



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)	23

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	43
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	59
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	105
4.9	FAQ Page via Template	108
4.10	Template Exercises	109
4.10.1	Ex1 - Use template variables	110
4.10.2	Ex2 - Experiment with different data types	110
4.10.3	Ex3 - Learn how to use nested data	111
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	217
8.4	Users Controller	220
8.5	Decouple with Interfaces	223
8.6	Parsing the Signup Form	230
8.7	URL Query Parameters	234
8.8	Exercises	239
8.8.1	Ex1 - Experiment With Form Fields	240
8.8.2	Ex2 - Adjust Your Paths	240
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
11.6 Learning bcrypt with a CLI	340
11.7 Hashing Passwords with bcrypt	345
11.8 Comparing a Password with a bcrypt Hash	350
12 Adding Users to our App	353
12.1 Defining the User Model	353
12.2 Creating the UserService	357
12.3 Create User Method	360
12.4 Postgres Config for the Models Package	368
12.5 UserService in the Users Controller	372
12.6 Create Users on Signup	374
12.7 Sign In View	377
12.8 Authenticate Users	382
12.9 Process Sign In Attempts	384
13 Remembering Users with Cookies	387

13.1 Stateless Servers	387
13.2 Creating Cookies	391
13.3 Viewing Cookies with Chrome	395
13.4 Viewing Cookies with Go	402
13.5 Securing Cookies from XSS	404
13.6 Cookie Theft	406
13.7 CSRF Attacks	408
13.8 CSRF Middleware	412
13.8.1 What is Middleware?	414
13.8.2 Coding the CSRF Middleware	419
13.8.3 Seeing the Changes	421
13.9 Providing CSRF to Templates via Data	422
13.10 Custom Template Functions	424
13.11 Adding the HTTP Request to Execute	429
13.12 Request Specific CSRF Template Function	434
13.13 Template Function Errors	438
13.14 Securing Cookies from Tampering	442
13.14.1 Digitally Signing Data	443
13.14.2 Obfuscation	446
13.15 Cookie Exercises	448
13.15.1 Ex1 - Set and View Cookies in your Browser	448

13.15.2 Ex2 - Explore the Cookie Docs	448
13.15.3 Ex3 - Redirect When a Cookie is Not Found	448
13.15.4 Ex4 - Attempt to Create Middleware	448
14 Sessions	451
14.1 Random Strings with crypto/rand	451
14.2 Exploring math/rand	457
14.3 Wrapping the crypto/rand Package	460
14.4 Why Do We Use 32 Bytes for Session Tokens?	462
14.5 Defining the Sessions Table	465
14.6 Stubbing the SessionService	469
14.7 Sessions in the Users Controller	474
14.8 Cookie Helper Functions	480
14.9 Create Session Tokens	484
14.10 Refactor the rand Package	487
14.11 Hash Session Tokens	492
14.12 Insert Sessions into the Database	497
14.13 Updating Existing Sessions	500
14.14 Querying Users via Session Token	503
14.15 Deleting Sessions	506
14.16 Sign Out Handler	507

CONTENTS	xiii
14.17 Sign Out Link	510
14.18 Session Exercises	513
14.18.1 Ex1 - Move the token management code to its own type	513
14.18.2 Ex2 - Add a template for the current user page	513
14.18.3 Ex3 - Try using a JOIN in SQL	513
15 Improved SQL	515
15.1 SQL Relationships	515
15.2 Foreign Keys	519
15.2.1 Foreign Keys with Existing Tables	521
15.2.2 Updating our code	522
15.3 On Delete Cascade	523
15.4 Inner Join	525
15.5 Left, Right, and Full Outer Join	527
15.6 Using Join in the SessionService	529
15.7 SQL Indexes	532
15.8 Creating PostgreSQL Indexes	536
15.9 On Conflict	538
15.10 Improved SQL Exercises	540
15.10.1 Ex1 - Design SQL database tables and relationships for an app	540
15.10.2 Ex2 - What are use cases for each type of JOIN?	541

16 Schema Migrations	543
16.1 What Are Schema Migrations?	543
16.2 How Schema Migration Tools Work	546
16.3 Installing <code>pressly/goose</code>	548
16.4 Converting to Schema Migrations	554
16.5 Schema Versioning Problem	557
16.6 Running Goose with Go	565
16.7 Embedding Migrations	570
16.8 Go Migration Files	572
16.9 Removing Old SQL Files	575
16.10 Goose SQL Exercises	576
16.10.1 Ex1 - Wrap the Goose CLI	576
17 Current User via Context	577
17.1 Using Context to Store Values	577
17.2 Improved Context Keys	582
17.3 Context Values with Types	586
17.4 Storing Users as Context Values	588
17.5 Reading Request Context Values	592
17.6 Set the User via Middleware	594
17.7 Requiring a User via Middleware	600

CONTENTS	xv
17.8 Accessing the Current User in Templates	608
17.9 Request-Scope Values	613
17.10 Context Exercises	616
17.10.1 Ex1 - Optimize the SetUser Middleware	616
18 Sending Emails to Users	617
18.1 Password Reset Overview	617
18.2 SMTP Services	621
18.3 Building Emails with SMTP	625
18.4 Sending Emails with SMTP	631
18.5 Building an Email Service	634
18.6 EmailService.Send	636
18.7 Forgot Password Email	641
18.8 ENV Variables	643
18.9 Email Exercises	650
18.9.1 Ex1 - Generate emails with templates	650
19 Completing the Authentication System	651
19.1 Password Reset DB Migration	651
19.2 Password Reset Service Stubs	653
19.3 Forgot Password HTTP Handler	658
19.4 Asynchronous Emails	664

19.5 Forgot Password HTML Template	666
19.6 Initializing Services with ENV Vars	670
19.7 Check Your Email HTML Template	676
19.8 Reset Password HTTP Handlers	679
19.9 Reset Password HTML Template	682
19.10Update Password Function	686
19.11PasswordReset Creation	688
19.12Implementing Consume	693
19.13Password Reset Exercises	696
19.13.1 Ex1 - Implement passwordless sign-in	696
19.13.2 Ex2 - Add an option to update a user's email address .	697
20 Better Error Handling	699
20.1 Inspecting Errors	699
20.2 Inspecting Wrapped Errors	704
20.3 Designing the Alert Banner	706
20.4 Dynamic Alerts	708
20.5 Removing Alerts with JavaScript	710
20.6 Detecting Existing Emails	711
20.7 Accepting Errors in Templates	714
20.8 Public vs Internal Errors	717

20.9 Creating Public Errors	720
20.10 Using Public Errors	722
20.11 Better Error Handling Exercises	725
20.11.1 Ex1 - Update our controllers	725
21 Galleries	727
21.1 Galleries Overview	727
21.2 Gallery Model and Migration	730
21.3 Creating Gallery Records	732
21.4 Querying for Galleries by ID	733
21.5 Querying Galleries by UserID	735
21.6 Updating Gallery Records	736
21.7 Deleting Gallery Records	737
21.8 New Gallery Handler	738
21.9 views.Template Name Bug	739
21.9.1 How do we fix it?	742
21.10 New Gallery Template	743
21.11 Gallery Routing and CSRF Bug Fixes	745
21.11.1 CSRF Bug	746
21.12 Create Gallery Handler	748
21.13 Edit Gallery Handler	750

21.14 Edit Gallery Template	753
21.15 Update Gallery Handler	756
21.16 Gallery Index Handler	758
21.17 Discovering and Fixing a Gallery Index Bug	762
21.18 Gallery Index Template Continued	764
21.19 Show Gallery Handler	767
21.20 Show Gallery Template and a Tailwind Update	771
21.21 Extracting Common Gallery Code	775
21.22 Extra Gallery Checks with Functional Options	779
21.23 Delete Gallery Handler	783
21.24 Gallery Exercises	784
21.24.1 Ex1 - Use middleware to lookup the gallery.	784
21.24.2 Ex2 - Add a published/unpublished option to galleries. .	785
21.24.3 Ex3 - Create a separate route for publicly viewing galleries.	785
22 Images	787
22.1 Images Overview	787
22.2 Setting Up Test Images	790
22.3 Adding the ImagesDir to the GalleryService	791
22.4 Globbing Image Files	793
22.5 Adding Filename and GalleryID to the Image Type	797

CONTENTS	xix
----------	-----

22.6 Adding Images to the Show Gallery Page	798
22.7 Show Image Handler	800
22.8 Querying for a Single Image	803
22.9 URL Path Escaping Image Filenames	805
22.10 Adding Images to the Edit Gallery Page	808
22.11 Delete Image Form	810
22.12 Delete Image Service Func	812
22.13 Delete Image Handler	812
22.14 Checking for Filename Vulnerabilities	814
22.15 Upload Image Form	816
22.16 Image Upload Handler	821
22.17 Creating Images in the GalleryService	825
22.18 Detecting Content Type	827
22.19 Rendering Content Type Errors	830
22.20 Deleting Images on Gallery Deletion	831
22.21 Redirect to Galleries After Auth	832
22.22 Image Exercises	833
22.22.1 Ex1 - Make the image extensions customizable	833
22.22.2 Ex2 - Make the image content types customizable	833
22.22.3 Ex3 - Create an ImageService	834

23 Preparing for Production	835
23.1 Loading All Config via ENV	835
23.2 Docker Compose Overrides	839
23.3 Building Tailwind Locally	841
23.4 Tailwind Via Docker	844
23.5 Serving Static Assets	848
23.6 Making main Easier to Test	851
23.7 Running our Go Server via Docker	852
23.8 Multi-Stage Docker Builds	855
23.9 Tailwind Production Build	856
23.10 Caddy Server via Docker	857
23.11 Production Setup Exercises	858
23.11.1 Ex1 - Experiment with a new app port	858
24 Deploying	859
24.1 Creating a Digital Ocean Droplet	859
24.2 Setting up DNS	862
24.3 Installing Git on the Server	863
24.4 Setting Up a Bare Git Repo	864
24.5 Setting Up a Local Git Repo	866
24.6 Checking Out Our Code on the Server	867

24.7 Email Sending Server Setup	868
24.8 Production .env File	869
24.9 Install Docker in Prod	869
24.10 Production Caddyfile	871
24.11 Production Data Directories	871
24.12 Running Our App in Prod	875
24.13 Post-receive Deploy Updates	876
24.14 Deploy via Git	877
24.15 Logging Services	879
24.16 Deploy Exercises	882
24.16.1 Ex1 - Explore services offered by Digital Ocean . . .	882
25 Bug Fixes and Updates	883
25.1 Fixing a Caddy Data Bug	883
25.2 Go 1.22's Updated ServeMux	885

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 changes. 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 localhost:3000. If all went according to plan, you should see a web page with “Welcome to my awesome site!” in a somewhat large font.

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

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

The code in this course is hosted on a private GitHub repo. This means you must request access to the code. This can be done at [courses.calhoun.io/github/invite](#) after purchasing the course.

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 at courses.calhoun.io/slack/invite after purchasing the course.

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 directory - `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 will 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, here 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 `\n` 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 requested, 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.

1.9. REGISTERING OUR HANDLER FUNCTION AND STARTING THE WEB SERVER

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 using 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.

1.9. REGISTERING OUR HANDLER FUNCTION AND STARTING THE WEB SERVER

```
package main

// May be provided automatically by your editor, so they may be removed if you hit save right away
import (
    "fmt"
    "net/http"
)

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

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

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

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

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

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

sense as you progress.

1.10 Go Modules

There are roughly two reasons to use Go modules:

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

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

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

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

For both of these reasons, we will 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 li-

ibraries 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 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.

Aside 1.8. Common Issues

If running **go install** works, but running the **lenslocked** binary gives an error along the lines of, “command not found: **lenslocked**”, chances are you do not have your

\$PATH environment variable set to include Go binaries. The process to do this will vary based on your operating system, but you ultimately want to add something like the line below to a file like `.bash_profile` or `.zshrc` and then restart your terminal:

```
export PATH=$PATH:$($ (go env GOPATH) )/bin
```

This may vary a bit with Windows, since it uses backslashes (\) instead of forward slashes in paths, and you may have to update the PATH somewhere else. It is also possible that the Go installed will do this for you, but I am unsure.

If you are using Windows it is also possible your binaries will have the `.exe` file extension. 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 instance, if you are no longer using a few libraries. You can run `go mod tidy` and this should handle tidying things up.

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

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 localhost:3000. You should now see your HTML being rendered as plain text.

Change your code to set the content type to `text/html` so that our page works again. 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@calhoun.io\">jon@calhoun.io</a></p>")
}
```

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 localhost:3000 without any path, then try the `/contact` path. Last, try paths we didn't define behavior for like `/signup`. What results are you seeing?

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

If the path matches `/`, we use the `homeHandler` to render the home page. If the path matches `/contact` we call the `contactHandler` function. If neither of those cases occur we 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 now 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 `HandlerFuncs`
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 provides 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** interface.
- **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 "about":
        aboutHandler(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 help 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. Additional 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 from 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 _____
exclamation           adverb

as he jumped into his convertible

_____ and drove off with his
noun

_____ 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 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 always 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` directory 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 a 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 relative 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 and 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.Println` 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 Age
```

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 comic 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>
```

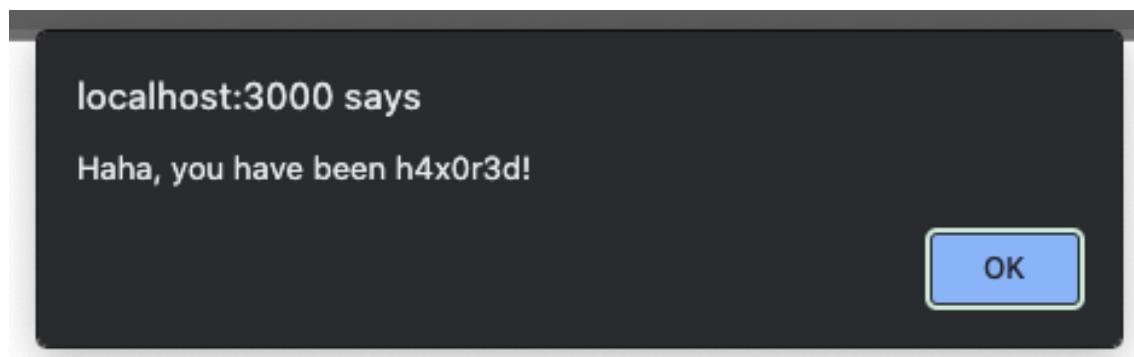
We are currently rendering the user's **bio** exactly as they provided it, which seems fine for now, but it could actually lead to XSS. Imagine if a user input the following as their bio information:

```
<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 result, 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:

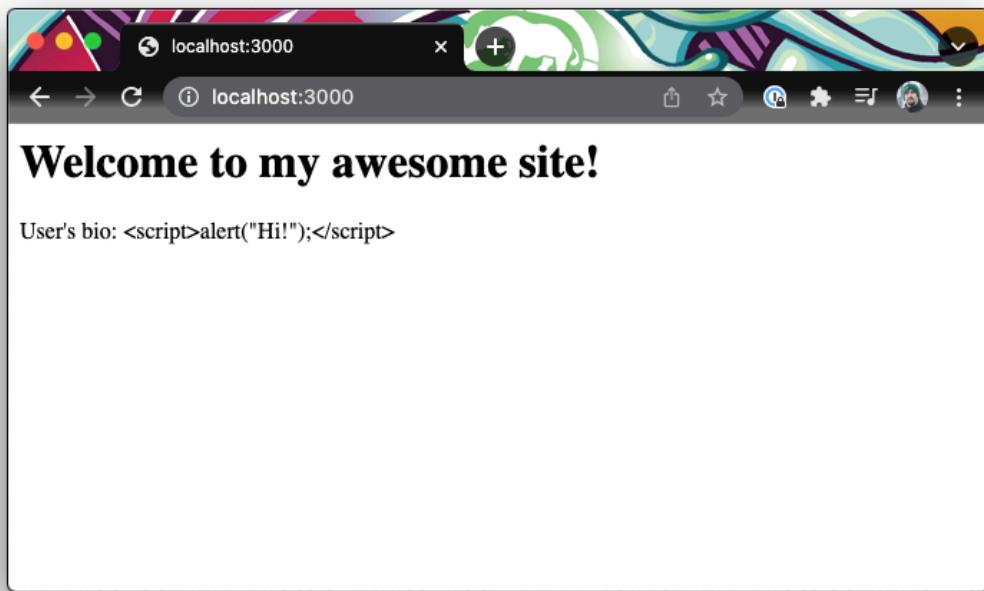
```
&lt;script&gt;alert("Hi!");&lt;/script&gt;
```

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

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("Hi!");&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 to add 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 `\u003c` 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 servers 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 that is specific to the operating system the code is running on. For instance, if we were running on Mac OS it would give us the following.

```
templates/home.gohtml
```

On the other hand, if we were running Windows we would see a backslash being used instead.

```
templates\home.gohtml
```

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 template that doesn't exist, or if we tried to use an invalid function, we would see an 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 defined
... + 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(tpPath)
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 the terminal. 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.StatusInternalServerError)
        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.StatusInternalServerError)
        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.StatusInternalServerError)
        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 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 fig-

uring 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.StatusInternalServerError)
        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.StatusInternalServerError)
        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/template/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 implementation
```

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 indie 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.

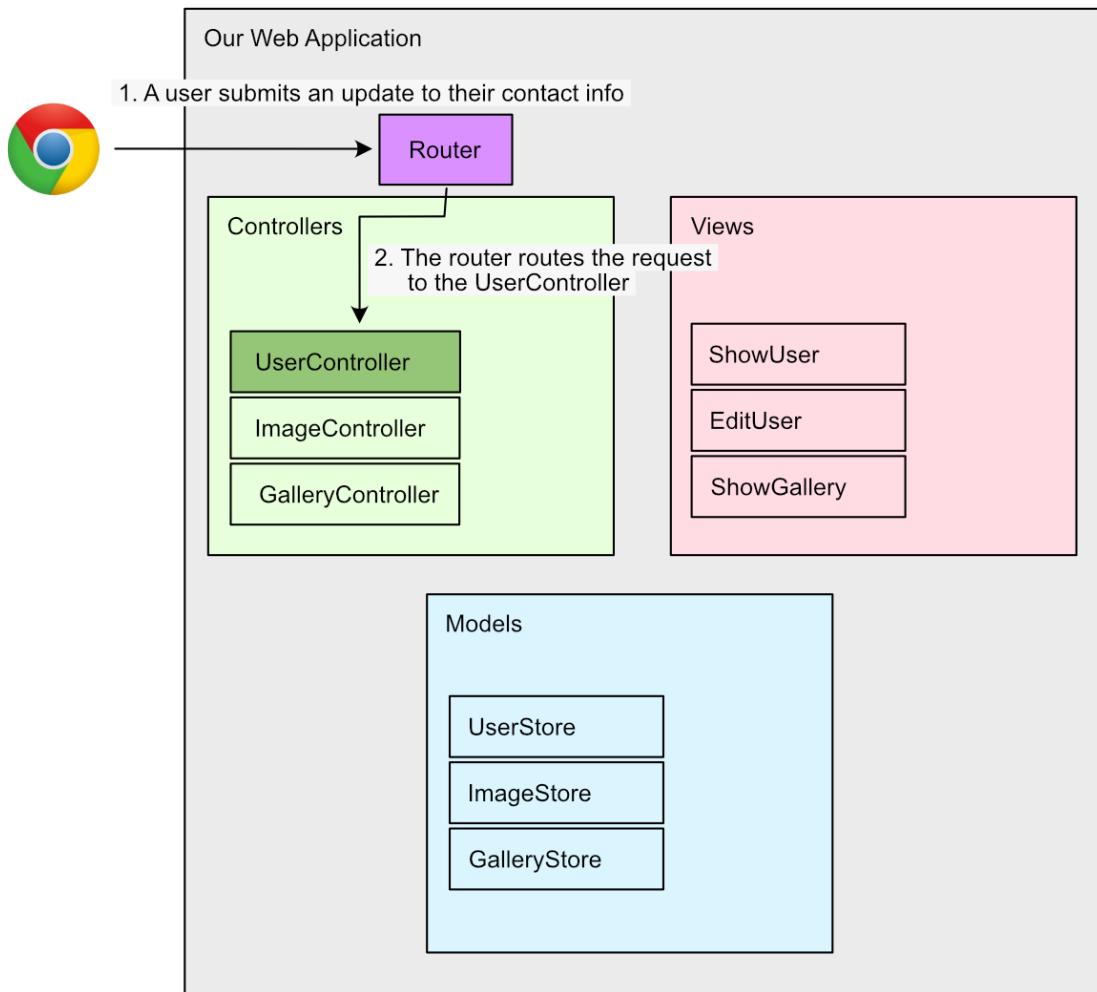
Source Code

- [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.

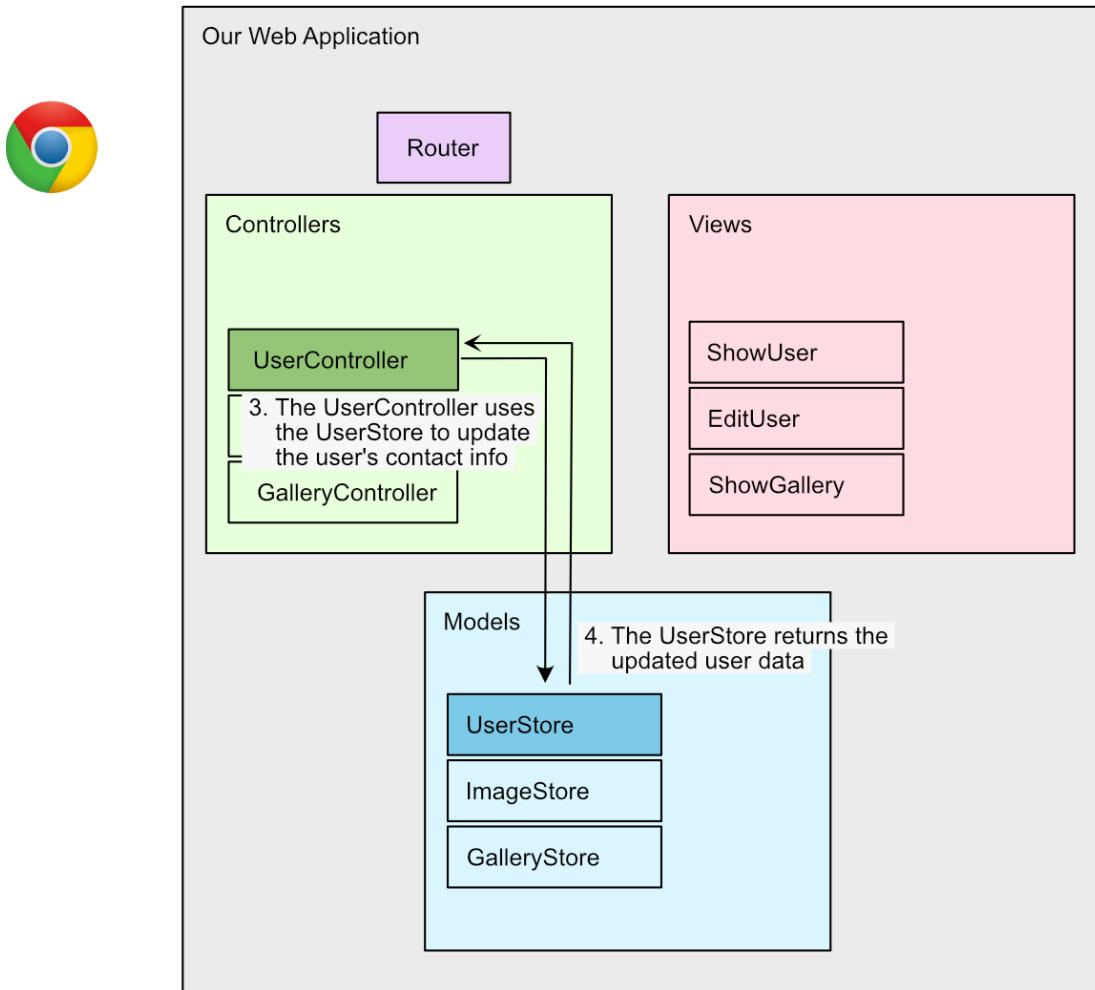
```
// Theoretical code. Don't add it to your project
package controllers

// 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.



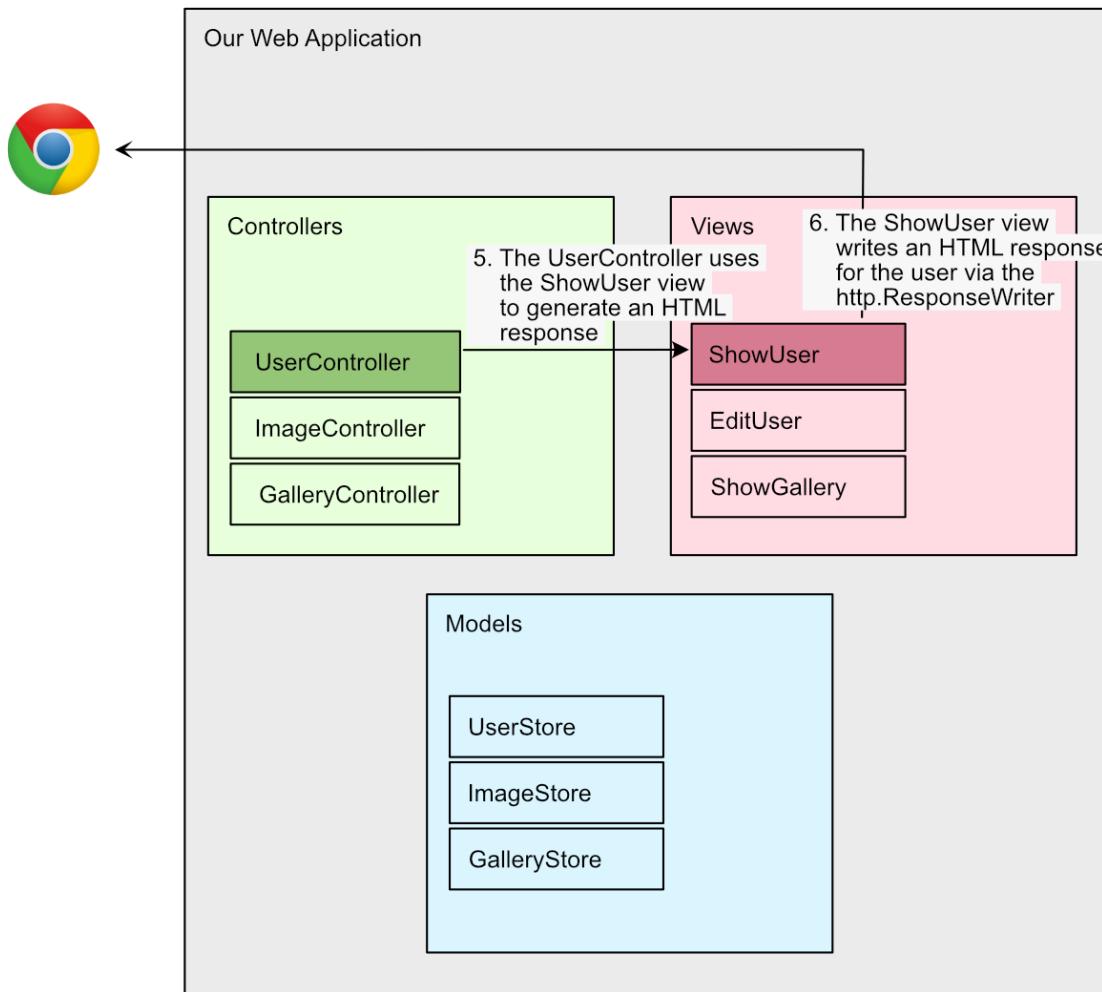
3. The UserController uses the UserStore to update the user's contact info

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 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.StatusInternalServerError)
        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.Error`

We will discuss `fmt.Error` 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.StatusInternalServerError)
        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.StatusInternalServerError)
        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 is 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 functions 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(tp1 views.Template) http.HandlerFunc {
    return func(w http.ResponseWriter, r *http.Request) {
        tp1.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 `tp1` 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 reinforce 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.Println(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 developer's 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 developer's 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 ipsum 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(tp1 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@lenslocked.com</a>`
        },
    }
    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.

