-scoped values across API boundaries and between processes. But technically we could store anything we want in a context.

Let's discuss why storing anything that isn't request-scoped in a context is likely a bad idea.

Imagine that we are building an application, and we know that things like our HTTP handlers need access to the database. We might decide to store a DB connection in the context, that way the handler can access it. We might also see that our handlers often need various templates, so we might bundle those and store them in the context.

```
// This might have tempates that we lookup by name, eg "home" and "contact"
var templates map[string]Template
ctx = context.WithTemplates(ctx, templates)

// A DB connection we can use to construct services or just
// query the DB with SQL
var db *sql.DB
ctx = context.WithDB(ctx, db)
```

At first glance this seems pretty handy.

With our current `controllers.Users` type, we have to construct the object, set the fields like the templates and services, and then add methods to that type for HTTP handlers.

```
// Our current code for the users controler
type Users struct {
        Templates struct {
                New    Template
                SignIn Template
        }
        UserService    *models.UserService
        SessionService *models.SessionService
}

func (u Users) Create(w http.ResponseWriter, r *http.Request) {
        // ...
}
```

If we stored all of our templates and a DB connection in the context, we would no longer need types like our **`controllers.Users`**. Instead, we can just write functions for handlers, and they could get the template and DB connection out of the context.

```
// package controllers

func CreateUser(w http.ResponseWriter, r *http.Request) {
        ctx := r.Context()
        // We have to read code to see that our function depends
        // on the templates being available in the context.
        templates := context.Templates(ctx)
        newUserTpl = templates["users/new"]
        user := ...
        // We also may not know about nested dependencies.
        // For instance, the models package might depend on the
        // database connection being stored in the context, but
        // that isn't obvious when reading over the
        // controllers.CreateUser function because it is hidden
        // in a call to models.CreateUser.
        err := models.CreateUser(ctx, &user)
        // ...
}
```

Using the new **`controllers.CreateUser`** function appears easier to use because we no longer have to setup a **`controllers.Users`** type. In reality, it will be much harder to maintain because it masks all of its dependencies. That is, we don't really know what types and values the function depends on by just

looking at the function definition. - Does it need a database connection? - Does it need an API client for Stripe? - Does it need the home page template?

While we could look at the code to try to determine these dependencies, the problem is exacerbated by the fact that `CreateUser` might call other functions with their own dependencies, and if those dependencies are also stored in the context we would never know about them being required.

All of this makes it much harder to test and maintain the code.

When we look at our original code, it is clear that our controller requires both a UserService and a SessionService.

```
type Users struct {
        Templates struct {
                New    Template
                SignIn Template
        }
        UserService    *models.UserService
        SessionService *models.SessionService
}
```

By using context values, we can hide dependencies that our application has and thus make it harder to test and maintain the code.

It is also true that storing request-scoped values can cause some of these same issues. The main difference is that with non-request-scoped values like the `UserService` can fairly easily be constructed when our application starts up and attached to a type like `controllers.Users`. Meanwhile, request-scoped values cannot be determined until we are processing a request, so we would need to construct new types or closures for every request. While possible, context values tend to work better.

In summary - limit context values to request-scoped values, and even then try to limit to situations where it is necessary.