```
<html>
  <head>..</head>
  <body>
    <h1>FAQ Page</h1>
    <pre>
      "{'Is there a free version? Yes! We offer a free trial for 30 days on any paid plans.'} {'What are your support hours? We
      have support staff answering emails 24/7, though response times may be a bit slower on weekends.'} {'How do I contact
      support? Email us - <a href='mailto:support@lenslocked.com'>support@lenslocked.com</a>}]" == $0
    </pre>
  </body>
</html>
```

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.StatusInternalServerError)
    }
}
```

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 template 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(tp1 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(tp1 views.Template) http.HandlerFunc {
    questions := []struct {
        Question string
        Answer   template.HTML
    }{
        // ... unchanged
    }
    // ... unchanged
}

```

```
    }{  
    // ...  
}
```

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 update 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 do so 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.gohtml` 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 template.

```
<html>
  <body>
    <!-- Note: The . passes all of our data into the "page" template so it also has access to it -->
    {{template "page" .}}
    <p>Copyright Jon Calhoun 2024</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 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 **** 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 ``.
- `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 `` 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.

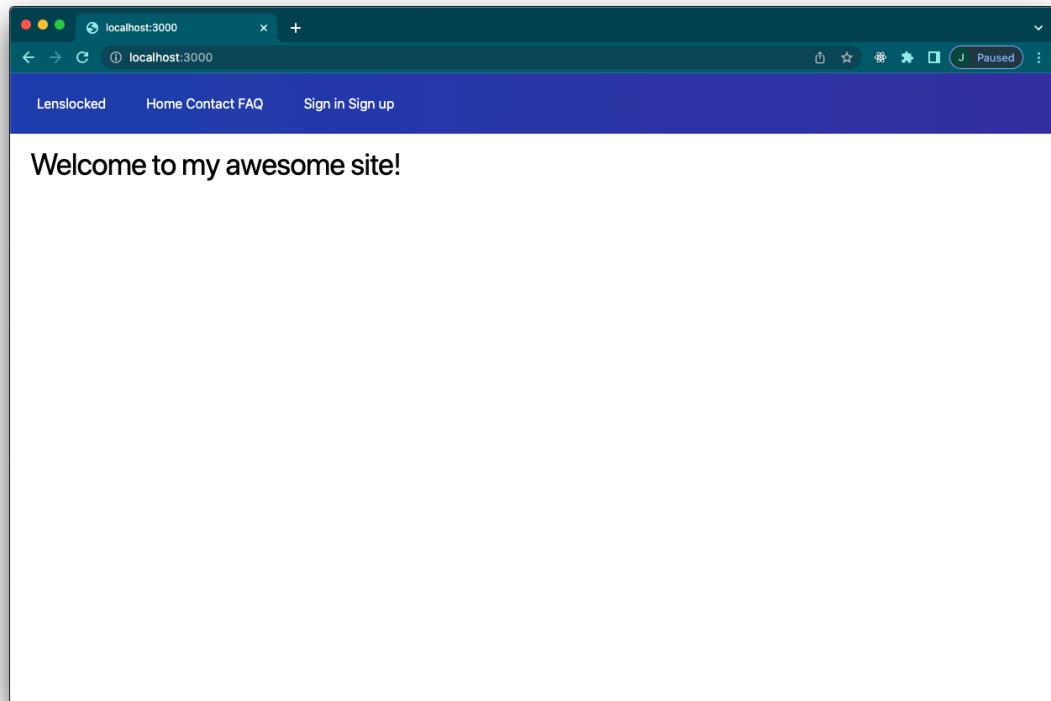
```
<nav class="px-8 py-6 flex items-center space-x-12">
```

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>
```

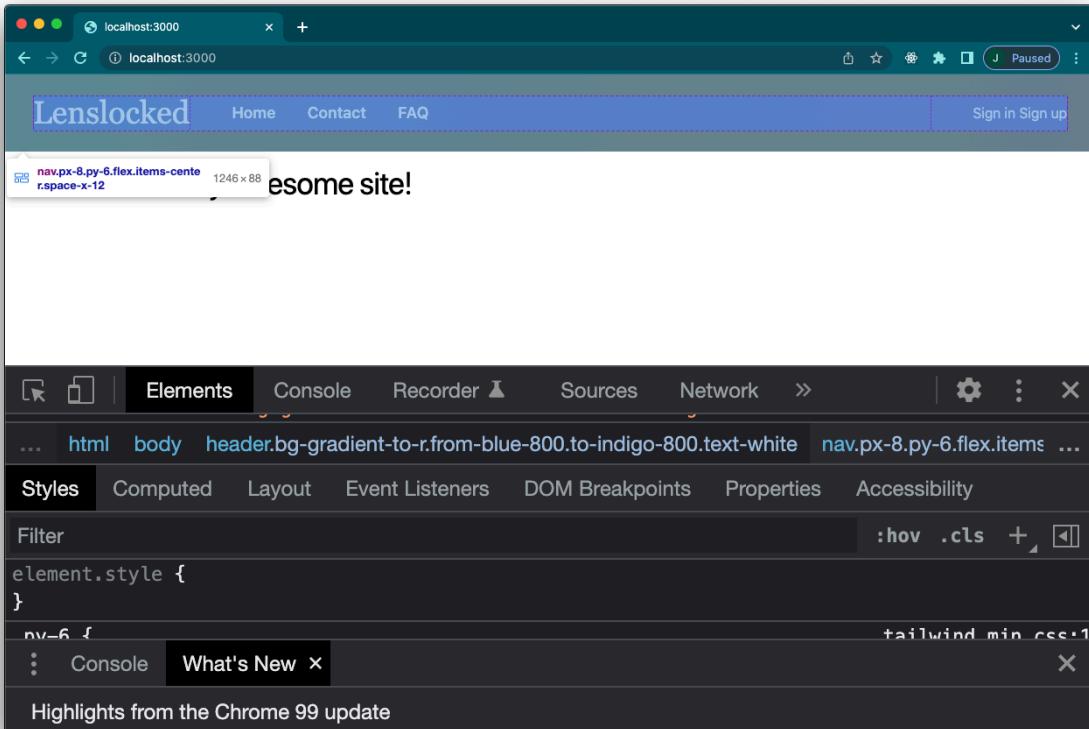
It is common to move the sign up and sign in links to the top right corner of the navbar. One way to achieve this with Tailwind is to tell our links section to grow to expand any unused space with the **flex-grow** class. This only works for direct children of an HTML element with the **flex** class 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 at 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 versions 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, but 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.

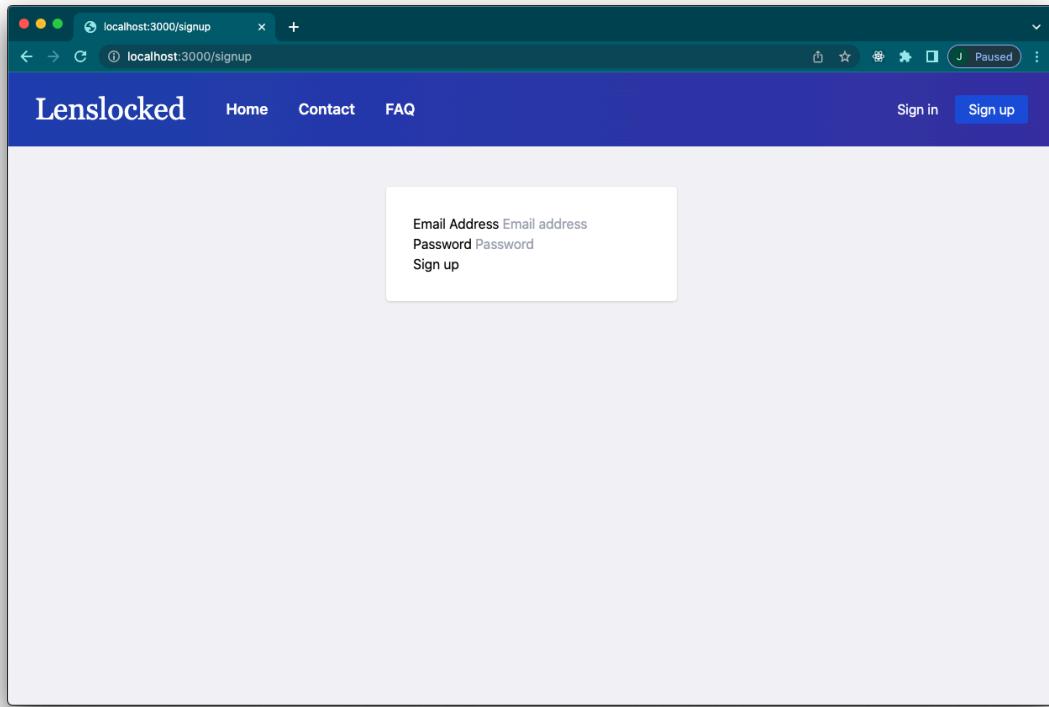
```
<body class="min-h-screen bg-gray-100">
```

Next, we are going to wrap the entire sign up form in a `<div>` 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 `<div>` 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
    type="submit"
    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>
```

Aside 8.3. The type tag

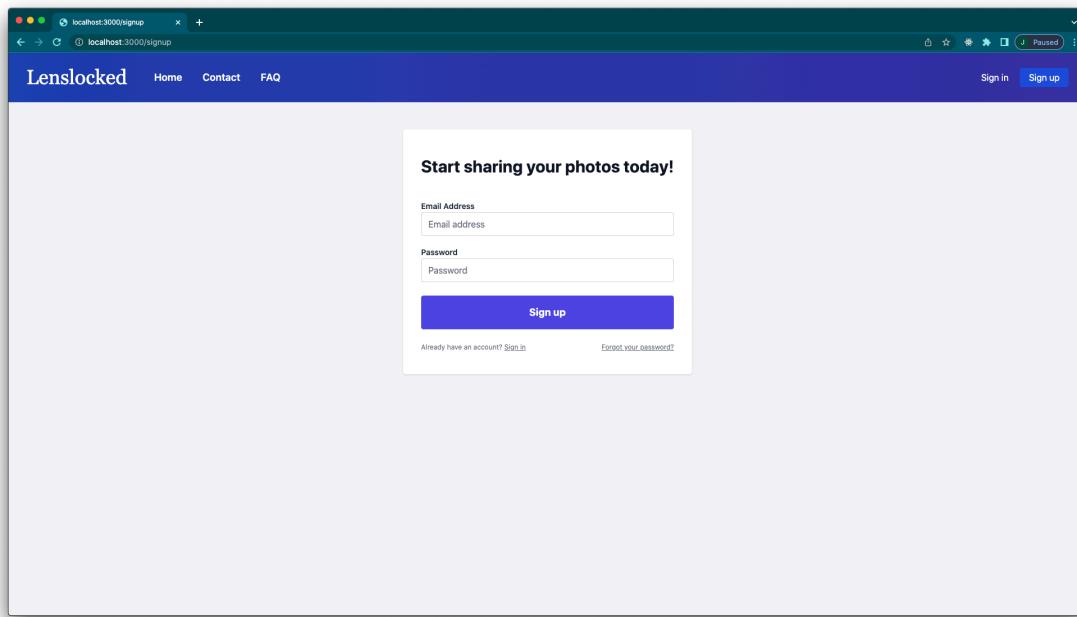
The source code for the written version of the course does not have the `type` tag in the button. It isn't strictly required, but I added it here since I have it in the screencasts and it felt like a good thing to have.

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 customer service representative over the phone. You call the customer support number and the representative 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 (`/galleries/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
<code>GET</code>	<code>/galleries</code>	Read a list of galleries
<code>GET</code>	<code>/galleries/:id</code>	Read a single gallery
<code>POST</code>	<code>/galleries</code>	Create a gallery
<code>PUT</code>	<code>/galleries/:id</code>	Update a gallery
<code>DELETE</code>	<code>/galleries/:id</code>	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 8.4. 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 8.5. 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.go**:

```
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 8.6. 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 from 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 packages. 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 8.7. 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 accept 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 8.8. 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 especially 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 8.9. 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 8.10. Idempotent functions

`ParseForm` is an idempotent function. What this means is that our code could call the `ParseForm` function multiple times in a single web request, and everything would work the same as when we call `ParseForm` 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 8.11. 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 8.12. 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 is 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=user@gmail.com> and we should see the email address being pre-populated.

Aside 8.13. 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 autocomplete 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 an 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**.

```
code docker version
# You should see a version output. This may be large.
code 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:
  db: # Our Postgres database
    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

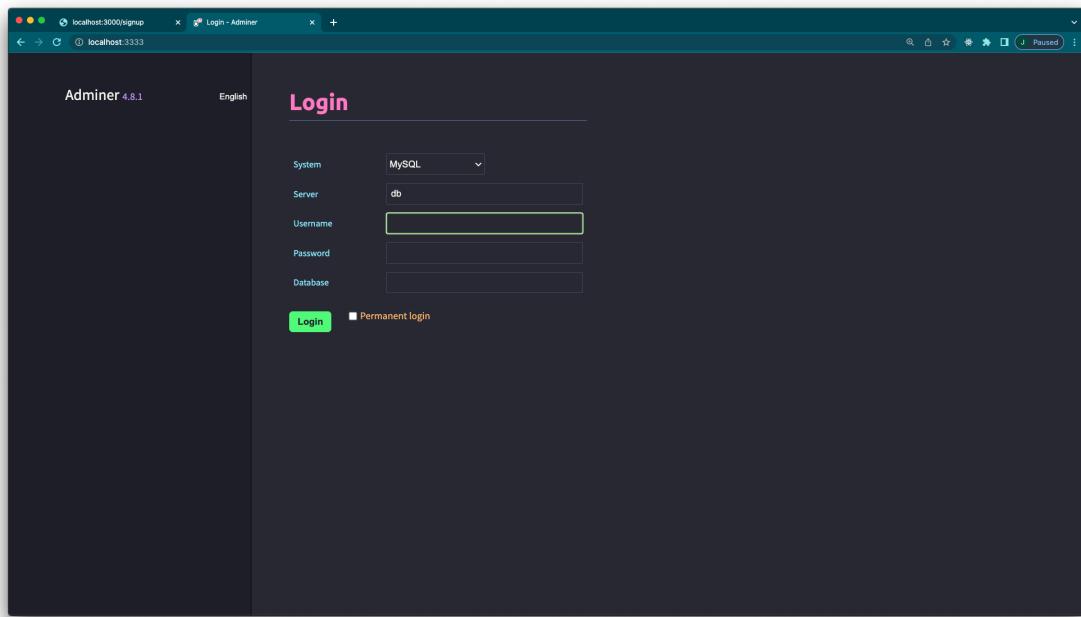
other development related tasks. It is using the [adminer image](#), which uses the software from [adminer.org](#).

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 `\q` 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.

	A	B	C	D	E	F	
1	id	age	first_name	last_name	email		
2							
3							
4							
5							
6							
7							
8							
9							
10							
11							

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
<code>int</code>	This is used to store integers between -2147483648 and 2147483647.
<code>serial</code>	This is used to store integers between 1 and 2147483647. The big difference b
<code>varchar</code>	This is like a string in Go or other programming languages, except we have to
<code>text</code>	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
<code>UNIQUE</code>	This ensures that every record in your database has a unique value for the
<code>NOT NULL</code>	This ensure that every record in your database has a value for this field. W
<code>PRIMARY KEY</code>	This constraint is similar to combining both <code>UNIQUE</code> and <code>NOT NULL</code> 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.

The screenshot shows the Adminer 4.8.1 interface. On the left, there's a sidebar with "Adminer 4.8.1" and "English" at the top. Below that are dropdown menus for "DB:" set to "lenslocked" and "Schema:" set to "public". There are four buttons: "SQL command" (which is highlighted in blue), "Import", "Export", and "Create table". A message "No tables." is displayed below the buttons. On the right, the main area has a header "PostgreSQL » db » lenslocked » public » SQL command" and a title "SQL command". The SQL command "DROP TABLE IF EXISTS users;" is entered into the text area. Below the text area, there's a message "Query executed OK, 0 rows affected. (0.003 s) Edit". At the bottom, there are buttons for "Execute" (highlighted in green), "Limit rows:", "Stop on error" (unchecked), "Show only errors" (unchecked), and a "History" button.

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.

<code>id</code>	<code>age</code>	<code>first_name</code>	<code>last_name</code>	<code>email</code>
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 semicolon (;).

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.

<code>id</code>	<code>age</code>	<code>first_name</code>	<code>last_name</code>	<code>email</code>
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 is 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/computing-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 learn 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 library, 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 from 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 dbname=postgres sslmode=disable")
    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 dbname=baloo")
    // ... 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 driver

} // ...

```

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 sslmode=disable")
```

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 able 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 now 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 running 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 Ad-miner (<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.

The screenshot shows the Adminer 4.8.1 web-based PostgreSQL client. On the left, the sidebar shows the database is 'lenslocked' and the schema is 'public'. Below the sidebar, there are tabs for 'orders' and 'users', both set to 'select'. The main area is titled 'Select: users' and shows the results of the query: 'SELECT * FROM "users" LIMIT 50'. The results table has columns 'id', 'name', and 'email'. One row is selected, showing 'id' as 1, 'name' as 'Jon Calhoun', and 'email' as 'jon@calhoun.io'. At the bottom of the results table, there are buttons for 'Save', 'Edit', 'Clone', and 'Delete'. The status bar at the bottom of the Adminer interface shows 'Import'.

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:

<code>id</code>	<code>name</code>	<code>email</code>
1	Jon Calhoun	<code>jon@calhoun.io</code>
(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 the `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 have 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.

<code>id</code>	<code>user_id</code>	<code>amount</code>	<code>description</code>
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 be 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 adds an additional cost to your bottom line. In many cases this cost is worth paying, but there are also circumstances where the cost isn't feasible. For instance, while providing Gophercises for free it would be hard to pay hundreds of dollars per month for an authentication service.

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 numbers
    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 Plain-text 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 user's 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 user's 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 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:

<code>id</code>	<code>password</code>
--	-----
1	<code>c66d8e7d45e7</code>
2	<code>eff5364b19f5</code>
3	<code>93435c1a22a9</code>
4	<code>b78e5973fae1</code>
5	<code>7c12e834ef44</code>

6		79e1f4ccf82e
7		890b86bf0779
8		df5d6e16ac62
9		30fa6a3d24fe

The attacker cannot determine any user's password directly from the hash, but since they also know our hashing function, they could create a list of common passwords that they think users might be using. This could even include randomly generated passwords produced by code the hacker wrote. Below is a shortened list as an example.

password
abc123
secret
none

Next, the hacker could use the information about our hashing function to run all of these through code that generates a table that has both the original password and the hash value it produces.

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 it with the rainbow table to see if any of the password hashes match. Comparing this to our leaked database they could learn the following user passwords:

<code>id</code>	<code>password</code>	<code>cracked_pw</code>
1	<code>c66d8e7d45e7</code>	<code>abc123</code>
2	<code>eff5364b19f5</code>	
3	<code>93435c1a22a9</code>	<code>password</code>
4	<code>b78e5973fae1</code>	
5	<code>7c12e834ef44</code>	
6	<code>79e1f4ccf82e</code>	<code>none</code>
7	<code>890b86bf0779</code>	
8	<code>df5d6e16ac62</code>	<code>secret</code>
9	<code>30fa6a3d24fe</code>	

While it didn't tell the attacker every user's password, it did give them several matches and they could either try logging in as that user, or they could even try other websites, like PayPal, to see if the user used the same password on multiple sites.

The first challenge for hackers using rainbow tables is that it takes a while to create each rainbow table. Hashing millions or billions of possible passwords is feasible, but it takes time. As a result, this attack works best when it can be applied to a large set of hashed passwords all at once.

That is where salting passwords becomes beneficial. Salting a password is the act of adding random data to each password before hashing it. Every user would get their own unique salt value, so we also store the salt in our database so that we can use it when verifying a user's password.

For example, when a user signs up they might type in their password of `abc123`. At that time we would generate the random salt value - let's say `ja08d` - and we would append it to their password giving us the string `abc123-ja08d`. We could then hash this new string, getting the hash value of `64047ee6222f`. We would then store all of this in our database.

<code>id</code>	<code>password_hash</code>	<code>salt</code>
1	<code>64047ee6222f</code>	<code>ja08d</code>

When the user signs in and gives us a password to authenticate, we know to add the **ja08d** salt to the input before hashing it and comparing it to their hashed password value, so we haven't lost any functionality, but how does this help us with rainbow table attacks?

Each user should be given their own salt value. This value will be stored in the database, so in the case of a data breach a hacker may have access to the salt values, but by using a unique salt for each password we have effectively made rainbow tables useless. Attackers now need to generate a rainbow table separately for each salt, which is equivalent to just guessing each individual password without the use of a rainbow table.

Aside 11.3. bcrypt has salt built in

We will be using **bcrypt** to hash our passwords, and because this algorithm was designed specifically for passwords it actually handles generating salts for us. Looking back at our **abc123** example, rather than having a new column in the database, bcrypt will append the salt to the hashed password giving a result like **64047ee6222f.ja08d**. Later when comparing passwords bcrypt knows to pull the **ja08d** salt out of the hash, allowing us to avoid a whole new column in our database for the salt.

Another related topic is a **password pepper**. Like a password salt, a pepper is random value added to the password before hashing, but rather than being user-specific, a pepper is application specific. The main benefit here is that because a pepper is the same across the entire app, it does not need to be stored in the database and can instead be stored in the application. The hope here is that if an attacker gains access to our database, but not other parts of our application, they will have a harder time cracking passwords.

In practice a pepper isn't necessary and the rest of our security measures are sufficient, so we won't be including it in our app, but it is worth knowing what

it is in case it comes up in the future.

Aside 11.4. 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 isn't 100% clear where it came from.

Some say it was inspired by Roman soldiers who would salt the earth to make it less hospitable during wars. Others claims it stems from [salting a mine](#), the act of adding gold or silver to an ore sample with the intent to deceive anyone who was considering buying the mine.

Regardless, salting was the first term to be coined, and then the term pepper came about because salt and pepper is a popular duo (at least in the United States).

11.6 Learning bcrypt with a CLI

Now that we understand the importance of hashing our passwords, let's look at how to use [bcrypt](#) to properly hash a password. To do this, we are going to build a simplified command line interface (aka CLI). In this lesson we will look at how to set up the CLI, then in the next few lessons we will add the bcrypt hashing and comparison functionality. Then later in the course we will use the code from this CLI to hash our user passwords. The goal now is simply to understand how bcrypt works, which is much easier when we can run the code.

Aside 11.5. 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 tutorial. The application we are building has several potential bugs, like erorring when the user forgets an input value. The goal of this lesson is not learning how to build a CLI, but instead to get some working code that allows us to see how `bcrypt` works. The CLI bits we learn are just an added bonus.

In short, there is no need to report bugs with this CLI. It serves its teaching purpose despite those bugs.

```
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:

```
# 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 argument. We can do this using `os.Args`, which is an array of all the arguments used when running an application. Add the following code to `bcrypt.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.

```
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 is why we are seeing the weird first argument. This is the temporary binary location. The second argument is `hash`, the string we provided after our source file name when calling `go run`.

We can make this look less awkward if we build a binary and run it, but keep in mind how you run the binary might be different if on Windows.

```
# 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.

```
./bcrypt compare "secret password" "some hash value"
```

This will give us the following output:

```
0 ./bcrypt
1 compare
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 switch statement to determine what our CLI should do.

```
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 us whether we are running the `hash` or `compare` functionality of our CLI. If a second argument is not provided our app will panic, and if an invalid argument is provided our app will print out an “Invalid command” message.

When a user opts to run the `hash` functionality, we take the third argument (index `2` with zero-based indices; eg `os.Args[2]`) which should be the password we want to hash and pass it into the `hash` function. Similarly, we take the third and fourth arguments when the `compare` command is used. In both of these cases our code will panic with a runtime error if enough arguments are not provided.

We can try running both commands in our terminal.

```
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 get started with. We can wait to clean the CLI up in a future exercise.

In the next few lessons we will see how to use bcrypt to implement the hashing and comparison functionality.

Source Code

- [Source Code](#) - see the final source code for this lesson.
- [Diff](#) - see the changes made in this lesson.

11.7 Hashing Passwords with bcrypt

We are now ready to add hashing functionality to our CLI. To do this, we first need to [go get](#) the [crypto/bcrypt package](#).

```
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 standard library. The primary reasons for this is to allow the developers to maintain them under a looser set of guidelines. For example, the Go standard library ensures backwards compatibility with all API changes, where libraries under the [/x/](#) path might make breaking changes when a new major version is released.

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](#)
- [CompareHashAndPassword](#)

The first, [GenerateFromPassword](#), is used to generate a password hash from an raw user password, and even handles generating a salt for us. We need to

provide it with a cost, which we will discuss shortly, but bcrypt provides us with a reasonable default value there as well.

The second, **CompareHashAndPassword**, is used to compare a user-provided password with a hash we have stored to verify whether or not they match. Earlier we learned that this comparison should be time-constant, and the comparison function provided by bcrypt handles all that for us.

Our CLI app already has a **hash** function for us to work with, so we can open up **bcrypt.go** and update it with the code below.

```
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 it. This function requires two arguments:

- The password, as a byte slice
- The cost

Our password is a string, so we need to convert it into a byte slice. We can do this with the following Go code.

```
[]byte(password)
```

bcrypt's hash generating function also requires a cost parameter, but what is that? We didn't discuss a cost when we discussed hashing earlier.

Password hashing functions often have a variable cost that is used to dictate how much work (and sometimes memory) must be used to hash a password. As the cost goes up, malicious parties will need to spend even more CPU power creating rainbow tables.

The cost is a variable because it is meant to change over time as computers get faster. A good metaphor is a racetrack changing the number of laps in a race as cars become faster. When people were first racing on foot, you might only have them race a single lap. Then as they started to race on horseback you might increase the race to last several laps. As motor vehicles were introduced, races gradually increased until we finally have races now that last hundreds of laps around a track that would take a person quite some time to walk around.

With password hashing the race car is our CPU, and the number of laps needed to be completed is the cost passed into our hashing function. As the cost increases, hashing will take more time, but as CPUs become faster the amount of time required will decrease. Our goal is to pick a cost that is high enough to make life difficult for attackers, but low enough that the end user doesn't wait around for our server.

Figuring out and keeping track of the best cost would be pretty tedious, so bcrypt provides us with a `DefaultCost` constant that we can use.

The `GenerateFromPassword` function returns a byte slice and an error. If we receive an error we can print it out and return, but this typically shouldn't happen. When we don't have an error we want to print the hashed password out, but printing a byte slice will give us a list of numbers.

```
[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, and since a byte is a `uint8` behind the scenes, we get a list of numbers. Instead, we want to print out a string. We can achieve this by converting the byte slice into a string.

```
string(hashedBytes)
```

Aside 11.6. Not All Bytes Make Valid Characters

Not all byte slices can be converted into a valid string because not all bytes map to valid characters. The conversion will work, but printing the string out in the terminal or storing it in a database might lead to issues with the invalid characters. In this case it works because bcrypt uses base 64 encoding behind the scenes, but keep this in mind when dealing with byte slices in the future.

If we run the **hash** command we should now see a hash value being printed out.

```
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 because bcrypt is generating a unique salt for us every time we hash the password, and the salt is stored in the resulting hash value. As a result, we get a different output each time we run our CLI. Later when we learn how to compare passwords bcrypt will reuse that salt and will actually generate the same hash, which is how bcrypt is able to verify passwords.

Aside 11.7. How is the Salt Stored in the Hash?

This isn't exactly how bcrypt works, but imagine bcrypt generated the salt **ja08d** and was hashing the password **secret password**. bcrypt could hash the value **secret password-ja08d** and it might get a resulting hash like:

```
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 of a hash and then use that salt to do the comparison. In fact, bcrypt also stores the cost in the resulting hash so we do not need to provide it later when comparing passwords with a hash value.

Our CLI now supports hashing passwords! In the next lesson we will look at how to compare a password with a precomputed password hash.

Source Code

- [Source Code](#) - see the final source code for this lesson.
- [Diff](#) - see the changes made in this lesson.

11.8 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.

```
go run cmd/bcrypt/bcrypt.go compare \
"secret password" \
"some hash here"
```

Aside 11.8. 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 (\) just before we hit the enter key. This will tell the terminal that we are still creating the command we want it to execute.

We are going to be using the `CompareHashAndPassword` function to compare a password with a password hash. Open up `brypt.go` and update the `compare` function with the following code.

```
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 that we simply need to check for an error. If one is returned, it means our password didn't match or that there was some other error - like

an invalid hash was provided. We can treat all of these cases as an invalid password.

On the other hand, if no error is returned it means the password and the hash matched. In that case we can print out that the password is correct.

Using a hash calculated from an earlier run, we can test this by running the following code:

```
go run cmd/bcrypt/bcrypt.go compare \
"secret password" \
'$2a$10$GIdKwhQ8jUnH8bNS/CDDM.yhb9tnPc6sZWI4NIWytGOvGwGAHIJLO'
```

Aside 11.9. Use Single Quotes For the Hash

Characters like `$2` can have a special meaning in bash strings with double quotes (`"`). To prevent our terminal from trying to interpret these characters as something special, we need to use single quotes (`'`) for our hash string. On most keyboards this is the character on the same key as the double quotes key, but shows up when it is pressed WITHOUT pressing shift. This is not the same as the backtick character used for multi-line strings in Go.

We can test the code out by generating a few hashes and trying to validate them with valid and invalid passwords. We can even experiment with changing the cost and seeing how it affects our code.

We won't be using this CLI moving forward, but it is worth keeping around to reference when we are ready to hash our user passwords.

Source Code

- [Source Code](#) - see the final source code for this lesson.

- **Diff** - see the changes made in this lesson.

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 table 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 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 own 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 it considers the string `jon@calhoun.io` to be different than `JON@CALHOUN.IO`. We know that 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 user. 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$10$v2341SCqeIH4P5haiQkfO0q5wc0qez00Dn6t50k6Y/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\$v2341SCqeIH4P5haiQkfO0q5wc0qez00Dn6t50k6Y/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 projects 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 driver 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;
```

<code>id</code>	<code>email</code>	<code>password_hash</code>
2	<code>test@calhoun.io</code>	<code>\$2a\$10\$ygwiWK3gwntEyGSNeIlx3uFDfdBNTwFMLOn/BHOKGdR6XD7umZj/.</code>
3	<code>bob@bob.com</code>	<code>\$2a\$10\$T8L00zkhRe19mSR2ifSjo.NBCaI2v8OE1K0eVWeWjpoLMOu5i9emm</code>
<code>(2 rows)</code>		

We can even use the bcrypt CLI to test our password was hashed correctly.

```
go run cmd/bcrypt/bcrypt.go compare \
"bob123" \
'$2a$10$T8L00zkhRe19mSR2ifSjo.NBCaI2v8OE1K0eVWeWjpoLMOu5i9emm'
```

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 name 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>
    </form>
  </div>
</div>
```

```

        </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>
```

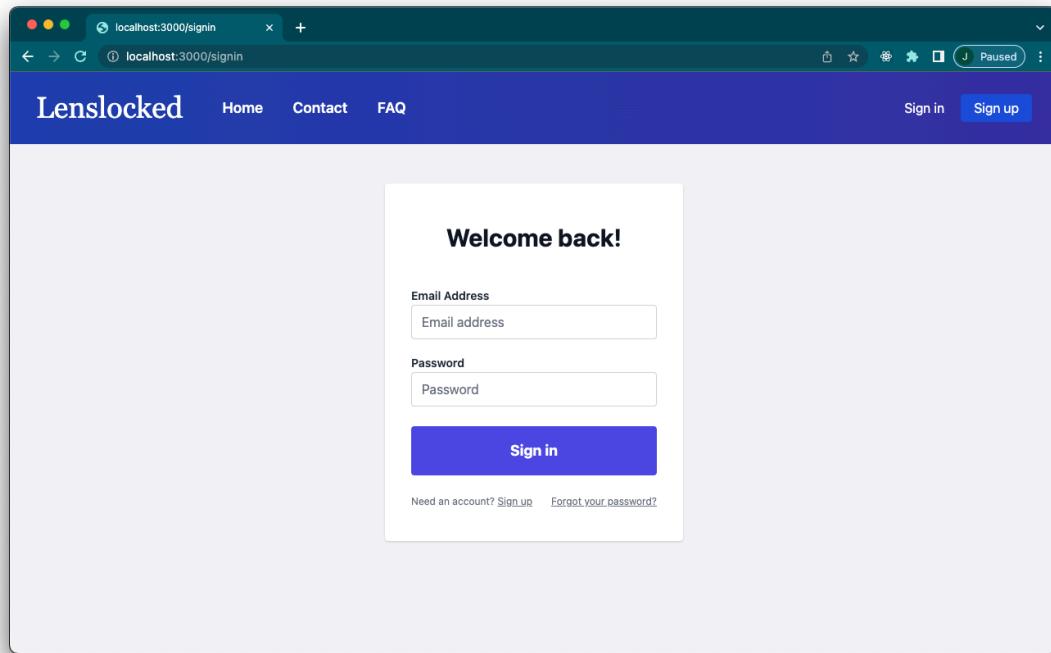
Next, we need to parse the template and set everything up. We do this inside `main.go` with the following code. Be sure to keep it next to the other line where `usersC` gets a template assigned so we can quickly find where all the user templates are setup in the future.

```
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

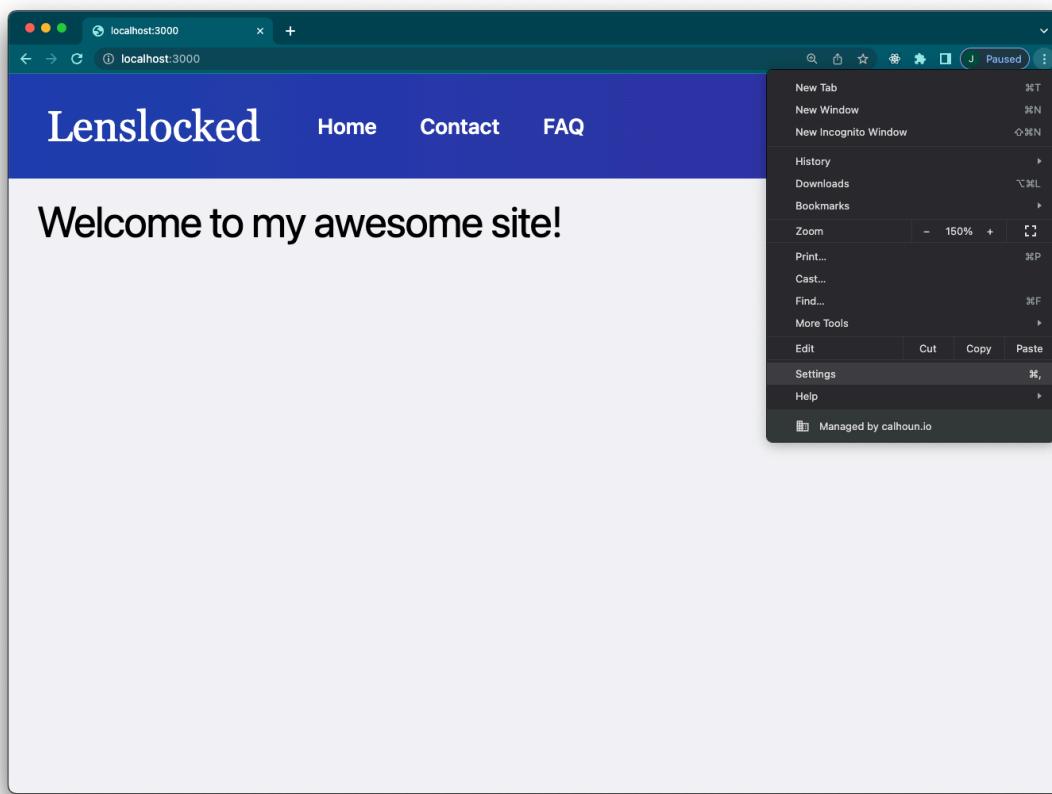
We need a cookie to proceed with this lesson, so be sure to restart the server and then sign in using the updated sign in page. With our new cookie code in place, this will result in a cookie being set that we can then proceed to examine in this lesson.

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.

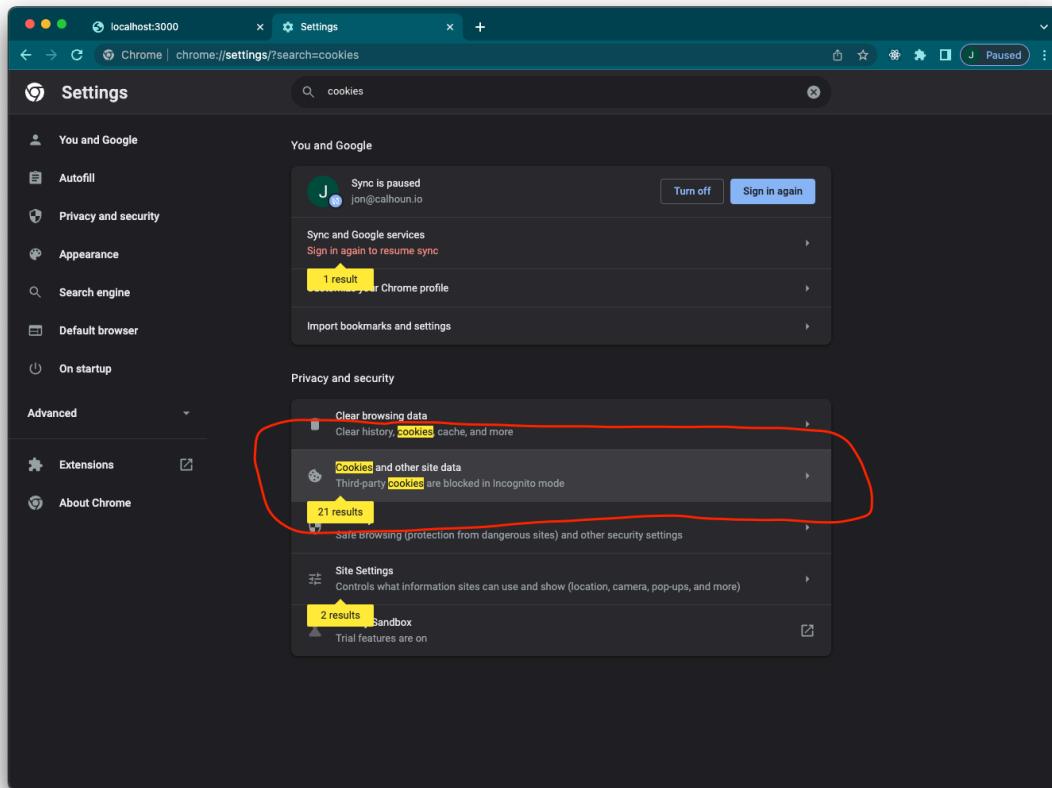
The first thing we want to do in Chrome is to open the settings. We can do this by clicking the three dots on the top right hand side of the browser window, then selecting the settings option.



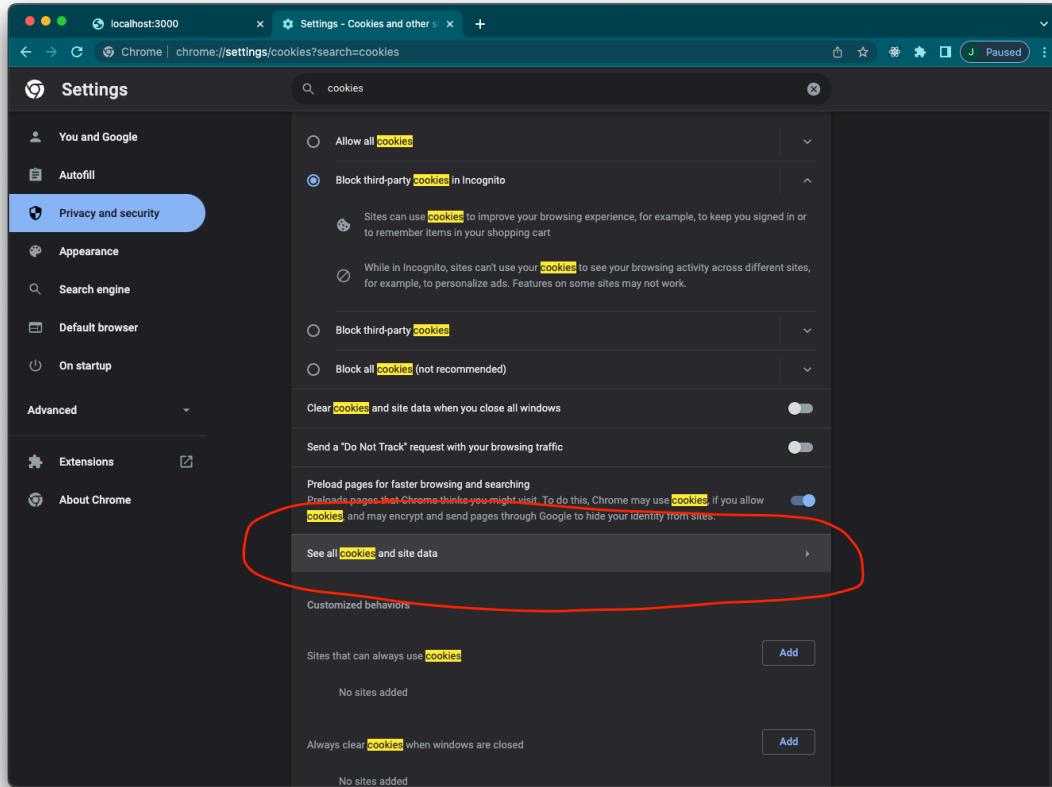
Selecting the settings option will open a new tab with Chrome settings. From here the easiest way to proceed is to type “cookies” into the settings search bar.

13.3. VIEWING COOKIES WITH CHROME

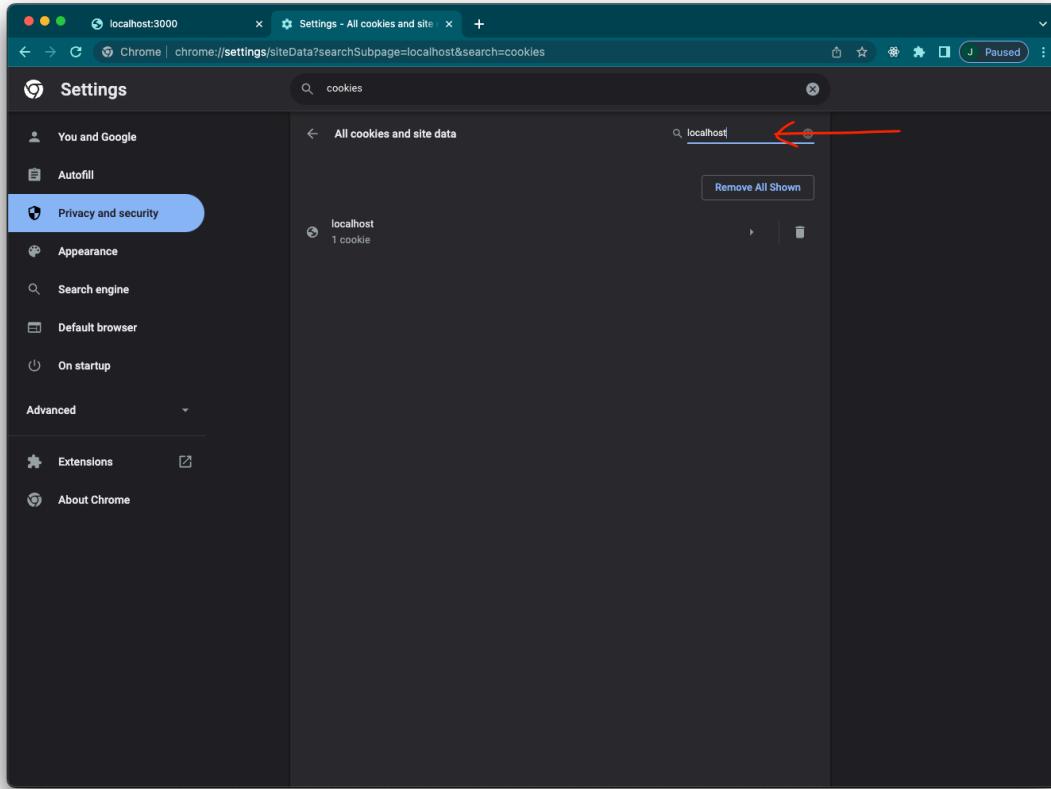
397



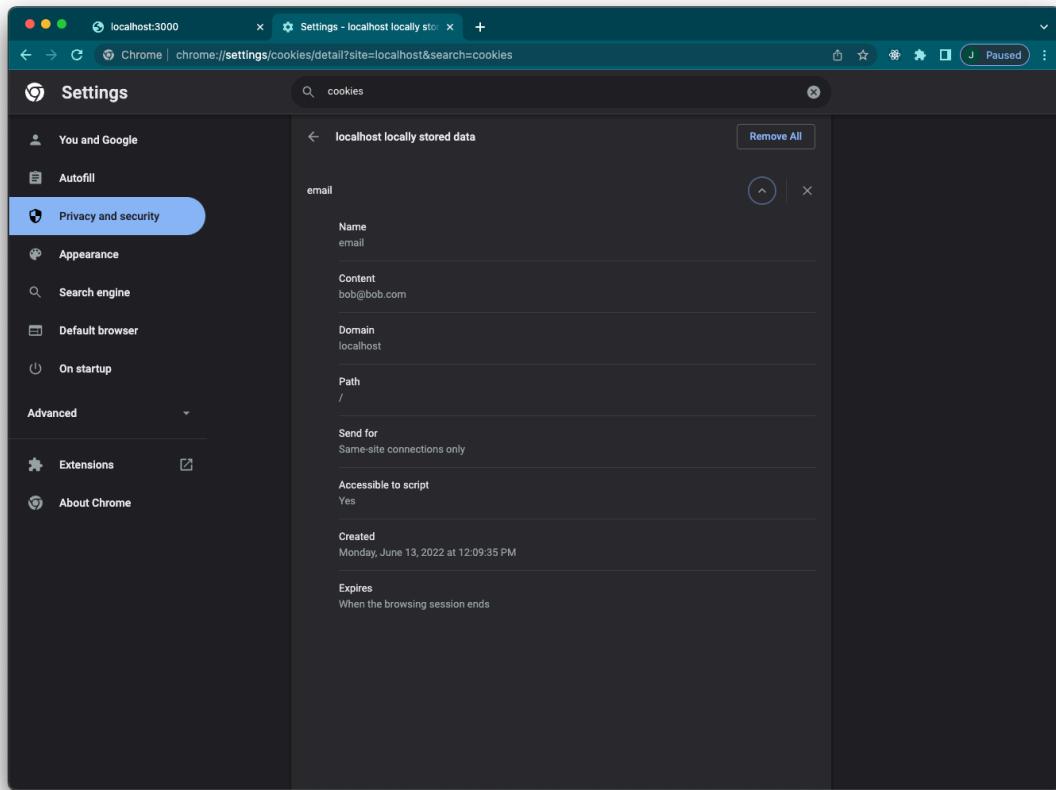
Next, select the “Cookies and other site data” option. Once on this page scroll down to the “See all cookies and site data” option.



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.

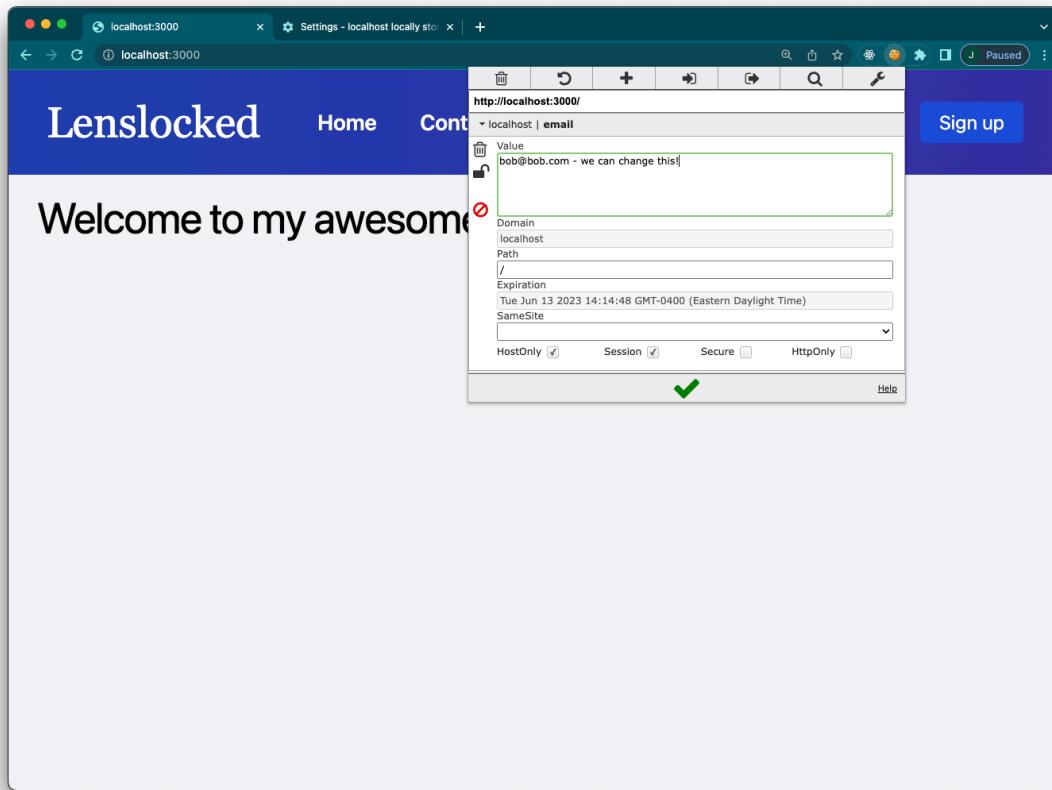


From here we can select the correct domain, **localhost**, and then select the cookie we want for that domain - **email**. We can then expand that cookie to see the information stored.



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 localhost:3000
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.

```

```

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.

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/UpPtI4FPYNh>

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 **TimerMiddleware**, 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="A6YSbCQAh6U7Hinm5rThWORzr8bpGxD1UX7EbOiw"
```

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 previously. 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
```

The first thing we are going to do is clone the template.

In Go the `net/http` package turns every web request into its own goroutine. This means that our server might be handling multiple web requests at the same time, and it is possible that two different users have made a request to the same page - for instance two users might be trying to sign up for new accounts at the

same time. When this happens, both requests will be using the same template. Normally this isn't an issue, but since every user has a unique CSRF token, we are going to be changing our template to use a request-specific function. If we were to share this template across multiple web requests, it is possible to introduce a race condition where the `csrfField` function renders the wrong CSRF token for one of our 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.

```
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.StatusInternalServerError)
        return
    }
    // ...
}
```

Aside 13.14. Could We Use a Mutex Instead?

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. Or put another way - we could ensure that only one web request could use the template at a time, which would ensure we do not have race conditions.

Both cloning the template and using a mutex are valid approaches to handling this problem. I opted to use the cloning approach as this allows us to respond to multiple web requests using the same template at the same time, which feels like

it will scale a little better at the cost of our server using a tiny bit more memory for each template clone.

Next we need to update the template's `csrfField` function to return a request-specific CSRF field. To do this, we call the `Funcs` method and pass in a new `template.FuncMap` with the updated `csrfField` implementation. Inside of the new function we use the `gorilla/csrf` package's `TemplateField` function to generate the HTML we need.

```
func (t Template) Execute(w http.ResponseWriter, r *http.Request, data interface{}) {
    tpl, err := t.htmlTpl.Clone()
    if err != nil {
        // ...
    }
    tpl = tpl.Funcs(
        template.FuncMap{
            "csrfField": func() template.HTML {
                return csrf.TemplateField(r)
            },
        },
    )
    // ...
}
```

Finally, we update the `Execute` method to render our cloned template, not the original one.

```
func (t Template) Execute(w http.ResponseWriter, r *http.Request, data interface{}) {
    tpl, err := t.htmlTpl.Clone()
    if err != nil {
        // ...
    }
    tpl = tpl.Funcs(
        // ...
    )
    w.Header().Set("Content-Type", "text/html; charset=utf-8")
    err = tpl.Execute(w, data) // <<< Update this line
    if err != nil {
```

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

Restart the server and reload the [sign up](#) page. We will now have a working `<input>` field with the correct CSRF token generated by `gorilla/csrf`.

```
<input  
    type="hidden"  
    name="gorilla.csrf.Token"  
    value="some-value-here">
```

We also need to update the sign in page to use the CSRF token, so we need to update the HTML in our template.

```
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. Visit the sign in page and 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.

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")
            },
        },
    )
    _, 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.Template
```

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.StatusInternalServerError)
        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 un-commented 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	7r1mpIjJc1
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 website 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
<hr/>	
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 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 RtGenRandom 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.

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 Readers**, 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.

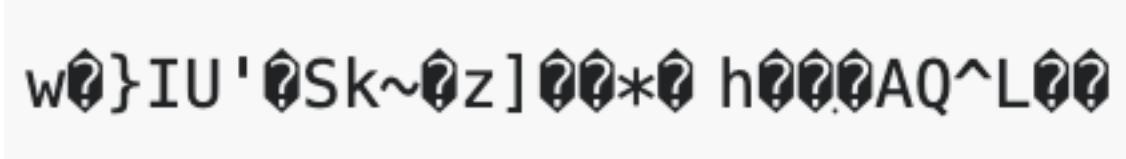
After that we check for errors and verify that we read the correct amount of data, and finally we print out the random data we read.

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 seed 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 first 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 guesser 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 organization 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

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 user's 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 be 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.

14.6 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.

14.7 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 user in
        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 Permanently](#) 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.

14.8 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 user in
        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.

14.9 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 more 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 implemented 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.

14.10 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 use 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.

14.11 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 tokens. When using bcrypt, every hash has a unique salt added to it. This means we wouldn't be able to hash a session token and then search our database for it, as it would have a different salt value. This helps prevent rainbow table attacks, but we only need to worry about those with passwords, not session tokens.

If we are getting technical, we could make this work by storing the user's ID in a cookie and then using the ID to look up a users hashed session token. We could then compare the plaintext token with the hashed token.

```
bcrypt.CompareHashAndPassword(hashedToken, plaintextToken)
```

We will not be doing this as it would be slow, add unnecessary complexity, and make it more challenging to support multiple sessions per user in the future. In short, it doesn't make sense to do extra work to use a hashing function that isn't well suited to our needs.

Our session tokens are:

1. Generated randomly.
2. Replaced every login.
3. Not reused across websites like passwords may be.

Given this, there area few options that we could explore, 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 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.

14.12 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 attempt 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.

14.13 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 create 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.

14.14 Querying Users via Session Token

We are now ready to 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.

14.15 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 Session-Service.

```
code models/session.go
```

We will name 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.

14.16 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 the 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.

14.17 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.

14.18 Session Exercises

14.18.1 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?

14.18.2 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.

14.18.3 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 two tables defined in our database - the users table, and the session table. In our sessions table we have the `user_id` column that allows us to determine which user a session belongs to.

```
CREATE TABLE sessions (
    -- ...
    user_id INT UNIQUE, -- This line defines our relationship
    -- ...
);
```

Using the `user_id` field to define what user a session belongs to **relationship** in SQL. Specifically, we are saying that each session *belongs to a user*. The relationship is singular because a session only has one `user_id` field, so it can only belong to a single user.

We can also derive the inverse relationship from the `user_id` field. That is, we could say that a user *has one session*.

The relationship is singular in this direction because the `user_id` column on

the sessions table has a **UNIQUE** clause. The unique restriction makes it impossible for two sessions to share the same user ID which ensures that a user has at most one session. If the unique restriction wasn't preset in the session table, a user could have many sessions.

In SQL databases, relationships like these are used to link various database tables together. Structuring data this way makes it easier to update data because we only need to update a single record when data changes.

This is much easier to understand with an example, so let's imagine that we are building a blogging application where we have blog posts, and various users can comment on each blog post. We might decide that we need data structured like the JSON below to render a single 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!"
    }
  ]
}
```

Our data has information about the blog post itself, as well as information about the comments and who wrote each comment.

If we were using a document store like [MongoDB](#), we might store the data in this exact format in our database. We could create something called a document, and store a blob of data like this in it.

When storing data in documents like this, it is quick and easy to query for a blog post because we only need to look up one document, and it contains everything

we need to render a blog post. These benefits do not come without trade-offs though. If a user wanted to update their `username`, we would then need to find every document where that user is present and update the document. The end result is a database structure where reading data is very quick, but it can be more challenging and slower to update data.

With SQL relationships, we could take another approach and create tables for the various pieces of data we need. We could start with a table to store each blog post, and in that table we would only store the data specific to a blog post.

```
CREATE TABLE posts (
    id SERIAL PRIMARY KEY,
    title TEXT NOT NULL,
    markdown TEXT NOT NULL
);
```

We could then create another table to store information about our users. That way we have a single place to update if a user wants to change their username or email address.

```
CREATE TABLE users (
    id SERIAL PRIMARY KEY,
    email TEXT UNIQUE NOT NULL,
    display_name TEXT NOT NULL
);
```

Finally, we could add a table for comments, and in this table we could define the relationships necessary to connect our tables. We could do this by adding both a `user_id` and a `post_id` column to our table.

```
CREATE TABLE comments (
    id SERIAL PRIMARY KEY,
    user_id INT,
    post_id INT,
    markdown TEXT NOT NULL
);
```

By adding the `user_id` field, we are saying that each comment belongs to a user. That user is the author of the blog post. The `post_id` column states that each comment belongs to a blog post, and as we saw earlier, this also means that a blog post has many comments.

With these relationships defined we could query our SQL database for a blog post, its comments, and the user who authored each comment.

```
SELECT * FROM posts
JOIN comments ON posts.id = comments.post_id
JOIN users ON users.id = comments.user_id;
```

We will learn more about how `JOIN` works later in the course, but for now all we need to know is that a `JOIN` allows us to query for all of this information in a single SQL query.

With the SQL database we can now make updates to individual pieces of data much faster and easier. If a user wants to update their `username`, we only need to update a single record in the `users` table. All of our SQL queries will then return the updated data. This also allows us to query for things we may not have planned for. For instance, we could query for all of the posts that a specific user commented on quite easily. Performing this query on a document store without planning for it is often much harder to do.

As with all things, there are also tradeoffs to using relationships instead of documents. Now when we want to query our database for everything related to a blog post, we need to retrieve data from multiple database tables which might be slightly slower than querying a single document.

Every database has tradeoffs that need to be explored and managed. SQL databases are very popular for web applications because they offer a very balanced approach. Both writing and reading data is quick and easy to do, and when a slow query does come up it can often be optimized to work quite well. SQL also gives us the flexibility to perform queries we didn't necessarily plan for.

As we add features to our application, the number of relationships in our database will expand. Users will have many galleries, and each gallery may have many images. We will also introduce other tables for things like resetting a user's forgotten password.

SQL provides several features to help us work with relationships. We can use the **JOIN** statement we saw earlier in this lesson to write a SQL query that returns related data from multiple database tables. We can also use foreign keys to ensure that we don't insert invalid relationship data. This would ensure that columns like the session's **user_id** doesn't have an invalid value. We can even use indexes to speed up our queries across multiple tables.

The rest of this section will be spent looking at some of these features before we move on with building our application.

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. Without a foreign key, we could put any value we want into the **user_id** column of our sessions, but with a foreign key in place our Postgres database would ensure that we only insert values that correspond to valid users in our database.

Below are two of the ways we could define our foreign key when creating the sessions table. Both will result in the same database setup.

```
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  
);
```

In the code above we are telling our database that the `user_id` field references the `id` column in the `users` table. In other words, we are saying that any values stored in the `user_id` column must be a valid user's ID. If we attempted to insert an invalid value, our database would return an error. Below is an example.

```
INSERT INTO sessions (user_id, token_hash) VALUES (123, 'abc');
```

If there isn't a user with the ID 123, this would result in an error like the one below.

```
ERROR: insert or update on table "sessions" violates foreign  
key constraint "sessions_user_id_fkey"  
DETAIL: Key (user_id)=(123) is not present in table "users".
```

The foreign key also ensures that we cannot delete a user without first handling any sessions linked to that user. For instance, if we had a session with the `user_id` value set to 1, the following code would result in an error because we cannot delete the user without first handling the session.

```
DELETE FROM users WHERE id=1;
```

This gives us the following output if the user has a session.

```
ERROR: update or delete on table "users" violates foreign
key constraint "sessions_user_id_fkey" on table "sessions"
DETAIL: Key (id)=(1) is still referenced from table
"sessions".
```

If we were to delete all of the sessions where the `user_id` is equal to `1`, then we would be able to delete our user without any errors.

We can use SQL relationships without adding foreign keys to our database, but adding a foreign key is a nice way to ensure data integrity.

15.2.1 Foreign Keys with Existing Tables

We saw how to add a foreign key to a new database table. We can also add a foreign key to an existing table using the `ALTER TABLE` command.

```
ALTER TABLE sessions
ADD CONSTRAINT sessions_user_id_fkey FOREIGN KEY (user_id) REFERENCES users (id);
```

The last half of this is the same as what we did before, but in this case we need to provide a name for the constraint. The naming schema that Postgres was using when creating a table is:

- Start with the table name - `sessions`
- Next, add the column with the foreign key constraint - `user_id`
- Lastly, add the `fkey` suffix to signify that it is a foreign key constraint.

Putting this all together we get `sessions_user_id_fkey`. We could use a different name if we wanted, but it is usually a good idea to try to match the pattern used throughout the entire database.

15.2.2 Updating our code

First, we need to reset our database. We can do this in a number of ways, but the simplest is likely to use docker to reset everything.

```
docker compose down  
docker compose up -d
```

Next, open the **sessions.sql** file where we defined our sessions table.

```
code models/sql/sessions.sql
```

Update it with the following SQL that includes a foreign key constraint.

```
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 to our database as we proceed, but if we wanted to run this SQL now we could by connecting to the database with docker compose and pasting in the contents the SQL files. We would need to ensure that we created the **users** table first, otherwise the database would give us an error when creating the sessions table.

```
docker compose exec -it db psql -U baloo -d lenslocked
```

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

We saw in the last lesson that once we add a foreign key, we cannot delete a database record unless any foreign key references to it are first cleaned up. In many cases, what this means is deleting all of the associated data. For instance, if we delete a user, we likely want to delete any sessions associated with that user, as they will no longer be valid.

In SQL we can automate this process with the **ON DELETE CASCADE** option. This can be added when creating the foreign key, and it tells our database that we want to delete this resource when the resource it references is deleted.

```
CREATE TABLE sessions (
    id SERIAL PRIMARY KEY,
    user_id INT UNIQUE REFERENCES users (id) ON DELETE CASCADE,
    token_hash TEXT UNIQUE NOT NULL
);
```

In the above example, the **ON DELETE** portion is telling our database that if a user is deleted, we also want any sessions associated with that user to also be deleted. If we were to run a **DELETE** SQL query on a user with a session, we would no longer see an error from the foreign key and instead we would see that the related session is automatically, and permanently, deleted.

We are going to add the **ON DELETE** clause to our sessions table to ensure that all sessions associated with a user are deleted when that user is deleted. We will first make the change in the **sessions.sql** file.

```
code models/sql/sessions.sql
```

We are going to update the SQL in the file to match the code below.

```
CREATE TABLE sessions (
    id SERIAL PRIMARY KEY,
    user_id INT UNIQUE REFERENCES users (id) ON DELETE CASCADE,
    token_hash TEXT UNIQUE NOT NULL
);
```

Once the code has been updated we can update our database by using docker.

```
# Drop our database
docker compose down
# Recreate it
docker compose up -d
# Connect to the database with psql
docker compose exec -it db psql -U baloo -d lenslocked
```

From here, paste the SQL necessary to create the users table and the sessions table and execute each **CREATE TABLE**.

```
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
);
```

If we wanted to test this change, we would first start up our application and create a user.

```
go run main.go
# Then create a user at localhost:3000/signup
```

We could then connect to our SQL database and delete the newly created user.

```
DELETE FROM users WHERE id=1;
```

Now instead of seeing an error about the user having a session, the session will be deleted along with the user. We can verify this by querying for sessions.

```
SELECT * FROM sessions;
```

id	user_id	token_hash
(0 rows)		

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 seen how to define a relationship more explicitly in SQL using a foreign key, let's explore how to query relational data using [JOIN](#).

Aside 15.1. Preparing your database

If you haven't already, be sure to reset your database and make sure it is up-to-date with changes we have been making. Later in the course we will learn how to simplify this process of migrating our database, but for now we need to do it manually. Once the database is setup correctly, start the application up and create a few new user accounts so you have data to query. At least three would be ideal.

We need to connect to our database with a tool that allows us to run SQL queries. We can do this with adminer or using psql via Docker.

```
# Connect to the database with psql
docker compose exec -it db psql -U baloo -d lenslocked
```

Once connected to the database, execute the following 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 values 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 take records from two or more database tables and to match them based on the given criteria. In the query we just ran, the matching criteria is that the **id** of a record in the **users** table needs to match the **user_id** from the **sessions** table.

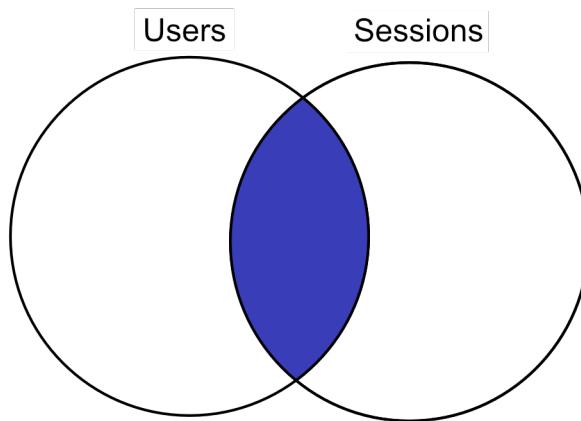
If a user has no sessions, the user won’t show up with this query. We can delete a session and then re-run the query to verify this is true.

Similarly, if a session points to an invalid user_id or had no user_id, it would not be present with this query. That isn’t currently possible with our foreign key setup, but if we removed the foreign key we could test it as well.

When we use **JOIN** we are using what is called an **INNER JOIN**. We could write the following SQL query and it would return the exact same results.

```
SELECT * FROM users
INNER JOIN sessions ON users.id = sessions.user_id;
```

SQL supports several ways to join tables. An inner join finds records where the two tables overlap.



We will cover other types of JOIN operations in the next lesson.

15.5 Left, Right, and Full Outer Join

As noted in the previous lesson, a **JOIN** in SQL is really an **INNER JOIN**. This means that it only returns records with a valid entry from both tables. In our database of users and sessions, an inner join will only return users with a valid session associated with them, and sessions with a valid user.

There are times when we may want to include records in our results, even if a related record doesn't exist. For instances, we might want to query for every

user regardless of whether or not they have a session, but we still want the session information if it is available.

We can achieve this using a **LEFT JOIN**.

```
SELECT * FROM users
    LEFT JOIN sessions ON users.id = sessions.user_id;
```

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. In this SQL code we are querying the **users** table, and we are joining it with the **sessions** table. As a result, we will receive every user in the results, and if any of those users have a session, we will receive that information as well. We will NOT receive sessions with an invalid user ID in our results.

When we run a join and a record returned doesn't have an associated record, we may see null or empty values for those fields. Keep this in mind when parsing results from a join in your Go code.

A **RIGHT JOIN** is basically the same, but reversed. Right joins will return all items from the joined table, even if they do not have an associated record in the table we are selecting from.

```
SELECT * FROM users
    RIGHT JOIN sessions ON users.id = sessions.user_id;
```

Lastly, we can do a **FULL OUTER JOIN** which will return every record from both tables, regardless of whether there is a related record.

```
SELECT * FROM users
    FULL OUTER JOIN sessions ON users.id = sessions.user_id;
```

The results from the outer join query might include:

- Users with a session
- Users without a session
- Sessions without a user

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 cases, we can filter our results with **WHERE**, although we may need to specify the table for each field we are filtering with.

```
SELECT * FROM users
  FULL OUTER JOIN sessions ON users.id = sessions.user_id
 WHERE users.id = 2;
```

As we progress through the course we will begin using JOIN and other SQL features to write the queries we need. At every step we will discuss why each query is being used in more detail, so there is no need to master SQL joins at this point. It is only necessary to know 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 add clauses to a JOIN to limit what data is being returned by the query. Take the following query as an example.

```
SELECT * FROM users
  JOIN sessions ON users.id = sessions.user_id;
```

We can specify the fields we want just like we did before, but there is a caveat. Now that we are getting results from two tables, it is best to include the table name so it is clear which table we want the field from.

```
SELECT users.id,
       users.email,
       users.password_hash,
       sessions.id
  FROM users
 JOIN sessions ON users.id = sessions.user_id;
```

We can also map fields to a custom name in our query result using `as`. This can be helpful when two tables have the same field - like the ID in both our users and sessions table.

```
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.

<code>id</code>	<code>email</code>	<code>session_id</code>
2	<code>jon@calhoun.io</code>	2
4	<code>jim@dundermifflin.com</code>	4
5	<code>pam@dundermifflin.com</code>	5

When placing the data into a Go struct, this typically doesn't matter because we will scan the data in a very specific order and know what each field is, but it can make it easier when working directly with SQL or when working with more complex queries.

The last thing to note before we start coding is that we don't need to return data from every table when performing a join. There are often cases where what we will want to query for a specific item in one table, then query for related records to that item.

Let's look at this using our application code, as it will make it much clearer, and will improve our code. Open the session service source file.

```
code models/session.go
```

If we look at the `User` method, we currently query for a session with a given token hash, and we retrieve the `user_id` from that session. We then use the `user_id` to query for that specific user. The end result is two SQL queries, which isn't awful, but might not be the fastest operation if our database is located on another server.

Instead of performing two 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
}
```

Start the web server and try signing in to verify that the code still works with the new query.

Source Code

- [Source Code](#) - see the final source code for this lesson.
- [Diff](#) - see the changes made in this lesson.

15.7 SQL Indexes

Imagine that we had a list of email address and we needed to find a specific one. How would we go about doing it?

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	...

If the list is small like the example above it is pretty trivial, but imagine that we had millions of users printed out on pieces of paper (so we can't use the computer to search for us).

The only way to find a specific user would be to look at every user, one at a time. We might be able to ask a friend to help take half the pages (like a computer using multiple processors), but we would still need to painstakingly looking at every page until we locate the user.

On the other hand, if the list of email addresses was sorted alphabetically, this becomes much easier. We could pick an entry in the middle of the list and ask ourselves, "What letter does this email address start with?" This would help us determine whether the user we are looking for is earlier or later in the list. We could then repeat this process until we find the email address. With practice, we could find a single email address in a large sorted list pretty quickly.

Computers (and SQL), perform searches in a similar manner. If data is not sorted, they must look at every record. They can do this very quickly, but it still might require looking at millions of items.

By sorting the data, we make it possible to perform the same queries much faster. As an example, a computer can search a million sorted email addresses with roughly 21 checks. That is significantly faster than looking at millions of records!

While searching is faster with sorted data, the downside is that we need to maintain any sorted lists in order for them to remain useful. That means if a user were to sign up, or another user updated their email address, we would need to update our sorted list of email addresses. It also means that if we wanted to sort our users by both their email address and their ID, we would need to store some of the user information in multiple lists.

SQL indexes are how we tell our database about data that we want to keep sorted. We typically do this for data that we query frequently, as it will speed up those queries.

To visualize this, let's go back to the spreadsheet of users we saw earlier.

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	...

In SQL, tables are sorted by their primary key (ID in our case) by default, as it is assumed we will need to query records by that value frequently. The spreadsheet above is also sorted by the user's ID.

Adding a SQL index can be similar to copying the **id** and **email** columns from our current sheet into a new spreadsheet, and then sorting that sheet 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	

With the second sheet we can now search for a user by their email address very quickly, then we can use their ID to find them in the original sheet. This

requires more searching than storing all of a user's information in both sheets, but it also makes it much easier for us to update a user's `password_hash` or their `display_name`, as we won't need to update that information on both sheets.

As we learned before, the downside is that if a user updates their email address, we will need to update both sheets, and we will need to figure out where the new email address should be placed in our sorted sheet.

The end result is that we can perform queries by email address faster, but updates to a user's email address take a bit more time, and we have to have storage space for two sheets now.

While this isn't a perfect analogy to how SQL indexes work, it is close enough to help illustrate the point. SQL indexes improve the speed of specific queries, but they can also increase the cost of updates and the storage space required for our database.

Due to the the downsides, it is important not to add indexes too frequently to a database. Too many indexes and updates may become very slow, but too few indexes and queries may be slow. We need to find a nice balance between the too many and too few indexes.

Common cases where indexes might be needed include:

- **Columns with a `UNIQUE` or `PRIMARY KEY` constraint** always need an index, and Postgres handles this for us when we add these constraints.
- **Fields that are frequently queried** will often need an index. For instance, we look up a user's email address every time they log in. We also query for the `token_hash` in our sessions table every time a user visits our page with a session cookie. These fields are both good candidates for an index.
- **Columns used frequently for joins** may also need an index. An application that frequently performs a JOIN on users and their comments might

find that the `user_id` column in a comments table should be indexed. Keep in mind that many foreign keys may not fall into this category, so it is probably best to wait to add an index until there is evidence that it would improve performance.

Another factor to consider is the size of the database table. If we are working with a small table, indexes aren't as important. This is especially true if we don't expect the number of records in the table to grow significantly, but we expect frequent updates.

We can always add an index to a database, so **common advice is to wait until there is clear evidence that an index is needed before adding one**. This will help ensure that a database doesn't have an excessive amount of indexes.

Finally, it is worth noting that it is possible to create indexes that utilize multiple columns in a database table. This can be handy when we frequently perform queries based on more than one column, such as the `make` and `model` of a car. The way these work is the data is first stored by the `make`, and then all records with the same `make` are sorted by their `model`. As a result, the order of the columns used in these indexes does matter.

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);
```

This creates an index with the name `sessions_token_hash_idx`, and it is sorted by the `token_hash` column of the `sessions` table.

In some SQL variants you can create an index while creating a table. Below is an example, but this syntax 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, in Postgres 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 will first sort the dogs by their breed first, and then any dogs with a matching breed will be sorted by their age next. 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

ON CONFLICT is a Postgres-specific way of telling our database what to do if it encounters a conflict while performing an insert. A common use case for this is to try to insert a record if it doesn't exist, but if it already exists we can instead update it.

Aside 15.2. What about MySQL, SQLite, etc?

Other SQL variants typically support this behavior, but the syntax will vary. In MySQL I believe it is ON DUPLICATE KEY, and I think it is something else in SQLite. If you do not plan to use Postgres, I recommend reading the docs of the database you are using to find the syntax. Searching for the term “upsert” along with your database variant might also help you find results.

In `models/session.go` we have quite a bit of logic to handle checking if a user has a session. Let's open it up and review the **Create** method.

```
code models/session.go
```

In the **Create** method, we first attempt to update a session. If this results in an error stating that the record doesn't exist, we then attempt to insert the record into the database. This is frequently referred to as an “upsert”.

In our current code, when this happens we have to communicate with our SQL database two separate times. Once to attempt the create, and then again when

we perform the update. Alternatively, we can use `ON CONFLICT` in Postgres to write a query that handles both of these cases for us in a single query.

```
INSERT INTO sessions (user_id, token_hash)
VALUES (1, 'xyz-456') ON CONFLICT (user_id) DO
UPDATE
SET token_hash = 'xyz-456';
```

With this query we are saying that we want to insert a new session into the database, but if there is a conflict with the `user_id` - meaning that user already has a session - we want to instead update the user's existing session and set its `token_hash` to the new value.

Let's take this and apply it to our code.

```
code models/session.go
```

Update the Create method with the following code.

```
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
}
```

We now only perform one SQL query regardless of whether the user has a session, and our SQL returns the ID of the session after updating it.

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 that define what tables are in our database:

- `users.sql`
- `sessions.sql`

These files define our database structure, commonly referred to as the database schema.

At the moment we are manually managing our database schema. When we want to add a new table or make any other database changes, we use a tool like `psql` to connect to our database and we manually type in some SQL.

This isn't uncommon when designing and prototyping a project. At this stage we are unlikely to have our application in production, so even if we make a mistake we can delete our local database and recreate it.

Long term we will want a consistent and reusable way of setting up our database. Ideally one that will ensure that every developer, and our production environment, all end up with the same database schema. Schema migrations are a common solution to this problem.

Put simply, schema migrations are series of ordered operations that we want to perform on our database to eventually get it into the correct state. A migration might create multiple tables, add a new column to a table, or it might remove unused columns. A migration can perform pretty much any SQL query to alter the database schema. The important key is that migrations are each intended to be run once, and in a specific order.

Looking at our current SQL files, our migration files might end up being:

1. create_users.sql
2. create_sessions.sql

If we were to run these migrations in order, we would end up with the database that we are currently using.

In order to ensure this consistency, schema migrations generally shouldn't be altered once they are added to a project. We can make changes while creating the migration locally, but once we commit it to the project we should assume that other developers and possibly our production environment have run the migration, so any future changes to the file won't be applied to their database.

Instead, we would create a new migration that has the changes we desire. If we realized that our users table needed a username column, we might create a migration named **add_username.sql** with the following SQL:

```
-- Add username field to the users table
ALTER TABLE users ADD username TEXT;
```

This would allow us to alter the users table without needing to change the committed `create_users.sql` migration.

While it is preferable to run every migration once and in order, occasionally we make mistakes and need a way to undo a migration. These are commonly called rollbacks, and almost all migration tooling will provide a way to undo a schema migration.

Going back to our username example, our undo step might use the following SQL:

```
ALTER TABLE users DROP COLUMN username;
```

When creating a new migration, we could also provide this SQL as the undo step. Then later if we realized our migration wasn't correct, we could undo it and get our database back to the state before the migration ran.

While rollbacks are useful in development, where they are most valuable is in production. Imagine that we released some new code that included a schema migration, only to later realize we introduced a security bug. We can't rollback the server to an old version of our code without first undoing any changes we made to the database schema, as our previous version of the code might not work correctly with a different database. If we provided a way to undo our schema migration, we could perform a rollback on the database, and then update our server to run the old version of our code until we can fix the bug.

As we progress through this section we will be learning both about how schema migration tools work, as well as how to utilize one in our project. We will be using [pressly/goose](#), a schema migration tool written in Go. Technically we could use any migration tool, even one not written in Go, but I find it is nice to use a tool written in Go so that we can run it automatically with our code.

16.2 How Schema Migration Tools Work

Before updating our code, let's first take some time to understand how schema migration tools work.

All migration tools take a list of migration steps and run them in a predetermined order. In many cases these steps will be a series of files and the order will be alphabetical by filename. 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. In the example above, our version numbers are **001**, and **002**.

Aside 16.1. Numerical vs Alphabetical Ordering

Files, and migrations, are typically sorted alphabetically, which means that files like **10.txt** will come before **2.txt**, even though numerically they are larger. This is because alphabetical sorting would only compare the **1** with the **2** in each filename, and would determine that **1** is less than **2** alphabetically.

Due to this, it is common to see migration file versions prefixed with zeroes.

In many cases that number will be a timestamp, as this helps ensure that every migration is run in the order it was created, and it is less likely to have two files with the same version. We will discuss timestamps and versions later when using **pressly/goose**.

The version for each migration is used to track whether it has been run or not. The rest of the filename is meant to help us as developers understand what each migration does. A common way that migration tools achieve this is to create a table in the database that tracks which migrations have been run.

```
CREATE TABLE migrations (
    id SERIAL PRIMARY KEY,
    version TEXT UNIQUE
);
```

The migration tool can then insert an entry for each migration that has been applied to the database, and it will always know what migrations need run versus which have been run already.

For instance, if our migration tool were to run the migration 001-create_users.sql, the following insert would also be run to track that the migration has been completed.

```
INSERT INTO migrations (version) VALUES ('001');
```

Not all schema migration 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 already been run.

If we need to undo a migration, migration tools will run our undo code and then remove the entry from the **migrations** table. These undo steps are often provided in one of two formats:

1. As a separate file with a suffix to notate whether it is a migration, or a rollback step.
2. As part of the migration file with special comments to notate which part is the migration, and which part is the rollback step.

The common naming schema is to refer to the migration step as the **up** step, and the rollback as the **down** step. This might mean our up and down steps have the following filenames:

```
001-create_users.up.sql  
001-create_users.down.sql
```

Or if our tool uses a special comment syntax, we might have a single migration file with comments like the following:

```
-- +goose Up  
CREATE TABLE users (  
    ...  
);  
  
-- +goose Down  
DROP TABLE users;
```

Goose does the latter, but we would want to read the docs for any tool we are using to verify how it expects migrations to be formatted.

16.3 Installing pressly/goose

There are a few migrations tools written specifically for Go application. Two of the most common ones are:

- [pressly/goose](#)
- [golang-migrate/migrate](#)

We are going to use **goose** for our application, and we will start by installing it using **v3**.

```
go install github.com/pressly/goose/v3/cmd/goose@v3
```

When running this command we should see some output suggesting that Go has installed goose. We should also be able to run **goose** on our command line.

```
goose -version
# goose version:v3.7.0
```

If this does not work, check the [How to Write Go Code](#) post to ensure you have everything setup correctly for installing Go programs.

We can see a list of goose subcommands available to us by running `goose` in the terminal.

```
goose

# ... some output not shown for brevity
#
# Commands:
#   up                  Migrate the DB to the most recent version available
#   up-by-one           Migrate the DB up by 1
#   up-to VERSION       Migrate the DB to a specific VERSION
#   down               Roll back the version by 1
#   down-to VERSION    Roll back to a specific VERSION
#   redo               Re-run the latest migration
#   reset              Roll back all migrations
#   status             Dump the migration status for the current DB
#   version            Print the current version of the database
#   create NAME [sql/go] Creates new migration file with the current timestamp
#   fix                Apply sequential ordering to migrations
```

With Goose we can create migration files using their template file, run migrations, roll back migrations, check the migration status of our database, and later we will even see how we can use goose and Go's [embed](#) package to embed our migrations into our binary.

Let's try creating a migration file using `goose`. We will start by creating a directory for our migrations and changing into that directory, then we will call the `create` subcommand of `goose`.

```
mkdir migrations
cd migrations
goose create widgets sql
# 2023/09/21 12:09:04 Created new file: 20230921120904_widgets.sql
```

When we use the `create` subcommand, we must follow it with the name of our migration, and then the type of migration it is. In our case we are using a SQL migration, but `goose` also supports [Go migrations](#).

Goose's Go migrations are a bit trickier to use compared to SQL migrations, as they require us to use `goose` in our Go code, but they are one of the main reasons that I prefer `goose` over other schema migration tools.

Running `goose`'s `create` subcommand will create a migration file with a name in the format:

```
<timestamp>_<migration-name>.<migration-type>
```

Let's open up our migration file so we can look at it and change it.

```
# Open the newly created file
# * will match any timestamped file as long as it has the _widgets.sql suffix
code *_widgets.sql
```

Inside the migration file will be the boilerplate needed to create a migration. In the case of our SQL migration, there are some comments that `goose` uses to determine which parts are the migration (called `Up` in `goose`), and which parts are for rolling back a migration (`Down` in `goose`), as well as a fake query.

```
-- +goose Up
-- +goose StatementBegin
SELECT 'up SQL query';
-- +goose StatementEnd

-- +goose Down
-- +goose StatementBegin
SELECT 'down SQL query';
-- +goose StatementEnd
```

The **StatementBegin** comments also help in case we need to write more advanced queries that contain semicolons within them. We won't be writing queries like this, but since goose generates the comments for us we will retain them to avoid any potential issues, and we will write our SQL inside of the **Begin** and **End** section.

Let's update the migration file to create a widgets table in our database.

```
-- +goose Up
-- +goose StatementBegin
CREATE TABLE widgets (
    id SERIAL PRIMARY KEY,
    color TEXT
);
-- +goose StatementEnd

-- +goose Down
-- +goose StatementBegin
DROP TABLE widgets;
-- +goose StatementEnd
```

Next we can run `goose` to see the status of our database. For this to work, we need to tell `goose` how to connect to our database. We are going to be using a connection string similar to the one below.

```
host=localhost port=5432 user=baloo password=junglebook dbname=lenslocked sslmode=disable
```

The string above is using the default setup from this course, but if we wanted

to verify that we had the correct string we can use our existing code to print out the connection string our Go application is using.

```
# Change to our root app directory
cd ..
code main.go
```

Inside the `main` function, add the following print statement to print out our postgres config as a connection string.

```
func main() {
    ...
    // Find this line
    cfg := models.DefaultPostgresConfig()
    // Add this line
    fmt.Println(cfg.String())
    ...
}
```

Now we run our code to get the output.

```
# Run our code to see the connection string
go run main.go
# Stop the server (ctrl+c) after the connection string prints out
```

With our connection string in hand we can head back to the migrations directory and tell goose how to connect to our database. We will start by calling the `status` subcommand, which tells us the status of our database with regards to migrations.

```
# Change back to the migrations directory.
cd migrations

# 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
<hr/>	
Pending	-- 20230921120904_widgets.sql

Moving forward we will need to continue providing goose with the connection string, but we will be changing the subcommand that we use. The next sub-command we will look at is **up**, which will run any migrations that need to be run.

```
goose postgres \
  "host=localhost port=5432 user=baloo password=junglebook dbname=lenslocked sslmode=disable" \
  up

# 2023/09/21 12:37:15 OK  20230921120904_widgets.sql (14.83ms)
# 2023/09/21 12:37:15 goose: successfully migrated database to
# version: 20230921120904
```

Aside 16.2. Terminal Hint: Scroll Through Previous Commands

In most terminals you can press the up arrow to scroll through previously run commands. Do this until you see the `goose` command, then press **ctrl+e** or right arrow to scroll to the end of a line where you can alter the subcommand without needing to type the connection string again.

We can also roll back migrations using the **down** subcommand.

```
goose postgres \
  "host=localhost port=5432 user=baloo password=junglebook dbname=lenslocked sslmode=disable" \
  down
```

Unlike the up subcommand which may run multiple migrations, down will only roll back a single migration.

With that we have a basic idea of how to use goose. Next, let's try to rework our application to use goose for managing our database. Specifically, we want to move our users and sessions tables into migrations that goose can run for us.

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

If in the migrations directory, stay there. If not in the migrations directory, navigate there.

```
cd migrations
```

Next, we need to delete any migrations that we have, as everything created up until this point were for demonstration purposes only.

```
rm *.sql
```

Next we can head back to our app's root directory and reset our docker containers to ensure that our database is in a fresh new state.

```
cd ..  
docker compose down  
docker compose up -d
```

Great, now that our database and our migrations directory is in a clean state we can focus on adding real migrations that our application will use. To do this, we will navigate back to the **migrations** directory and create migration files using `goose`.

```
# Alternatively, the -dir flag could be used with goose, but  
# I find it easier to just navigate to the correct dir.  
cd migrations  
  
# Create the migration file  
goose create users sql  
  
# Open it to add some SQL  
code *_users.sql
```

We have already written the **Up** portion of our migration, which can be found in **users.sql** file.

```
cat ../models/sql/users.sql
```

Copy the file's contents.

```
CREATE TABLE users (  
    id SERIAL PRIMARY KEY,  
    email TEXT UNIQUE NOT NULL,  
    password_hash TEXT NOT NULL  
) ;
```

Paste the contents into our migration file that we created earlier.

```
-- +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 our users migration. In this case the opposite step is to drop the table.

```
-- +goose Down
-- +goose StatementBegin
DROP TABLE users;
-- +goose StatementEnd
```

Now we need to repeat the process for our sessions table.

```
goose create sessions sql
code *_sessions.sql
```

Update it with the following code.

```
-- +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 both files are saved, and then we are ready to run the migrations using `goose`!

```
goose postgres \
  "host=localhost port=5432 user=baloo password=junglebook dbname=lenslocked sslmode=disable" \
  up
```

This should give us output similar to the output below.

```
2023/09/21 12:55:38 OK  20230921125220_users.sql (6.78ms)
2023/09/21 12:55:38 OK  20230921125449_sessions.sql (7.86ms)
2023/09/21 12:55:38 goose: successfully migrated database to
version: 20230921125449
```

Our application's database has been migrated to the most recent version, and our schema is what our application expects. We can now start our application and test everything out.

```
cd ..
go run main.go
```

Create a user or two in order to verify that everything is working as expected.

Source Code

- [Source Code](#) - see the final source code for this lesson.
- [Diff](#) - see the changes made in this lesson.

16.5 Schema Versioning Problem

By default, `goose` creates migrations with a timestamp as the version. We can see this by looking at our current migrations.

Timestamps as versions gives us a repeatable order to run and test migrations in, and they ensure that each new migration we create will come after an existing one. They work quite well for local development, but can present challenges when merged with changes that other developers are working on.

Let's look at an example of what the schema versioning problem is, and how it might occur, by walking through a series of steps that two developers might take. *Note: Dates in the migrations are made a bit more readable with dashes (-), but the problem is the same for any date-based versioning.*

We will start off with an application that has a few existing migrations.

```
2022-01-06_first_migration.sql  
2022-01-08_some_migration.sql  <-- production schema version
```

From there, our developers start working on two new features that each require a database schema change.

- 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**

Both of the developers are working on their own changes locally, so neither knows about or has the other's migration file in their source code. This is fairly common when using source control like **git**, as the changes haven't been merged together yet.

On Jan 13, Bob finishes and his feature and merges it into source control. The application gets deployed and his migration is run. The production database now has a schema version of **2022-01-12**

```
2022-01-06_first_migration.sql  
2022-01-08_some_migration.sql  
2022-01-12_bob.sql      <-- production schema version
```

A day later, Jon finishes his changes and merges his code into source control. This gives us a list of SQL migrations with the following ordering:

```
2022-01-06_first_migration.sql  
2022-01-08_some_migration.sql  
2022-01-11_jon.sql      <-- jon's schema version  
2022-01-12_bob.sql      <-- production schema version
```

Jon may notice that Bob had a migration as well, so he might run all of the migrations locally to verify that they work together. This will update his schema version to match production, but the production database has never had Jon's migration applied to it because Jon's migration version is BEHIND the current version of the production database.

Now when Jon's code is pushed to production, one of three things may happen.

1. The migration tool will not see any migrations with a newer version number, so no migrations will be run.
2. The migration tool will see the new migration, but will note it has an older version number and will return an error.
3. The migration tool will see the new migration, and will run the missing migration.

Goose falls into case (2) - it will notice the new migration and return an error.

In the first case we could end up with a migration that never gets run in production, and it could be a nightmare to debug.

In the second case we will know about the error and we will need to update Jon's migration to use a newer version number. While a little tedious, this isn't too terrible to do.

At first glance the third case sounds optimal, but the problem with it is that our production database will now be running schema migrations in a different order than our developers have tested, and a different order than a fresh new database will execute the migrations. In our example, Jon's migration will always be run before Bob's migration if we reset our database and run the migrations, but this is not what happens in production. This can lead to bugs where one ordering works fine, but the other doesn't, and it can be frustrating to debug and fix an error when it occurs on a production database. (*Note: This is why many applications have a staging deploy - to catch issues like this.*)

This is the schema versioning problem, and we want to avoid it. How can we achieve that?

The first approach is to update timestamps in migrations when these situations occur. This can be done during a code review process, or just before merging code into the main branch. In the example above, we might rename Jon's migration to use a new date like below.

```
2022-01-06_first_migration.sql
2022-01-08_some_migration.sql
2022-01-12_bob.sql <-- current schema version
2022-01-14_jon.sql
```

Goose will give us an error if we miss a case like this, so we have a reasonable safety net to work with.

Another similar option is to use timestamp versions during development, and to rename migrations with incremental version numbers just before merging them into the main branch. Eg prefixes like **0001**, **0002**, **0003**, and so on are all that end up in our source control, but **2022-01-14** can be used in local development until it is about to be merged.

Using this approach makes version numbers more readable, and duplicates become easier to spot. It also has the added benefit of helping us catch or avoid the schema versioning problem before code is deployed to production.

Let's go back to our example where Jon has a migration change, and Bob's migration is already in production. If we were using incremental versions, the versions in our main branch would be:

```
00001_first_migration.sql  
00002_some_migration.sql  
00003_bob.sql           <-- production schema version
```

Meanwhile, Jon's local branch will look like:

```
00001_first_migration.sql  
00002_some_migration.sql  
2022-01-11_jon.sql        <-- jon's schema version
```

When Jon goes to merge his code, he can either:

1. Finalize his version number, then try to merge that code into the main branch of source control.
2. Merge changes from the main branch into his own code and then finalize his migration version number.

If Jon uses the first approach, his migration will get the version **00003** and will be a duplicate version number since Bob already committed something with that version number. This would be pretty easy to catch and fix during the merge process.

Alternatively, if Jon uses the second approach his local migrations would look like the list below.

```
00001_first_migration.sql  
00002_some_migration.sql  
00003_bob.sql  
2022-01-11_jon.sql
```

At this time Jon's migration comes **after** Bob's, which is what we want, and if we were to give it an incremental version number it would be assigned **00004**.

```
00001_first_migration.sql  
00002_some_migration.sql  
00003_bob.sql  
00004_jon.sql
```

Now there are caveats to using this approach. Developers need to follow the procedure properly, and developers will also need to rollback their migrations while it has a timestamp version, merge in code from the main branch, fix their version to an incremental one, and finally run their migration again to verify it works. With that said, almost all of these steps should be performed or verified in any other process, otherwise it could result in issues when attempting to run a migration in production.

Goose provides some tooling to help with this process. Most notably, the **fix** subcommand.

```
goose fix
```

When running the fix subcommand, goose will take all of our migration files that have timestamp versions and will rename them with incremental version numbers like **00001**, **00002**, and so on.

As noted earlier, timestamp migrations need to be rolled back BEFORE running the fix subcommand, otherwise our local database will think it is migrated to a much larger version than it is actually on. This is because a timestamp

version is a large number like 20230921125449, meanwhile the fixed versions are small numbers like 000012.

Aside 16.3. 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. Rollback migrations with goose down or reset.  
goose down  
# 2. Pull changes that other developers have pushed to our team's repo.  
#   This assumes that the origin remote branch is configured to something  
#   like GitHub where I can pull code other devs have submitted.  
git pull origin <main branch> --rebase  
# 3. Run goose fix to rename my migrations with the correct versions.  
goose fix  
# 4. Run the migrations to verify they work.  
goose up  
# 5. Test everything  
go test ./...  
# 6. Commit and merge everything to the main branch my team uses.  
#   This might be via a GitHub pull request, or something else.
```

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. The main thing is to be informed about the tradeoffs of your approach and to keep an eye out for issues.

Let's fix the ordering for our migrations. We will start by resetting our database, which will undo all of our migrations.

```
# Navigate to the migrations directory if not already there
cd migrations
goose postgres \
  "host=localhost port=5432 user=baloo password=junglebook dbname=lenslocked sslmode=disable" \
  reset
```

If we check the status of our migrations, they should all be pending at this point. That means we can fix the version numbers.

```
goose fix

# Output similar to:
#
# 2023/09/21 13:56:04 RENAMED 20230921125220_users.sql => 00001_users.sql
# 2023/09/21 13:56:04 RENAMED 20230921125449_sessions.sql => 00002_sessions.sql
```

Finally, we are ready to run our migrations and commit our source code to any source control we may be using.

```
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.

16.6 Running Goose with Go

In this lesson we are going to look at how to run our SQL schema migrations using our Go code. More specifically, we are going to import a `goose` library into our code that will allow us to perform operations that the `goose` CLI allowed us to use - up, reset, etc.

There are a number of reasons why we might want to run migrations via our Go code. One of the most common reasons is to ensure that migrations get run whenever our application starts up. Running migrations when our app starts helps ensure that we don't have a bug that is solely because we forgot to run database migrations.

Another reason is to remove the need for others to install and use the `Goose` CLI. Specifically, we could import the `goose` library, then build subcommands into our own application for running migrations. Eg we might build out our app so it creates a CLI named `lenslocked`, and we use it like:

```
lenslocked up # runs migrations
lenslocked down # undos a migration
lenslocked start # starts the web server
```

We could then add additional commands down the road as they became useful. We won't be doing this in the course, but it could be helpful for many applications.

Finally, importing the `goose` library allows us to use migrations written in Go. This is once again something we won't be doing in the course, but additional information can be found in the [goose docs](#).

Now let's get `goose` imported into our code. We first need to `go get` the library so it is added to our application.

Note: Installing `goose` is different and didn't add it to our `go.mod` file.

```
go get github.com/pressly/goose/v3
go mod tidy
```

Aside 16.4. Why do we need to use "go mod tidy"?

In a nutshell, running the `tidy` subcommand will clean 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 retrieve 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 `Migrate` functions below the `Open` function in the source file.

```
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
}
```

In this code we first tell `goose` that we will be connecting to a Postgres database. After that we call the `Up` function which will cause `goose` to run all of the migrations in a given directory, much like using the `goose` CLI would.

After adding this code we might find our code giving us an error about `goose`. This is because `go mod tidy` likely removed `goose` from our `go.mod` file since it didn't see any code in our application that uses `goose`. When this happens, auto-imports may incorrectly import an older version of `goose`. To fix this, head to the imports in the `postgres.go` file and update the `goose` import to use `v3`. This will

```
import (
    "database/sql"
    "fmt"

    _ "github.com/jackc/pgx/v4/stdlib"
    "github.com/pressly/goose/v3"
)
```

After we have the import corrected, we can use the `tidy` subcommand to update our `go.mod` file and download any other dependencies our app may need.

```
# Note: The compat flag isn't necessary if your go.mod file
# uses a recent version of Go.
go mod tidy -compat=1.17
```

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 16.5. 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. We can verify this by resetting our migrations and then starting our Go application. It should run the migrations as part of the startup process.

```
# Change to the migrations dir for running Goose
cd migrations

# Reset migrations
goose postgres \
    "host=localhost port=5432 user=baloo password=junglebook dbname=lenslocked sslmode=disable" \
    reset
# Check the status of migrations
goose postgres \
    "host=localhost port=5432 user=baloo password=junglebook dbname=lenslocked sslmode=disable" \
    status

# Change back to our app's main dir & run our code
cd ..
go run main.go
```

We should see output like below signifying that Goose was run as part of our application starting up, and that two migrations were performed.

```
2023/09/09 16:15:51 OK  00001_users.sql (23.57ms)
2023/09/09 16:15:51 OK  00002_sessions.sql (8.63ms)
2023/09/09 16:15:51 goose: successfully migrated database to version: 2
Starting the server on :3000...
```

Source Code

- [Source Code](#) - see the final source code for this lesson.
- [Diff](#) - see the changes made in this lesson.

16.7 Embedding Migrations

The way our application is currently setup, our SQL migration files are not read until we run our Go application. This means that if we ran our Go code somewhere where the migration files were not present, we would either receive an error, or it might not detect any migration files and assume that there aren't any to run. This last case is the most troubling one, as it could mean our application isn't running any migrations when we are expecting several to be run.

An alternative approach to reading the migration files when our application starts is to instead embed them into the application's binary. By doing this, we ensure that no matter where the binary is run from, it will have access to the SQL migrations. This can be especially helpful for a variety of production environments.

The way we are going to achieve this is very similar to what we did with the template files in `templates/fs.go`. We will start with a function that runs migrations using an `fs.FS` filesystem.

`code models/postgres.go`

We will be adding this as a new function named `MigrateFS`, and we will use the code provided by the [Goose docs](#) to help us figure out what our code should look like.

```
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
    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
    }()
}
```

```
    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. Once our `MigrateFS` function is complete it will unset the base file system by passing in `nil` via the deferred code. This ensures that we undo any global variable changes we made before ending our function, and allows other migrations to run that may not use our file system.

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 a variable to store the `embed.FS`.

```
package migrations

import "embed"

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

After that we can update our code in `main.go` to use the new code.

```
code main.go
```

Find the call to `models.Migrate` and replace it with one to the new `MigrateFS` function.

```

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`. As a result, all of our `.sql` files will be at the top level directory, not nested inside of a `migrations` directory. When referring to the current directory in an FS, Goose wants us to provide a period (.) as the directory.

Restart the 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.

16.8 Go Migration Files

Note: We won't be using any code from this section in our application. It is only meant to illustrate how Go migration files work with Goose.

Goose also supports migration files written in Go code. This can be handy when we want to access other parts of our Go code while running a migration. For instance, we might want a migration to normalize a column in a database using some Go code we wrote to normalize all future items written to that column. We could do this with a Go migration.

We can create a Go migration using the `goose` CLI, but we tell it we want our migration to be of the type “`go`” rather than “`sql`”.

```
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
}
```

*Note: If your code has a **Context** added to it, this is just a new thing added in a more recent version of `Goose` and is meant to give each migration step access to a context.*

If we fill in the `upXXX` and `downXXX` functions with the logic we want to use for the migration, they will be used as part of our migration steps. Goose even retrieves our migration filename to determine the version for the migration.

When using Go migrations with Goose, we MUST run our migrations using Go code. If we attempt to use the `goose` CLI like we did in the past, we would see 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 occurs because the `goose` CLI isn't capable of building our go migration files into runnable code. We must perform the compilation step on our own and then call the `goose` library within our code.

In our case we have everything we need to run Go migrations already taken care of in our `main.go` source file, so we can run this migration by running our application.

```
go run main.go
```

Aside 16.6. Bug in Goose 3.15.0

If you happen to be using Goose **3.15.0**, a bug was introduced that causes this to not run our Go migrations if they aren't also included in the embedded file system. This was fixed in **3.15.1**, so upgrading should resolve the issue.

When we build our code, it imports the `migrations` package to access the `fs.FS` file system with our SQL migrations. This will also cause any Go files in that package - like our widgets migration - to be compiled and registered to

Goose. From there our `main.go` source file calls `goose.Up` which will run all of our migrations, including any written in Go code.

This all 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`.

What our code doesn't currently have is a way to roll back migrations written in Go code, nor can we roll these back with the goose CLI. As a result, we would likely want to add to do this if we planned to use Go migrations with Goose.

We could also look into [wrapping the Goose CLI](#) and creating our own CLI that includes our Go migrations. We won't be covering that now since we won't be using Go migration files, but it may be worth exploring outside of this course.

We won't be using any files used in this lesson, so delete any files created as while learning and be sure to reset the database before moving on.

16.9 Removing Old SQL Files

We have a few files that are no longer being used, so we are going to delete those before moving on with the course.

```
rm -rf models/sql/
```

This will delete the entire `sql` directory that we created with our original SQL tables. We do not need these since our tables are now defined in the `migrations` directory as SQL migrations. It is best to delete these to avoid any confusion where a developer might update these files and not realize they are no longer being used.

Source Code

- [Source Code](#) - see the final source code for this lesson.
- [Diff](#) - see the changes made in this lesson.

16.10 Goose SQL Exercises

16.10.1 Ex1 - Wrap the Goose CLI

Look at the Goose docs and the example where they [wrap the Goose CLI](#). Most notably, look at the `main.go` source file and examine how the code creates a program that will run much like the existing Goose CLI.

Attempt to add this to your own application by first creating a directory for the command at `cmd/goose`, and then adding a new main source file that replicates what is in the example. Be sure to import the correct SQL driver (we are using github.com/jackc/pgx/v4/stdlib), and to use the correct connection string.

As an additional step, consider using the default postgres config to automatically connect to our database without requiring that input from the CLI. What changes to the code would be necessary for this to work?

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 the current user 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 `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)
}

```

This approach works, but writing this same code in every handler would get repetitive. Instead, we want to use an approach that allows us to reuse the same code for any HTTP handler that needs access to the current user. It would also be nice to avoid querying for the same user twice in a single web request, so an approach that caches our current user somewhere would also be nice.

The most common way to achieve this is via Go's [context.Context](#) type.

Context can be used for a variety of things in Go. It can be used to set timeouts for a process, allowing us to limit each web request to a set amount of time (eg 10 seconds). We can also use a context to cancel a process when another event occurs, such as a user closing their browser window before our web server finishes processing the request. While both of these uses of a context are very useful, we won't be diving into either of these advanced cases. Instead, we will be exploring how a context can be used to store request-scoped values.

A request-scoped value is a value that is specific to the query our code is responding to. For example, every web request we receive may originate from a different IP address, so this would be a request-scoped value. The current user may also differ for each web request, making it another good fit for a context value.

Values that are not request-scoped are values that do not change based on the

request. For instance, our database connection is the same regardless of who is making a web request to our application, so this isn't a good fit for a request scoped value. Similarly, a logger for outputting debugging information likely won't change from request to request, so it is not a request-scoped value. With non-request-scoped values, there are almost always better ways to share them with our code than using context values.

As with most things, there are exceptions. If we created a unique logger with every request that includes request-specific information, such as the current users ID, or a request ID, it could be argued that this is now a request-scoped value. The main takeaway here is that we should try to provide values without using the context where possible, but from time to time the context might be the best fit.

We will elaborate on request-scoped values in a later lesson in this section. For now, the key thing to know is that we will be working towards writing code that looks up who a user is once and stores it in the context, giving the rest of our application access to the current user as needed.

Contexts can store context-specific values using the [WithValue](#) function.

```
func WithValue(parent Context, key, val any) Context
```

This function accepts an existing context, along with a value to be stored and a key for accessing it. It then returns a new context with that value stored under the given key.

Context values can then be retrieved from a context by calling the [Value](#) method and passing in the key associated with the value we want. This function is defined in the [context.Context](#) interface.

```
type Context interface {
    Deadline() (deadline time.Time, ok bool)
    Done() <-chan struct{}
    Err() error
```

```
// This is the method for retrieving values from a context
Value(key any) any
}
```

While this may feel similar to a **map** in Go, the implementation and how we will end up using it is a bit different.

Let's look at a basic example of storing and retrieving a context value, then in later lessons we can discuss best practices with context values. The entire example is available on the [Go Playground](#), but we can also code this in our **exp.go** source file.

code cmd/exp/exp.go

Once the source file is open, delete everything inside the **main** function so we have a clean slate to work from.

We first need a context to store a value in. We can create an initial context with the [Background](#) function.

```
ctx := context.Background()
```

Next, we use the **WithValue** function to create a new context with a value stored in it. In our example we will use the key "favorite-color" and the value "blue".

```
ctx = context.WithValue(ctx, "favorite-color", "blue")
```

Behind the scenes the **WithValue** function will wrap our context with a new one that has our key and value. This is similar to how we wrap errors to add additional information about what went wrong. As a result, we need to assign

the value returned from the `WithValue` function to a variable like our `ctx` variable.

Now that we have a context with a value, we can retrieve that value with the `Value` method.

```
value := ctx.Value("favorite-color")
```

While this will give us back what we stored in the context, the `Value` method has a return type of `any`. This means if we wanted to work with our value as a string we would need to convert it back into a string from the `any` type. We will discuss doing this later.

Aside 17.1. `any` vs `interface`

If you are familiar with Go's empty interface (`interface{}`), `any` is the same thing. In both cases we are saying that the value can be any type, because we are saying that we don't require any specific methods from the type. Since any type can satisfy those requirements (or lack thereof), any type can match both `interface{}` and `any`.

A complete, runnable program is shown below. It will create a context, store a value in it, retrieve that value, then print it to the terminal.

```
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)
}
```

The example code we just wrote is intended to show the basics of using context values. It doesn't follow best practices, nor does it do anything too interesting or useful with the value. We will learn more about these things as we progress through this section and start using the context package in our application.

For now the key takeaway is that we can store and retrieve values in a context, which means that as long as our code has access to a context, we can store a user in the context and provide it to the rest of our code. Every web request has a context associated with it, so we will be able to write code to lookup the current user, assign it as a context value, and then our HTTP handlers will have access to the current user.

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

We are going to continue updating the `exp.go` source file as we experiment a bit with the context package, but feel free to code along in the Go playground instead.

Aside 17.2. Warning - Complex Topic

In this lesson we discuss a somewhat complex topic that might be hard to grasp at first. It will likely make more sense as you become more experienced with Go, but at first it can feel a bit daunting.

I am including this information because I believe it is important to explain why we are doing things the way we are, but in reality you can very likely proceed and build your application without entirely understanding why or how the key type works as long as you follow the same patterns we use in the course.

So in short, attempt to understand the material, but don't let it stop you from proceeding if you are bit confused.

In our first example of using context values, our key was a string with the value "favorite-color".

```
ctx = context.WithValue(ctx, "favorite-color", "blue")
```

Keys for context values can be of any type - int, string, or even a custom type. Below is an example where a new `ctxKey` type is created and then used as a context 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 the `ctxKey` type might appear to be the same as a string, it is actually a different type. Both the type and the value of a key must match in order to retrieve a value with a given key.

In other words, the following two keys will be treated uniquely by the context package when storing and retrieving values.

```
var a ctxKey = "favorite-color"
var b string = "favorite-color"
```

While any type can be used as a context value key, it is common practice to avoid using types like string or int to avoid the risk of a context value being overwritten by other code in an application. For example, we might set our “favorite-color” to blue, and code in another package could use that same key and set a different value.

```
// 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", 0xFF0000)

// Oh no, this returns "red"!
value := ctx.Value("favorite-color")
```

In this example we also see that the value can be set to something with a completely different type, which could really cause an issue in our code.

In reality this doesn’t happen frequently, especially with a key like “favorite-color”, but it can happen. If we are using more common keys like “ID”, the odds of it happening increase even further.

If we want to avoid the chances of a key conflict occurring, we can use a unexported type for our context key. This helps ensure that no code outside of our package can alter our key/value pairs in a context. 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", 0xFF0000)

    // 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)
}
```

While our two values appear to be stored with the same key, the types will differ so both will still be stored and accessible within the context. When we run this code we will get the output:

```
blue
16711680
```

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 the key, we ensure that only code inside of the same package can overwrite or access our context values.

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

We are going to continue updating the `exp.go` source file as we experiment a bit with the context package, but feel free to code along in the Go playground instead.

Context values are stored with the type `any`. This allows us to store any data type - string, int, or a struct - inside the context. The downside is that when we retrieve the data from our context it has the `any` type which means we cannot call any methods or access any specific fields from that type. To do this, we need to convert it back to its original type.

Aside 17.3. What about generics?

There have been a few discussions around using generics to improve context values, but so far no major changes have been made to the standard library. If the context package is ever updated to include generics support I'll update this lesson accordingly.

In Go we can convert a value from the `any` type to a specific type using a [type assertion](#).

```
var a any = "hello"

// ok tells us if the assertion was successful or not
// s has the value of a in the new type - string - if this was successful
s, ok := a.(string)
fmt.Println(s, ok)
```

Similar to accessing a value in a map, a type assertion returns an optional second value that is a boolean. This represents whether or not the type assertion succeeded. Unlike a map, if we omit this value and the assertion fails, our code will panic.

```
var a any = "hello"

i := a.(int) // this will panic
```

As a result, it is a best practice to capture the second value and verify the type assertion succeeded. *Note: Personally, I feel that there are exceptions to this, but without really understanding the tradeoffs I do not recommend omitting the second value.*

We can utilize a type assertion to turn our context values back into their original type.

```
ctx := context.Background()
ctx = contextWithValue(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!")
```

In this code we are storing the string **blue** in our context. We then retrieve the value, but it has a type of **any**. We use a type assertion to convert it back into a string and verify that the assertion worked as expected before printing out the value as a string.

When printing out the value, converting it back into a string isn't necessary or that useful, but if we wanted to verify that the context value started with a

specific letter we would want to work with a variable that has the `string` type. We can only do that if we have the context value in the correct type.

```
value := ctx.Value(favoriteColorKey)

strValue, ok := value.(string)
if !ok {
    fmt.Println("not a string")
} else {
    fmt.Println(strings.HasPrefix(strValue, "b"))
}
```

In our code we are going to want to store a `models.User` in our context that represents the current user. We will want this to have the correct type so that we can access a user's ID and other attributes as needed. We will achieve this by using a type assertion like we saw here.

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

Our next step is to utilize what we learned about type assertions and context values to write the code for storing and retrieving users via a context. We will add this code in a new package named `context`, and use a source file named `users.go`.

```
mkdir context
code context/users.go
```

We can start with the key we will be using. This will use an unexported type like we discussed in a previous lesson.

```
package context

type key string

const (
    userKey key = "user"
)
```

Next we add a function that allows us to store a user into a context. We are going to provide a function for this so that we do not need to export the key used for our context value, and instead the only way to update the value is by calling the method we provide. While this doesn't give us perfect type safety, it gets us pretty close.

```
func WithUser(ctx context.Context, user *models.User) context.Context {
    return context.WithValue(ctx, userKey, user)
}
```

Now a user can be added to a context in a manner similar to what we saw with the **WithValue** function, except our custom function requires the value to have the type ***models.User**.

The final step is to write a function to help us retrieve a user from a context. We are going to do this rather than calling a context's **Value** method to avoid exporting the key used for this value, and also to ensure the value gets converted into the correct type.

```
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,
```

```
// so it is important to review code changes in this package.
    return nil
}
return user
}
```

With our context package created we are going to update our `exp.go` source file to test the new code.

code cmd/exp/exp.go

In this source file we are going to start by creating a new context and a user to store in that context.

```
package main

import (
    "context"

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

func main() {
    ctx := context.Background()

    user := models.User{
        Email: "jon@calhoun.io",
    }
}
```

Next we want to call the `WithUser` function from our custom context package, but if we try to import this package we will get an error similar to the one below.

```
cmd/exp/exp.go:6:2: context redeclared in this block
cmd/exp/exp.go:4:2: other declaration of context
```

This occurs because we are importing two packages with the same name. We are going to resolve this by telling our imports to refer to the standard library's context package as `stdctx` for this particular source file.

```
package main

import (
    stdctx "context"

    "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)
}
```

Finally, we want to retrieve the user from the context and look at the value.

```
retrievedUser := context.User(ctx)
fmt.Println(retrievedUser.Email)
```

Now if we run our `exp.go` source file we should see “`jon@calhoun.io`” being printed out using the user that was stored in the context.

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

The code in this lesson will break our current user page until we add the code in the next lesson.

Now that we are prepared to store and retrieve users with a context, we need to determine how our application will get access to a context. Luckily, every web request already has a context associated with it that we can utilize.

We access the context by calling the `Request.Context` method. We can 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. This will ensure that the user gets set on the context before any HTTP handler runs.
2. We are going to update our http handler functions to read the current user from the context rather than querying the database. This will avoid any duplicate database queries, and will simplify the code in our handlers.

We will start with the second step. Specifically, we are going to update the `CurrentUser` handler to read the current user from the request context.

In the next lesson we will create the middleware that sets the current user on the context, but until then the code from this lesson will break the current user handler as well as the sign in process until we finish the next lesson.

Open the users controller source file.

```
code controllers/users.go
```

Find the `CurrentUser` function. We will be updating this function to look for a user via the context rather than using the cookie and doing a database query. The resulting code is shown below.

```
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)
}
```

We start by retrieving the context from the request with `r.Context()`. We then pass that context into the `User` function from our context package, which will return the current user value from the context. It is currently going to be nil all the time, but we will address that in the next lesson.

Once we have the user we check to see if the user is nil or not. If the user is nil, it means no user is signed in and we need to redirect them back to the sign in page. Otherwise we can print out the current user's email address.

The code is now much simpler in large part because we are offloading the logic to read a cookie and query the database for the current user. We haven't written this middleware yet, but we can already see how it will make accessing the current user simpler in all of our HTTP 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 ensure that the user gets set on this context for every web request. We can achieve this by writing middleware that runs before our HTTP handlers run, giving us an opportunity to lookup and set the user.

More specifically, our 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.

The code in our middleware will look similar to the code that we originally used to be in the `CurrentUser` handler. It won't be exactly the same, but it will be very similar.

The key difference between our middleware and the original `CurrentUser` code is that we won't require a user be present. If we cannot lookup the current user for any reason, such as a missing session cookie, we will continue to the next handler without setting a user. The goal of our middleware isn't to require that a user be present; it is to set the user on the request context if a user is present. Later we will write another middleware to require a user's presence, and having these separate will make it easier to require a user for certain pages, but not all pages.

Currently, our `controllers` package contains pretty much all code related to HTTP. It doesn't have HTML rendering - that is for the `views` package - nor does it have any logic directly related to our SQL database, but anything related to cookies, HTTP handlers, etc is all in the `controllers` package. By that logic any new middleware we create will fit nicely into the `controllers` package.

Aside 17.4. Experiment with code structure

While we are learning one way to structure our code, there is more than one valid way and it can depend on a variety of factors such as how much code is involved in a project, how many team members are working on it, personal preference, and more. If we wanted, we could move the cookie-related logic into a `cookie` package, move all the HTTP handlers into a `handler` package, and we might even have a `middleware` package. Don't be afraid to refactor or try something new as a project grows and evolves.

Our application isn't very big right now, so utilizing the single `controllers` package makes sense to me when compared to having many packages, but if our application had many more HTTP handlers my opinion might change.

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 - we could break the HTTP handlers and the middleware up into two files if we wanted.

```
code controllers/users.go
```

We are going to add a `UserMiddleware` type. We know from the old `CurrentUser` code that we will need access to a SessionService to query for a user via their session token, so we can add that field as well.

```
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 that 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 both the `chi` package and our CSRF middleware use this pattern. It is best to be consistent when there isn't a good reason not to be.

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.ServeHTTP(w, r)
    })
}
```

Rather than create a new type that implements the `http.Handler` interface, we use the `http.HandlerFunc` type. This allows us to simply return a function that meets our needs. *Note: We learned about Handler and HandlerFunc earlier in the course in Section 2 if you need a recap.*

Next we can fill in the logic for our function. We want to first try to read the cookie and handle any errors. Unlike our HTTP handlers, our middleware is going to proceed to the `next` handler for most errors, as these might not prevent our application from returning a valid page. For instance, someone is trying to view our home page and there is an error reading their session cookie, that particular error won't prevent us from rendering the home page.

```
// 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 being
    // set, then return.
```

```

    next.ServeHTTP(w, r)
    return
}

```

Next we look up the user with the token. Once again, we can proceed to the next HTTP handler even if there is an error, though there may be cases where it is useful to log some of these errors.

```

// 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 just
    // cannot set a user.
    next.ServeHTTP(w, r)
    return
}

```

Now that we have a user our last step is to get the context from the HTTP request, create a new one with the user stored in it, set that new context as the request context, and finally proceed to the next HTTP handler.

```

// 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
```

While we are here, we can do a bit of cleanup. Near the bottom of the `main()` function, find the `fmt.Println` line and move it to just before the `http.ListenAndServe` function call. Ideally we want this print statement to occur just before we start the server so there isn't a big delay between the statement printing and our server actually starting.

```
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 new user middleware we just created. We will do this near our CSRF middleware setup, and use the existing session service.

```
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),
)

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)))
}

```

This code may seem confusing, but remember that both `SetUser` and `csrfMw` both accept an `http.Handler` and return a new one, so this is roughly equivalent to doing:

```

var h http.Handler
h = r // r is our router
h = umw.SetUser(h)
h = csrfMw(h)
http.ListenAndServe(":3000", h)

```

It is important to understand the order that the middleware will run. Otherwise, we might check if a user is present before the middleware to set a user ever runs leading to a bug. Alternatively, we don't want to waste time looking up the user if it turns out the CSRF protection is going to return an error for an invalid request.

In our current code the CSRF middleware will run first since it is the outermost wrapping. After that our SetUser middleware runs, followed by anything defined in the router. Later when we add middleware to ensure a user is present

we will need to ensure it runs after the set user middleware, otherwise it won't work properly.

Rather than having several nested function calls, we could use Chi's `Use` function to apply our middleware. The end result is the same, but the `Use` function might be clearer for you and your team. For now we will leave it as-is and verify that the code we have is working. To do this we need to:

1. Restart the server.
2. Sign in to a valid account.
3. Verify that the `/users/me` path works and renders the proper user account.

We now have access to the current user via the request context in any of our HTTP 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 a user is viewing a list of their galleries, we need to know who the current user is to render the correct galleries. When editing a gallery, we need to ensure that the user has access to that gallery. In both cases, a user must be present to handle the web request.

We are going to create middleware that helps with this by requiring that a user be set on the context before proceeding. Otherwise, our server will respond with a redirect to the sign in page. This will make it easier to write some of our HTTP handlers, as it will alleviate the need to verify that a user is present; we can instead assume that a user is present if the handler code is running.

We will be adding the require user middleware to the `UserMiddleware` type in the controllers package.

```
code controllers/users.go
```

We will name the method `RequireUser`, and we will assume that the `SetUser` middleware has been run BEFORE our middleware so we don't needlessly perform the same database lookups. This is why we want the middleware for setting a user to run on all web 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)
    })
}
```

In the require user middleware we will retrieve the current user from the request context. If the user is nil, we will redirect to the sign in page. Otherwise we have a valid user and can proceed with the next HTTP handler.

With our handler created, we need to apply it to the HTTP handlers that need a user to be present. We can do this in `main.go` where our routes are located.

```
code main.go
```

First we need to rearrange our code ab it. Currently, we declare middleware at the end of the `main()` function, which means we cannot use middleware when declaring our routes. We are going to move our middleware declaration to before our routes are created.

We also declare a few routes before we set up our services and controllers, then we proceed to declare a few more routes. We are going to rearrange this code so that all of our services and controllers are constructed prior to declaring any routes. This will make it easier to manage our code long term, as it will be a series of steps:

1. Set up a DB connection and handle migrations
2. Set up our services
3. Set up our middleware
4. Set up our controllers
5. Set up our router and routes

The updated code is shown below. Remember that this is the same code we had before, just rearranged a bit.

Note: The code is broken into a few code blocks to make it easier to consume and render in ebooks, but all of the code should go one after another in the `main()` function.

Step 1 - Set up the database connection.

```
func main() {
    // Set up a database connection
    cfg := models.DefaultPostgresConfig()
    fmt.Println(cfg.String())
    db, err := models.Open(cfg)
    if err != nil {
        panic(err)
    }
    defer db.Close()

    err = models.MigrateFS(db, migrations.FS, ".")
    if err != nil {
        panic(err)
    }
}
```

Step 2 - Set up our services. Put this directly after the database setup in the main function.

```
// Set up services
userService := models.UserService{
    DB: db,
}
sessionService := models.SessionService{
    DB: db,
}
```

Step 3 - Set up our services. Put this directly after the services setup in the main function.

```
// Set up middleware
umw := controllers.UserMiddleware{
    SessionService: &sessionService,
}

csrfKey := "gFvi45R4fy5xNBlnEeZtQbfAVCYEIAUX"
csrfMw := csrf.Protect(
    []byte(csrfKey),
    // TODO: Fix this before deploying
    csrf.Secure(false),
)
```

Step 4 - Set up our controllers. Put this directly after the middleware setup in the main function.

```
// Set up controllers
usersC := controllers.Users{
    UserService:    &userService,
    SessionService: &sessionService,
}
usersC.Templates.New = views.Must(views.ParseFS(
    templates.FS, "signup.gohtml", "tailwind.gohtml"))
usersC.Templates.SignIn = views.Must(views.ParseFS(
    templates.FS, "signin.gohtml", "tailwind.gohtml"))
```

Step 5 - Set up our router and routes. Put this directly after the controllers setup in the main function.

```
// Set up router and routes
r := chi.NewRouter()
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",
))))
r.Get("/signup", usersC.New)
r.Post("/signup", usersC.Create)
r.Get("/signin", usersC.SignIn)
r.Post("/signin", usersC.ProcessSignIn)
r.Post("/signout", usersC.ProcessSignOut)
r.Get("/users/me", usersC.CurrentUser)
r.NotFound(func(w http.ResponseWriter, r *http.Request) {
    http.Error(w, "Page not found", http.StatusNotFound)
})
```

Step 6 - Start the server. Put this at the very end of main.

```
// Start the server
fmt.Println("Starting the server on :3000...")
http.ListenAndServe(":3000", csrfMw(umw.SetUser(r)))
}
```

Now that our main function is more organized, we are going to utilize the **Use** method provided by Chi's router for applying middleware. Find the section of our **main()** function where we setup our routes and make the following changes.

```
// Set up our router
r := chi.NewRouter()
// These middleware are used everywhere.
r.Use(csrfMw)
r.Use(umw.SetUser)
// Now we set up routes.
r.Get("/", controllers.StaticHandler(views.Must(views.ParseFS(
    templates.FS,
    "home.gohtml", "tailwind.gohtml",
))))
// ... our routes remain unchanged
```

Now that we are applying routes directly to our router, we don't need to manually apply them when we call **ListenAndServe**. Head to the last line in our **main()** function and update it to use the router without wrapping it in any extra middleware.

```
http.ListenAndServe(":3000", r)
```

With our refactoring complete, we are ready to add the **RequireUser** middleware that we created. We currently only need this on the current user page, but long term we know that any page with the **/users/me** prefix will need a current user. For instance, if we added a route to handle a **PUT** request to this path to handle updates to the current user's information (like their email address), we would need a user to be present for that page as well. When we know that

an entire path prefix needs specific middleware, we can use Chi's sub-router functionality. This will apply middleware to all the routes that match a specific path prefix.

Find the following route in our `main.go` source file:

```
r.Get("/users/me", usersC.CurrentUser)
```

Replace the route with the following code.

```
r.Route("/users/me", func(r chi.Router) {
    r.Use(umw.RequireUser)
    r.Get("/", usersC.CurrentUser)
})
```

Any routes defined inside the function we pass to `r.Route` will have a `/users/me` prefix automatically applied to its route. This means we can say `r.Get("/", ...)` and it will match `/users/me` or `/users/me/`. These routes will also all have the require user middleware applied to them, meaning we can add new HTTP methods and routes and they will all require a user to be present in the request context.

We won't use this code, but below is an example to help illustrate what is possible with a sub-router.

```
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", ...)

    // Will match the path PUT /users/me
    r.Put("/", ...)
})
```

As we build our application, we will start to use nested routes to make it easier to manage routes while making it easier to apply middleware that are shared by a set of routes. 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 for any path with the `/app` prefix.

With our middleware added, we can update the current user HTTP handler to assume a user is present.

```
code controllers/users.go
```

If a user isn't present this code will panic, so we are relying on the middleware being setup properly. It may be useful to denote this comments.

```
// SetUser and RequireUser middleware are required.
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 could also update our `/signout` path to require a user if we wanted. This isn't necessary, since our code will continue to work even if a user isn't present, but if we wanted to do this we could use Chi's `With` function, which is similar to `Use`, but returns a new Chi router that has the middleware applied.

We won't be using the code, but an example is shown below.

```
// The SetUser middleware is applied to every route declared on the router
r.Use(umw.SetUser)
// This page will have the SetUser middleware applied
r.Get("/", ...)

// With only applies it to routes declared with the return value from With.
// This makes it easier to declare inline routes that use a specific middleware.
r.With(umw.RequireUser).Post("/signout", usersC.ProcessSignOut)

// This route will have the SetUser middleware still, but it will NOT have the
// RequireUser middleware applied.
r.Get("/other", ...)
```

Middleware can be helpful for simplifying HTTP handlers, but be careful to not overuse them. Sometimes it is best to have repeated code in HTTP handlers if it makes it clearer what the requirements are for testing that HTTP handler. As with most things, it is a balancing act of trade-offs.

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. We also show links to sign in or sign up even if a user is signed into their account. It would be nice to update our code to select which to display based on whether a user is signed in or not. It may also be useful to display the current user's avatar, email address, or team depending on what type of application we are building. In this lesson we are going to look at how to implement features like this by giving our templates access to the current user.

In previous lessons we added the user to our `http.Request` using the context. We also know that we can add functions to our templates that uses data from the `http.Request`. We did this when we added the `csrfField` function to our templates. We first stubbed the function:

```
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")
            },
        },
    )
}
```

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

Then once we were ready to execute we clone the original template and overwrite the `csrfField` function with one that uses information from the `http.Request` object.

```
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)  
            },  
        },  
    )  
    // ...  
}
```

Adding a current user function to our templates will be done in much the same way. We will first stub a function for accessing the current user, then when we go to execute the template we will add a request-specific method to the cloned template.

Given that we have seen how this approach works, we won't review too much and will dive right into implementing it.

Aside 17.5. 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 that is what we did. We created a custom data type for our views that had fields we wanted access to on any page, such as the current user, and the CSRF token. We would then access these fields inside of our templates as needed.

The user was still determined via the `http.Request`, so it was a very similar approach, but the key difference is that we needed to pass the correct data to any template that might want to access the CSRF token or the current user, while with the approach we are using in the course - using template functions - the functions are available in any template regardless of what data is passed into the template.

There are pros and cons to both approaches. For instance, we need to clone our templates before every request for the function approach to work, but the upside is it helps us avoid some bugs where we might need to pass data into several nested templates. Neither approach is innately better, but I do find that this approach is easier to utilize once a developer understands the stubbed function and why it is necessary. I am mostly just noting this now in case someone took both versions of the course and is wondering why this particular part of the code changed.

When adding functionality like this, it can help to imagine that we already have the template function and we just need to use it in our templates. By taking this approach, we get a better sense of how the function should work before we spend any time implementing it. Then once we understand what the function needs to do we can add it to our templates.

We can do this by pretending we have a `currentUser` function in our templates. We can then update our templates to figure out an ideal way for the function to work. In short, we are writing the template code the way we wished it would work, then working backwards to make it a reality. Let's try this in the tailwind template.

```
code templates/tailwind.gohtml
```

Find the `<div>` block that contains the sign out and sign in links and update it with the following code.

```
<div class="space-x-4">
    {{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 href="/signin">Sign in</a>
        <a href="/signup"
            class="px-4 py-2 bg-blue-700 hover:bg-blue-600 rounded">
            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. This is both easy to use, and easy to understand for any other developer who comes along.

If we attempt to compile and run our code, our templates will start returning an error. We need to add the `currentUser` function to our templates.

`code views/template.go`

Add a `currentUser` stub function in the `ParseFS` function. This enables us to parse templates that use the `currentUser` function even though we don't have access to the `http.Request` to implement the function just yet.

```
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")
            },
        },
    )
}
```

```

        },
    // ...
}
```

Next, head to the **Execute** method. Here we will add the real implementation of the **currentUser** function using the HTTP request passed into Execute. The code is pretty short since we already have a **User** function provided by the

```

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.Context())
            },
        },
    // ...
}
```

With our changes added, we need to restart the application and verify everything works. We can do this by signing out and verifying that we always see a sign in and sign up link, then once we sign into an account we should only see the sign out link.

Source Code

- [Source Code](#) - see the final source code for this lesson.
- [Diff](#) - see the changes made in this lesson.

17.9 Request-Sco ped Values

The context package doesn't enforce any limitations on how it can be used. We can quite literally store any values inside of it and our code will compile and run.

What the context package does provide are some guidelines about what types of data are a good fit for a context. Specifically, the docs state:

... carries deadlines, cancellation signals, and other request-scoped values across API boundaries and between processes.

The notable part here is that it specifies “request-scoped values.”

In this lesson we are going to discuss the pros and cons of storing various types of data in a context package so that we can better understand why the context package suggests limiting values to request-scoped values.

Imagine that we are building an application, and we know that our HTTP handlers will need access to a database. One way to approach this is to use the code we have, but an alternative approach might be to write middleware that adds the database connection as a context value. This would give all of our HTTP handlers access to the database without needing to create types like our users controller. We could even take this a step further and store our HTML templates inside the context as well.

```
// 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. We would no longer need our **Users** type in the controllers package.

```
// We wouldn't need this type, since we could get all this
// information from the context.
type Users struct {
    Templates struct {
        New      Template
        SignIn   Template
    }
    UserService     *models.UserService
    SessionService *models.SessionService
}
```

Instead, we could write functions that extract the database connection and the templates from the context.

```
// Fake example of how our HTTP handlers may look if retrieving the DB
// connection and the templates from the context.
func CreateUser(w http.ResponseWriter, r *http.Request) {
    ctx := r.Context()
    templates := context.Templates(ctx)
    newUserTpl = templates["users/new"]
    user := ...

    // We could let the CreateUser function get the DB connection from the
    // context as well.
    err := models.CreateUser(ctx, &user)
    // ...
}
```

Using the new **CreateUser** function appears easier to use because our setup process is drastically simplified. We no longer need to create the **Users** type and assign a templates, a UserService, or a SessionService. The downside is that it is no longer clear what dependencies the **CreateUser** function has.

- Does CreateUser need a database connection in the context?
- Does CreateUser call any functions that require a database connection?

- Does CreateUser need an API client to talk to Stripe?
- Does CreateUser need any templates to render a response?

We cannot answer any of these questions without closely examining the code. This problem is made even more challenging with nested function calls - like calling the `models.CreateUser` function - that might each have their own set of dependencies that are hidden inside a context. All of this makes it much harder to test and maintain the code.

Now let's go back to the code we have in our application.

```
type Users struct {
    Templates struct {
        New      Template
        SignIn  Template
    }
    UserService     *models.UserService
    SessionService *models.SessionService
}
```

In this code it is clear that our `Users` type requires both a `UserService` and a `SessionService`. It is also clear that we need a few templates. All of this information will make it significantly easier to test and maintain our HTTP handlers.

By using context values, we can hide dependencies. This makes our code much harder to test and maintain.

Storing request-scoped values can cause some of these same issues, but unlike a database connection or our templates, request-scoped values cannot be assigned when our application starts up. We would need to assign those dynamically when handling a web request using something like a closure. While this does work, context values tend to be easier to use for request-specific values even with their downsides.

In summary - limit the values stored in a context to request-scoped values, and even then try to limit it to situations where it is necessary. Anytime we add a

context value it adds a hidden dependency in our code, so it typically works best for values that can be expected on nearly every web request (eg the current user).

17.10 Context Exercises

17.10.1 Ex1 - Optimize the SetUser Middleware

In our middleware for setting the current user, we currently ignore a few potential errors and even if a session token is the empty string, we may still query for it. Update the code to log error messages and avoid unnecessary database lookups.

Chapter 18

Sending Emails to Users

18.1 Password Reset Overview

In this section we are going to be implementing the password reset process for our application. This is the process that will allow a user to regain access to their account if they forget their password, but still have access to the email address used to create their account. A typical password reset process hinges on the assumption that if a user has access to a specific email address, that they should also have access to any accounts that are assigned that email address.

In this lesson we are going to talk about how the entire process works so that we have a broad understanding of what we are about to work on.

1. User forgets their password

While not necessarily a requirement, the process usually starts when a user forgets their password and needs to reset it. Alternatively, some users may use this process to update their password.

2. User visits the forgotten password page.

Next a user must visit a page to submit their account information and initiate

the forgotten password process. Eg they might head to the path `/forgot-pw`, which will render a form that asks for the user's email address.

3. User submits the forgotten password form.

When the forgotten password form gets submitted, our server initiates the forgotten password process. This includes verifying that we have an account with the provided email address, creating a password reset token associated with that user, and emailing the user a link to reset their password using the reset token.

4. User reads password reset email.

To verify that a user has access to their email address, they must open an email our application sends them and click a link that includes the password reset token in it. This makes the process easy for a user while giving us a way to verify that the user does indeed have access to the email address they are trying to log into our application with.

5. Our application parses the reset token and renders a password update form.

When the user clicks the password reset link we email to them, it will send them to another page on our server and will include the reset token. Our server won't process this reset token until the user tries to update their password, but it will use the token to create a password reset form that includes the reset token as a HTML input (often a hidden one). This ensures that the user can submit both their new password and the reset token in a single form.

The user will then enter their new password into the form and hit submit.

6. Our server processes the password reset form.

When the password reset form is submitted our server will first need to validate the reset token. We do this by looking it up in our database and ensuring that it exists. Then we look up the user it is associated with so we know which user to update. Once the reset token has been validated we delete it so it cannot be

used more than once.

With the reset token validated and a user looked up, we can then proceed with updating the user's password. After that we can either redirect the user to the sign in page, or just sign them in since they have already authenticated themselves via their email. If we opt to sign them in, we can redirect them to a page like their dashboard.

Aside 18.1. Magic sign-in links are similar

If you have ever used an application that uses magic sign-in link - that is a link that gets emailed to you and when clicked will sign you in - this uses a nearly identical process. The server generates a token for that user, emails it to them, and when the user clicks the link it will include that token and confirm that the user has access to that email address. Once that is confirmed the user can be signed in.

Putting this all together, we will need to code the following:

Views (templates):

1. An HTML page with a form to type your email address and request a password reset token.
2. An HTML page that tells the user to go check their email.
3. An HTML page with a form to update a password.

Reset tokens model (service):

1. A migration adding a database table for reset tokens.

2. A reset type mapped to the DB table.
3. A models service to create & consume reset tokens.

Emails model (service):

1. An email service to manage sending the forgot password email and other future emails.

Controllers (handlers):

1. An HTTP handler to render and process the reset pw form
2. An HTTP handler to render and process the forgotten password form.

Main updates:

1. Setup routes
2. Setup template processing

This is a lot to update, so we will break this whole process into two sections in the course. In the first section we will work on sending emails from our server and understanding that entire process, then in the next section we will look at how to generate reset tokens and finish the password reset process. This will enable us to focus on email specific topics like SMTP before diving into the rest of the password reset code.

18.2 SMTP Services

In order to send out password reset emails, we need a way to send emails to our users. Historically, developers would set up a mail server and use it to send emails to their users. This worked because the Simple Mail Transfer Protocol (SMTP) is a standard protocol. Unfortunately, self-hosted mail servers have been abused heavily to send spam, so it is much harder to self-host an email server these days. We can still do it, but it is far more likely that a big email provider like Gmail or Yahoo will end up flagging our emails as spam.

Another option is to use our personal email provider to send emails. Gmail has historically provided a way to connect via SMTP, which we can use in our Go code to send emails. This can work well for a hobby project sending a very small number of emails, but almost all email providers have low rate limits as well as terms of service that may prevent this from being a good long term solution.

In practice, almost all web applications use a paid email service to send emails. There are three main reasons for this:

1. Reliability - we don't want important emails ending up in the spam folder.
2. Simplicity - setting up a mail server and maintaining it could become a full-time job that isn't a good use of time.
3. Scale - while sending email from a Gmail server may work when starting out, many services like this have limits to the number of emails that can be sent so these options don't scale well. We can also see a lot of emails ending up in spam if we send too many emails from our own hosted server.

Paid email services tend to have all the kinks ironed out, allowing developers to focus on their app rather than learning all the ins and outs of managing a mail server. They can also provide additional features such as:

- Dashboards to track total emails sent, bounce rates, spam reports, etc.
- Testing inboxes for developers.
- Improved deliverability by rate limiting and using other tricks to avoid emails being flagged as spam.
- And more depending on the service.

The major downside is the fact that it costs money to use a third party service. Many offer free tiers, but these often come with limitations due to people abusing them to send spam emails. Another potential downside is vendor lock-in. It can be tricky to extract all of an app's email data from an email service, so switching services can mean losing some historical data or extra work to retain that information.

For a paid application the cost isn't typically an issue. Most paid apps will earn more per user than will be spent on an email server. Where the cost becomes problematic is when learning or building hobby projects. These projects typically earn little to no money, and service fees can add up quickly. Luckily, many email services have a free tier that will work for our needs, so we will utilize one of those free tiers in our application.

When we want to send email with a third party mail service, there will typically be two ways to do this:

1. A custom API provided by the email service
2. Simple Mail Transfer Protocol (SMTP)

Custom APIs tend to be slightly easier to use for common cases, since they work much like any other API would work. This is especially true when the company provides an API library in Go, as we can just import their library and use their code to communicate with the service. The downside is that this can

create a bit more vendor lock-in with that particular service, since switching will mean rewriting the parts of our code that used their library.

My personal experience is that the lock-in is fairly minimal with how we design and write our code, but the lock-in does exist to a small degree.

Another option is to connect to an email service using SMTP. SMTP is an internet standard, so almost all languages have an SMTP library, and most email services support SMTP for sending emails. Using SMTP is sometimes more challenging to get working when compared to some of the custom APIs, but the upside is we can switch from provider to provider fairly easily.

We are going to look at a few providers who support SMTP and have a free tier and choose one of those. As of this writing, the following services all meet that criteria:

- [SendGrid](#)
- [mailtrap](#)
- [sendinblue](#)

Of these three options, mailtrap has a nice inbox designed for capturing test emails and displaying them in their UI. This allows us to sign up for accounts with email addresses we don't own - like [bob@bob.com](#) - and when we send that user an email with our test credentials mailtrap will not send a real email and will instead render it in their UI. This UI also makes it easy to view the HTML, plaintext, and other parts of an email, allowing for quick validation that our emails were constructed properly. These features can all be very helpful when learning.

Once we have picked a mail service and signed up, we need to get our SMTP credentials. Specifically, we need a **host**, **port**, **username**, and **password** for SMTP.

Name	Example value
<i>Host</i>	sandbox.smtp.mailtrap.io
<i>Port</i>	587
<i>Username</i>	unique for each account
<i>Password</i>	unique for each account

After creating an account and gathering this information, write it all down for future use in future lessons. The demo code in the next few lessons will use mailtrap credentials, but even if using the same service it is best to verify the host or port isn't different after signing up.

We are going to integrate with mailtrap using SMTP libraries rather than their API. The primary motivator here is to allow everyone to pick a mail service they prefer and to future-proof the course a bit. Even if the services listed remove their free tiers, new services are likely to both offer a free tier and offer SMTP.

Aside 18.2. I don't use SMTP in my projects

Throughout this course I have been trying to use and suggest libraries and integrations that I personally use in my own projects. SMTP is an exception to this. In my personal applications, I use [Mailgun](#) pretty exclusively, and I use their custom API to connect and send emails.

The primary reason I use Mailgun is that I have had a paid account on Mailgun for quite a long time and it has always worked well for me. I have multiple domains registered there that I can manage with a single account, and I have a long history of email records if I need to access them.

I do not recommend Mailtrap in this course solely because it doesn't have a truly free tier. They offer a free trial for a month, after which it costs \$35/mo. For most

of my students learning and building hobby applications that is quite expensive, so it didn't feel like a good fit for the course.

I opt to use their API over SMTP because it is what I started with, and it has always worked well for me. SMTP on the other hand has occasionally caused me headaches on edge cases. I also have no intentions to switch email providers anytime soon, as it would take a good bit of effort to update my domains and DNS records. Probably more effort than it would take to rewrite the actual code I use to use SMTP instead.

SMTP isn't a bad choice, otherwise I wouldn't use it in this course, but if you know you will be using a specific email service moving forward it might be worth looking at their APIs for sending your application's emails.

With our email service prepared, we are ready to start learning how to construct SMTP emails.

18.3 Building Emails with SMTP

When constructing an email with SMTP, it will have a format similar to the one below.

```
MIME-Version: 1.0
Date: Sun, 22 Jan 2023 11:54:48 -0500
To: jon@calhoun.io
From: test@lenslocked.com
Subject: This is a test email
Content-Transfer-Encoding: quoted-printable
Content-Type: text/plain; charset=UTF-8

This is the body of the email
```

In their simplest form, SMTP emails are pretty easy to construct. We only need

a few key/value pairs, and a body for the email. We could likely create this email using the `text/template` library, and using variables for things like the date, subject, and the body.

Over time, emails have evolved to become more complex. In addition to writing a plaintext body, most email clients expect us to also provide an HTML body that has things like hyperlinks (`<a>` tags). To support this, SMTP allows for multipart emails that have both a plaintext body and an HTML body. We can even include attachments to our emails.

Below is an example of an email with both a plaintext body and an HTML body.

```
MIME-Version: 1.0
Date: Sun, 22 Jan 2023 11:55:59 -0500
Subject: This is a test email
To: jon@calhoun.io
From: test@lenslocked.com
Content-Type: multipart/alternative;
boundary=2df315b0bd754b2cea495f617b327626853ede3bcbf4608725384a95937f

--2df315b0bd754b2cea495f617b327626853ede3bcbf4608725384a95937f
Content-Transfer-Encoding: quoted-printable
Content-Type: text/plain; charset=UTF-8

This is the body of the email
--2df315b0bd754b2cea495f617b327626853ede3bcbf4608725384a95937f
Content-Transfer-Encoding: quoted-printable
Content-Type: text/html; charset=UTF-8

<h1>Hello there buddy!</h1><p>This is the email</p><p>Hope you enjoy it</p>
--2df315b0bd754b2cea495f617b327626853ede3bcbf4608725384a95937f--
```

Rather than trying to create our emails from scratch and inevitably introducing a bug, we are going to use a library to construct our emails and send them using our SMTP server.

Go's standard library has an SMTP package, which might sound tempting, but unfortunately the library is quite limited in what it does and it is frozen with no plans to add additional features. For SMTP, third party libraries tend to be a better option in Go.

We will use [go-mail/mail](#) in the course. It doesn't have the most bells and whistles, but it will get the job done and help us construct and send SMTP emails.

Aside 18.3. Alternative SMTP libraries

A few newer Go libraries exist, but as of this writing most are under active development and I experienced issues with each while preparing the course notes. As a result, I opted to stick with go-mail/mail, as I know it can reliably be setup with pretty much any mail server.

Feel free to experiment with some of the other SMTP libraries out there. A few of them looked quite promising and may be a great option if they are stable.

We will start by adding the library to our application.

```
go get github.com/go-mail/mail/v2
```

Be sure to use the [v2 docs](#) for the mail package.

Next we will use the library to construct an email and print it out to the screen. This will enable us to verify that it looks like we might expect before trying to connect to an SMTP server. We will be using the [Message](#) type to build our email.

We will write our code in our [exp.go](#) file as we are learning, then we can proceed to copy the relevant code to our application when we want to write our production code.

```
code cmd/exp/exp.go
```

We can begin with the simplest email we can and print it out to the screen using the `WriteTo` method.

```
package main

import (
    "os"

    "github.com/go-mail/mail/v2"
)

func main() {
    msg := mail.NewMessage()
    msg.WriteTo(os.Stdout)
}
```

Be sure to check the imports and verify that the correct library is being imported. After that save and run the code.

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

This will give us output similar to that below.

```
MIME-Version: 1.0
Date: Tue, 31 Oct 2023 14:35:12 -0400
```

We need to add a few more keys and values to make our email useful. Namely, we need to know who the email is to, who it is from, the subject, and a body. We will do this with both a plaintext and an HTML body to see how the `mail` package handles that for us.

```
package main

import (
    "os"
    "github.com/go-mail/mail/v2"
)

func main() {
    from := "test@lenslocked.com"
    to := "jon@calhoun.io"
    subject := "This is a test email"
    plaintext := "This is the body of the email"
    html := `<h1>Hello there buddy!</h1><p>This is the email</p><p>Hope you enjoy it</p>`

    msg := mail.NewMessage()
    msg.SetHeader("To", to)
    msg.SetHeader("From", from)
    msg.SetHeader("Subject", subject)
    msg.SetBody("text/plain", plaintext)
    msg.AddAlternative("text/html", html)
    msg.WriteTo(os.Stdout)
}
```

Note: Be sure to double check your imports. If your editor is adding imports automatically, it may import the wrong package.

Typically the plaintext body and the HTML body should have roughly the same content, but with one using HTML markup. In this case both are different so that we can more easily differentiate between the two.

If we were to run this code we would see output similar to below.

```
MIME-Version: 1.0
Date: Tue, 31 Oct 2023 14:37:55 -0400
From: test@lenslocked.com
Subject: This is a test email
To: jon@calhoun.io
Content-Type: multipart/alternative;
    boundary=2f19fe97157f23786b302bca99fcec1e20a9673c470b73e7d1fc96a67eff

--2f19fe97157f23786b302bca99fcec1e20a9673c470b73e7d1fc96a67eff
Content-Transfer-Encoding: quoted-printable
Content-Type: text/plain; charset=UTF-8
```

```
This is the body of the email
--2f19fe97157f23786b302bca99fcece20a9673c470b73e7d1fc96a67eff
Content-Transfer-Encoding: quoted-printable
Content-Type: text/html; charset=UTF-8

<h1>Hello there buddy!</h1><p>This is the email</p><p>Hope you enjoy it</p>
--2f19fe97157f23786b302bca99fcece20a9673c470b73e7d1fc96a67eff--
```

After seeing the code, most of what we added is likely self-explanatory. We are setting various headers such as the subject, who the email is to, and who the email is from. There are additional headers that can be used, such as the **DKIM-Signature** which is being used by Google and other email providers to eliminate spam and phishing emails. Most of these potential headers can be found on this [mailtrap blog post](#).

With the **mail** package we will typically be setting most information using headers. The key exception to this is the body of the email. For that we need to use the **SetBody** and the **AddAlternative** methods and provide the format of each. In our case the two formats we are providing are:

- **text/plain**
- **text/html**

It isn't necessary to include both, but it is helpful for readers who may not do well with the HTML body, and for avoiding any issues with email clients.

When we have two different bodies added to our email, the content type gets set to multipart/alternative. This tells email clients that there are two body parts. It then proceeds to use a boundary string to help separate those body types, and inside of each boundary are the headers and body parts for each type.

If all of this is confusing, don't worry. We will be letting the **mail** package do all the heavy lifting for us. All we need to understand now is that we can create emails and add both an HTML and a plaintext body to it, and the recipients

client will decide which to use based on the user's preferences and the email client.

We now know how to construct an email using the `mail` package, so we will proceed with learning how to send the email.

Source Code

- [Source Code](#) - see the final source code for this lesson.
- [Diff](#) - see the changes made in this lesson.

18.4 Sending Emails with SMTP

We will continue working in the `exp.go` source file. In this file we are going to update our code to connect to our email service via SMTP and send an email.

Open up the source file.

```
code cmd/exp/exp.go
```

The same `mail` package provides a few utilities to interact with SMTP servers. The most relevant at this point is the `Dialer` type, which can be used to connect to an SMTP server and send an email.

We can use the `NewDialer` function to create a dialer with our SMTP credentials.

```
// All of these values will vary depending on your mail service.
const (
    host      = "sandbox.smtp.mailtrap.io"
    port      = 587
    username = "fill this in"
```

```

        password = "fill this in"
    }

func main() {
    // ... creating the email

    dialer := mail.NewDialer(host, port, username, password)
}

```

Once we have a dialer, we have a two options for sending emails:

1. Call [Dial](#) to connect to the SMTP server, then use the [SendCloser](#) returned to send emails and eventually close the connection.
2. Call the [DialAndSend](#) function which will connect to the SMTP server, send the email, then close the connection.

The first looks roughly like this:

```

sender, err := dialer.Dial()
if err != nil {
    // TODO: Handle the error correctly
    panic(err)
}
defer sender.Close()
err = mail.Send(sender, msg)
if err != nil {
    // TODO: Handle the error correctly
    panic(err)
}

```

The second option is a bit shorter since it does all the sending and closing for us.

```

err = dialer.DialAndSend(msg)
if err != nil {
    // TODO: Handle the error correctly
    panic(err)
}

```

Both options have merit depending on your use case. The first - using **Dial** and a **Sender** - is ideal when we want to connect to the SMTP server, then use that connection for a while to send various emails. An example of this might be a for loop where we process various documents and send out emails after processing each document. The primary benefit is that each email could be constructed inside a for loop and then sent without needing to reconnect to the SMTP server, or to build every message upfront.

The second option - using **DialAndSend** - is useful in circumstances where we just want to send a single email and close our server connection. Our current code is an example of this, so we will use the second option.

Head back to **exp.go** and call the **DialAndSend** function to send an email.

```
func main() {
    // ...

    dialer := mail.NewDialer(host, port, username, password)
    err := dialer.DialAndSend(msg)
    if err != nil {
        panic(err)
    }
}
```

Run the code and verify that the email was sent.

Remember that if you are using mailtrap's email testing feature, the email won't actually be sent. Instead, it will end up in your mailtrap account under the testing inbox. You can also verify the HTML source, the text, and other aspects of the message.

Source Code

- [Source Code](#) - see the final source code for this lesson.
- [Diff](#) - see the changes made in this lesson.

18.5 Building an Email Service

Let's take what we have in our experimental code and turn it into something our application can use. We can achieve this by creating a service in our models for interacting with emails.

```
code models/email.go
```

We can start by defining an `EmailService` type that we will eventually use to send emails. That is, we want to eventually write code like:

```
es := NewEmailService(...)
es.Send(email)
```

While we could use the `mail.Dialer` type directly in our code, this would add a tight coupling with the `mail` package. This would make it harder to refactor our code and use another SMTP package if the need arose.

If we were to instead create our own `EmailService` type and to hide all those details inside of this type's methods, we could refactor how the `EmailService` works down the road and not need to change any other code in our application.

Let's create an `EmailService` type and inside of it have an unexported field for the `Dialer` we plan to use. We can also add a `DefaultSender` field that can be set if needed, otherwise we will use a constant defined in our code.

```
package models

const (
    // DefaultSender is the default email address to send emails from.
    DefaultSender = "support@lenslocked.com"
)

type EmailService struct {
    // DefaultSender is used as the default sender when one isn't provided for an
```

```
// email. This is also used in functions where the email is a predetermined,
// like the forgotten password email.
DefaultSender string

// unexported fields
dialer *mail.Dialer
}
```

Note: Be sure to double check your imports. If your editor is adding imports automatically, it may import the wrong package.

Next we need a way to set this up, so we will create an **SMTPConfig** type. This will define all of the information we need to connect to our email service via SMTP.

```
type SMTPConfig struct {
    Host     string
    Port     int
    Username string
    Password string
}
```

With our config type created, we can create a function that accepts an SMTP config and returns an EmailService with a configured dialer.

```
func NewEmailService(config SMTPConfig) *EmailService {
    es := EmailService{
        dialer: mail.NewDialer(
            config.Host, config.Port, config.Username, config.Password),
    }
    return &es
}
```

Now we have an EmailService type that we can add methods to like **Send**, and whenever it is used the code making those function calls won't need to know what SMTP package we are using behind the scenes. This will make it easier to refactor our code in the future if necessary. It can also make it easier to

accept interfaces and use mocks during tests, but we won't be covering that in this course.

In the next lesson we will utilize the code we have here to send emails and use the DefaultSender that we setup.

Source Code

- [Source Code](#) - see the final source code for this lesson.
- [Diff](#) - see the changes made in this lesson.

18.6 EmailService.Send

Earlier when we created an email, we manually set quite a few headers using the `mail` package. These included things like who the email was to, the subject, and who it was from. We also had to set the body. When doing this, there wasn't any clear indication of which headers could be set. To remedy this, and to avoid having any dependency in the rest of our code on the `mail` package, we are going to create our own `Email` type that defines the information in an email. Once we have created that type we will add a `Send` method to the EmailService that will accept an Email and send it using our SMTP configuration.

Open up the email model.

```
code models/email.go
```

Add the following code to define our Email type. For now we are going to stick with the bare minimum of fields, but we could add more later if we decide they would be useful.

```
type Email struct {
    From      string
    To        string
    Subject   string
    Plaintext string
    HTML      string
}
```

Now we are ready to add a **Send** method to our EmailService. This method will use the **Email** to construct a message with the **mail** package, and then it will use the unexported **dialer** field to connect to our SMTP server and send the email.

```
func (es *EmailService) Send(email Email) error {
    msg := mail.NewMessage()
    msg.SetHeader("To", email.To)
    // TODO: Set the From field to a default value if it is not
    // set in the Email
    msg.SetHeader("From", email.From)
    msg.SetHeader("Subject", email.Subject)
    msg.SetBody("text/plain", email.Plaintext)
    msg.AddAlternative("text/html", email.HTML)
    err := es.dialer.DialAndSend(msg)
    if err != nil {
        return fmt.Errorf("send: %w", err)
    }
    return nil
}
```

Now we have a way to send emails without any dependency on the **mail** package in the rest of our code and we can focus on improving the code to make it a bit easier to use.

We can improve the body portion of our message construction by checking to see if either the HTML or plaintext bodies are empty. If both are present, we need to add an alternative body. If only one is present we can use it for the body.

```

func (es *EmailService) Send(email Email) error {
    msg := mail.NewMessage()
    msg.SetHeader("To", email.To)
    // TODO: Set the From field to a default value if it is not
    // set in the Email
    msg.SetHeader("From", email.From)
    msg.SetHeader("Subject", email.Subject)
    switch {
    case email.Plaintext != "" && email.HTML != "":
        msg.SetBody("text/plain", email.Plaintext)
        msg.AddAlternative("text/html", email.HTML)
    case email.Plaintext != "":
        msg.SetBody("text/plain", email.Plaintext)
    case email.HTML != "":
        msg.SetBody("text/html", email.HTML)
    }
    err := es.dialer.DialAndSend(msg)
    if err != nil {
        return fmt.Errorf("send: %w", err)
    }
    return nil
}

```

We can also improve how we determine who an email is from by using default values. Specifically, when the `From` field is set in an `Email`, we want to use it, but if it is empty we should fall back to the `DefaultSender` field on the `EmailService`. If the email service doesn't have a value set for the `DefaultSender` field, we will fall back to the constant declared in our package. This may seem like a lot, but it gives us a lot of flexibility down the road.

Adding all this logic inside the `Send` function might distract from its purpose, and it could be challenging to test in isolation. We are going to add a new helper function to handle this to help with both cases, plus it will make it easier to reuse the logic if needed.

```

// Used to set the sender of the message. The priority is:
//   - email.From
//   - EmailService.DefaultSender
//   - DefaultSender (package const)
func (es *EmailService) setFrom(msg *mail.Message, email Email) {
    var from string
    switch {

```

```

    case email.From != "":
        from = email.From
    case es.DefaultSender != "":
        from = es.DefaultSender
    default:
        from = DefaultSender
    }
    msg.SetHeader("From", from)
}

```

With our helper function in place, we can go back to the `Send` method and update it to use the helper when deciding who a message is from.

```

func (es *EmailService) Send(email Email) error {
    // ...
    // This replaces msg.SetHeader("From", ...)
    es.setFrom(msg, email)
    // ...
}

```

Finally, we can update `exp.go` to test that this all works as expected.

code cmd/exp/exp.go

We start by constructing our email.

```

func main() {
    email := models.Email{
        From:      "test@lenslocked.com",
        To:        "jon@calhoun.io",
        Subject:   "This is a test email",
        Plaintext: "This is the body of the email",
        Html:       `<h1>Hello there buddy!</h1><p>This is the email</p><p>Hope you enjo
    }
    // ...
}

```

This is mostly the same as what we had, but it uses our new `Email` type.

Aside 18.4. VSCode Tip - Multiselect and ChangeCase

Using VSCode's multiselect and an extension like [change-case](#) our existing code can be converted into the new Email type using the existing `to`, `from`, and other variables. This can be seen in the videos for this lesson if you have the Complete Package of the course.

Next we need to setup our email service so we can send our email.

```
func main() {
    email := models.Email{
        //...
    }
    es := models.NewEmailService(models.SMTPConfig{
        Host: host,
        Port: port,
        Username: username,
        Password: password,
    })
}
```

Finally we can call `Send` to send our email. We also want to check for any errors that may occur.

```
func main() {
    email := models.Email{
        //...
    }
    es := models.NewEmailService(models.SMTPConfig{
        // ...
    })
    err := es.Send(email)
    if err != nil {
        panic(err)
    }
    fmt.Println("Email sent")
}
```

Run the code and verify that an email was sent correctly.

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

Be sure to check your test inbox if using Mailtrap. It is also a good idea to test some of the extra logic we added; see what happens if only a plaintext body is provided, or when the **From** field isn't set on an **Email**.

Source Code

- [Source Code](#) - see the final source code for this lesson.
- [Diff](#) - see the changes made in this lesson.

18.7 Forgot Password Email

We have a way to send any email using our email service, but in practice we will generally be sending templated emails. That is, we will be sending the same email with a few values changes. In the case of our forgotten password email, it will always be a message that explains how to reset a password, and the only change from email to email will be who we are sending the email to, and the URL they will use to reset their password. We can make this process easier by providing a function that handles most of the details for us and only requires the necessary information for the email.

Open the email model:

```
code models/email.go
```

We are going to create a method called `ForgotPassword` which will accept two parameters:

- Who the email is to.
- The URL that should be used to reset the user's password.

With those two pieces of information we will build and send our email. The code for this is shown below.

```
func (es *EmailService) ForgotPassword(to, resetURL string) error {
    email := Email{
        Subject:      "Reset your password",
        To:          to,
        Plaintext:   "To reset your password, please visit the following link: " + resetU
                    HTML:       `<p>To reset your password, please visit the following link: <a href
    }
    err := es.Send(email)
    if err != nil {
        return fmt.Errorf("forgot password email: %w", err)
    }
    return nil
}
```

Now we can call this function with only the necessary data and it will send the same forgotten password email to whichever user needs it. We can see how much this simplifies our code by testing it in our `exp.go` source file.

code cmd/exp/exp.go

Our main function will end up looking similar to the code below. We no longer need to use the `Email` type at all.

```
func main() {
    es := models.NewEmailService(models.SMTPConfig{
        Host:      host,
        Port:      port,
        Username: username,
        Password: password,
    })
    err := es.ForgotPassword(
```

```
    "jon@calhoun.io",
    "https://lenslocked.com/reset-pw?token=abc123")
if err != nil {
    panic(err)
}
fmt.Println("Email sent")
```

Note: I opted to use a fake resetURL in this example since we aren't constructing those yet.

For more complex emails we may want to use a template library to construct the plaintext body and the HTML. When doing this, we would likely want to embed the templates in our code like we do for the HTML view templates, but for something as simple as this email we can manually construct the body. We also don't need to worry about code inject as long as the resetURL is something we construct without using any user-provided input.

Source Code

- [Source Code](#) - see the final source code for this lesson.
- [Diff](#) - see the changes made in this lesson.

18.8 ENV Variables

Up until this point we have been storing credentials like API keys directly in our source code. While this may not seem like a big deal while we are the only ones working on our application, it can become a major issue if many developers are working on a project, and not all of them have access to the keys. It can also be an issue if we open source our project and those credentials are exposed to the world, as anyone with our API keys could log into a service and utilize our account. If abused in the wrong way, this could be used to scam our users, increase our bills, or even get our account banned.

If we remove the API keys from our code, almost all version control systems (like git) will keep the keys in the history of our source code. The only way to remedy this issue is to both remove our keys from the source code, and to reset our credentials making the old ones invalid. *Whether we are using mailtrap or another service, there should be a way to reset the credentials. We will want to do that before proceeding with the lesson.*

Aside 18.5. Other Sensitive Keys

We have other sensitive keys hard-coded into our application as well. For instance, our CSRF key is stored in `main.go` when we probably don't want anyone to have access to it. Later in the course when we prepare for production we will review our application and remove any sensitive keys from our source code and utilize ENV variables.

With the exception of the CSRF key, we have mostly avoided having any sensitive data in our code. Now that we are using API keys, we need to look at how we can give our application access to the keys without hard-coding them into our source code. We are going to achieve this by providing sensitive data to our application via environment (ENV) variables. When our application starts, it will read these ENV variables and use them when setting up our SMTP config. Interestingly, this will also give us the ability to use different API keys in different environments (like local development vs production).

There are a variety of ways to set environment variables. The simplest is to prefix your `go run` command with the them. For instance, imagine we had an application that needs an Stripe API key. We could use the following to run our application with an ENV variable named `STRIPE_KEY` and the value `abc123` assigned to it.

```
STRIPE_KEY=abc123 go run main.go
```

If we run our code this way, we can call `os.Getenv` to access the `STRIPE_KEY` environment variable, and it will have a value of `abc123`. We can do this with multiple keys by putting spaces between each key/value pair.

Setting environment variables this way only works for this single command, and it would be tedious to type out every secret our app needs before running the `go run` command. We could end up needing a dozen ENV variables, leading to a very lengthy command.

Another way to approach this is to update our terminal environment to include the keys and values we use. *You may have encountered this when updating the PATH environment variable to include programs installed via `go install`.*

The downside to this approach is that these variables are set for every program you run, and there are occasionally times where key names may overlap between projects.

A common approach to this problem is to store our ENV variables in a file and then use a library or tool of some sort to load these secrets as ENV variables that our program can use. When doing this, it is important that the file being used is NOT stored in source control (like git); if it was, this would defeat the entire purpose of extracting it from our source code.

By taking this approach, developers can each have a unique set of ENV variables that they use when running the application, and our production server can have its own unique set of ENV variables in its `.env` file. We can also change this file at any time to test different configurations.

The `.env` file is a regular file stored on the hard drive, so anyone with access to our production server will have access to it, but if we were to open source our project we wouldn't leak the keys along with the source code.

We are going to use the `godotenv` library to assist us in setting this up. We will

start by installing the library.

```
go get github.com/joho/godotenv
```

Next, we are going to create a `.env` file where we will store our environment variables.

```
code .env
```

In this file we will add our SMTP credentials. It is a good idea to use prefixes to help organize our keys. In this case all of our SMTP keys will have the `SMTP_` prefix.

```
SMTP_HOST=sandbox.smtp.mailtrap.io
SMTP_PORT=587
SMTP_USERNAME=your-username
SMTP_PASSWORD=your-password
```

We do NOT want this file stored in our source code's version control, so we need to make sure we tell it to ignore this file. Git is by far the most common source control, and we tell git to ignore a file by adding it to a `.gitignore` file.

```
code .gitignore
```

Add the following line to the file. If your file already has contents from Section 2 just be sure to add this to its own line somewhere in the file.

```
.env
```

While we don't want our `.env` file with real keys to be stored in source control, it is a good idea to provide a template so that other developers know what keys are required. One way to do this is by creating another template file that can be copied to bootstrap the `.env` file. Let's do that now.

```
code .env.template
```

This file will get checked into source control, and we can add any common non-secret values like the `SMTP_HOST`. This way other developers will have the default values to work with, but can change it if necessary for their own local development. When the value is a secret of some sort, we can either leave it blank or provide a string like, "fill this in" to let the next developer know it needs to be set.

```
SMTP_HOST=sandbox.smtp.mailtrap.io
SMTP_PORT=587
SMTP_USERNAME=
SMTP_PASSWORD=
```

The last step is to update our code to read from these environment variables. The [godotenv](#) page has an example of how to do this:

```
package main

import (
    "log"
    "os"

    "github.com/joho/godotenv"
)

func main() {
    err := godotenv.Load()
    if err != nil {
        log.Fatal("Error loading .env file")
    }

    s3Bucket := os.Getenv("S3_BUCKET")
```

```
secretKey := os.Getenv("SECRET_KEY")
// now do something with s3 or whatever
}
```

We can do use this code to deduce what changes we need to make in our code to read ENV variables. We will start by updating `exp.go`, but long term we will want to make the changes in our `main.go`.

code cmd/exp/exp.go

Update the code to use the `godotenv` library to load and read the environment variables. Be aware that all ENV variables are read as strings, so we need to convert them to integers and other data types if that is what we want in our code. We can see that with the port in the code below.

```
package main

import (
    // ...
    "github.com/joho/godotenv"
    // ...
)

// Delete the const block

func main() {
    err := godotenv.Load()
    if err != nil {
        log.Fatal("Error loading .env file")
    }
    host := os.Getenv("SMTP_HOST")
    portStr := os.Getenv("SMTP_PORT")
    port, err := strconv.Atoi(portStr)
    if err != nil {
        panic(err)
    }
    username := os.Getenv("SMTP_USERNAME")
    password := os.Getenv("SMTP_PASSWORD")

    es := models.NewEmailService(models.SMTPConfig{
```

```
        Host:      host,
        Port:      port,
        Username: username,
        Password: password,
    })
err = es.ForgotPassword("jon@calhoun.io", "https://lenslocked.com/reset-pw?token=abc123"
// ...
}
```

Once the code is updated be sure to run it and verify it is working.

Long term we will use environment variables for all of our configuration variables, which will include:

- Postgres connection info.
- CSRF info (key and whether or not to run in secure mode).
- SMTP connection info.
- Plus any other API keys we may use in the future.

Aside 18.6. What about other libraries?

There are other libraries out there with more features, the ability to read from YAML, JSON, and other file types, the ability to verify that a key is present, and more. Feel free to experiment with another library if that interests you.

I personally tend to stick with something that uses ENV key/value pairs because these can easily be used across applications. As a quick example, when we deploy our application later in the course we use this same `.env` file in Docker to set up our database.

Source Code

- [Source Code](#) - see the final source code for this lesson.
- [Diff](#) - see the changes made in this lesson.

18.9 Email Exercises

18.9.1 Ex1 - Generate emails with templates

Our emails are currently pretty simple, but with more complex emails we might want to utilize Go's [text/template](#) and [html/template](#) packages to generate our emails. Experiment with creating emails - both HTML and plaintext - using templates.

If you want to take this further, be sure to include these templates via embedding. You could even experiment with moving the templates to their own package where the templates are compiled and then exported as variables that can be used by our EmailService.

Chapter 19

Completing the Authentication System

19.1 Password Reset DB Migration

When a user requests a password reset, we need to create a secret token that we can email to them to verify their identity. The user won't see the token, but it will be added to the link we send them as a URL parameter.

When creating these password reset tokens, they will be nearly identical to our session tokens. We will only store the hash of the token in our database, and each user will have one or zero reset tokens associated with their account at any time. The major difference is that we are going to add an expiration date to the reset token so that it is only valid for a limited time.

While this may sound odd, it shouldn't come as a surprise given that these tokens will allow a user to both sign in and change the password on an account, so they need to be just as secure as a session token.

Technically we could add expiration dates to our sessions making them even more similar.

Let's start by creating a migration for the password reset tokens. PasswordResetToken is quite long to type, so we will call them PasswordReset in our code. With that in mind we get the following migration:

```
cd migrations
goose create password_reset.sql
# We are working in a solo environment, so we don't need to use timestamped
# versions and can set a fixed sequential version immediately if we want.
goose fix
code *_password_reset.sql
```

We are working on this app by ourselves, so we can go ahead and run `goose fix` right away knowing there won't be any versioning issues. If we were working with a team of developers we would wait until we are about to commit, sync our code with source control, and then fix the version.

Once we open the migration file, we need to add the following SQL to it. The only new thing here is the `expires_at` column which has a type of `TIMESTAMPTZ`. This is a timestamp with a time zone in Postgres, and we want to ensure it is never null so we cannot accidentally create a password reset token that doesn't expire.

```
-- +goose Up
-- +goose StatementBegin
CREATE TABLE password_resets (
    id SERIAL PRIMARY KEY,
    user_id INT UNIQUE REFERENCES users (id) ON DELETE CASCADE,
    token_hash TEXT UNIQUE NOT NULL,
    expires_at TIMESTAMPTZ NOT NULL
);
-- +goose StatementEnd

-- +goose Down
-- +goose StatementBegin
DROP TABLE password_resets;
-- +goose StatementEnd
```

Be sure to save the file, then restart the app to verify that the migration runs before proceeding to the next lesson.

```
cd ..  
go run main.go
```

Within our output we should see a message indicating that the migration ran successfully.

```
2023/11/02 20:17:11 OK  00003_password_reset.sql (63.99ms)  
2023/11/02 20:17:11 goose: successfully migrated database to version: 3
```

Source Code

- [Source Code](#) - see the final source code for this lesson.
- [Diff](#) - see the changes made in this lesson.

19.2 Password Reset Service Stubs

Now that we have our database migration, let's create the Go model used to represent our password reset tokens. We will create a new source file for this code.

```
code models/password_reset.go
```

We will call our type `PasswordReset`, and it will have look and behave nearly identical to the `Session` type with the exception of an `ExpiresAt` field that represents when a password reset token expires.

```
package models

import "time"

type PasswordReset struct {
    ID      int
    UserID int
    // Token is only set when a PasswordReset is being created.
    Token   string
    TokenHash string
    ExpiresAt time.Time
}
```

Much like our session type's **Token** field, the **Token** field here will only be populated when we create a new password reset token. This is because we only store the hash in our database, and we cannot recreate it when looking up a password reset in our database.

For the expiration we are using the **time.Time** type from the standard library. This works well when the time cannot be null, but if it can be null we would want to consider a type like [sql.NullTime](#).

```
// This is part of the database/sql package
type NullTime struct {
    Time time.Time
    Valid bool // Valid is true if Time is not NULL
}
```

This and several other **NullXXX** types are provided by the `sql` package, and can be useful when we need to know if a database entry is null or not.

With our model defined, let's create a service for interacting with this database table. We can also add a few fields based on what we know from creating our sessions service.

```
type PasswordResetService struct {
    DB *sql.DB
    // BytesPerToken is used to determine how many bytes to use when generating
```

```
// each password reset 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
}
```

Much like the sessions code, we are going to use the `BytesPerToken` to set a default number of bytes for each token. If this is not provided, our code will need to default to `MinBytesPerToken`, which is defined in our `session.go` source file. With this constant now representing the minimum number of bytes for multiple types of tokens we might want to move it to a more central location, but with our codebase being as small as it is we will leave it as-is. We could always refactor in the future when we have multiple constants like this.

The next thing we need to consider is how we are going to set our expiration dates for password reset tokens. One way to achieve this is by defining a duration that each token should be valid for, then we could add that duration to the current time when we create each token.

Much like our bytes per token, we can define a default duration and allow it to be overridden via a field on the service.

```
const (
    // DefaultResetDuration is the default time that a PasswordReset is
    // valid for.
    DefaultResetDuration = 1 * time.Hour
)

type PasswordResetService struct {
    // ...

    // Duration is the amount of time that a PasswordReset is valid for.
    // Defaults to DefaultResetDuration
    Duration time.Duration
}
```

Durations in Go often use a format like we see here. This makes them much easier for developers to read, while also ensuring they are converted into the

proper type behind the scenes; that is, they will be stored as a `time.Duration`, which is ultimately an `int64`. We can even create more complex durations with minutes, seconds, etc.

```
const DemoDuration = 7*time.Hour + 30*time.Minute + 15*time.Second
```

This works because each constant (`time.Hour`, `time.Minute`, etc) is actually the number of nanoseconds for that duration, so a `time.Hour` is really 3600000000000 nanoseconds. When we multiple this by a number like 7 we get the number of nanoseconds in 7 hours, but our code is much easier to read than the following example.

```
const DemoDuration = 7*3600000000000 + 30*600000000000 + 15*1000000000
```

Next we are going to stub out the methods that we think we will need. Following that we will write code that uses our stubbed methods. This will help us verify that we are designing our services to be easy to use, rather than designing them around what is more convenient to initially code.

Let's look at an example to help demonstrate this point. We are going to need a function that allows us to create a password reset token for a given user. If we were to look at our `PasswordReset` type, we would see that this means we need a user's ID, so the most convenient way to write this code would be to require a `userID` as a parameter.

```
func (service *PasswordResetService) Create(userID int) (*PasswordReset, error) {
    // ...
}
```

Now if we were to attempt to use this from our controller, we would see that when we process the forgotten password form, the user inputs an email address. This means we do not have access to their user ID without first looking up the

user, checking for errors, and then finally calling the **Create** method on the password reset service.

Alternatively, we could design our password reset service to accept an email address and handle looking up the user for us. This would make it far easier to handle password resets, as the complexity of verifying if an account exists or not is handled by the password reset service.

Another example is looking up a password reset token after a user submits the password reset form. If we didn't think about the use case, we might provide a method to look up password reset tokens, and another to delete it once it is used. This seems okay at first, but in practice we will never look up a password reset token without subsequently deleting it, so we could replace these two methods with a single **Consume** method that handles all of this logic.

Stubbing methods can help us verify that the functions we are exporting have the correct set of parameters, and that they provide the best way to achieve our goals.

Head back to our `password_reset.go` source file and add the following method stubs.

```
func (service *PasswordResetService) Create(email string) (*PasswordReset, error) {
    return nil, fmt.Errorf("TODO: Implement PasswordResetService.Create")
}

// We are going to consume a token and return the user associated with it, or return an error if
func (service *PasswordResetService) Consume(token string) (*User, error) {
    return nil, fmt.Errorf("TODO: Implement PasswordResetService.Consume")
}
```

With these method stubs we can add HTTP handlers to our controller package for resetting passwords and verify that they work well for our needs. If they don't, we can refactor them with minimal wasted effort.

Source Code

- [Source Code](#) - see the final source code for this lesson.

- [Diff](#) - see the changes made in this lesson.

19.3 Forgot Password HTTP Handler

The best way to verify that a service is designed well is to use it. While we can sit around and theorize what each function should look like, we are all human and we will inevitably forget small details and make mistakes. When using an API those issues tend to become much more apparent.

We plan to use our password reset service inside our users controller, so we can start there.

```
code controllers/users.go
```

We are going to first start with a page to render our forgotten password form. We will need a template for this, so we can add it to our **Users** type.

```
type Users struct {
    Templates struct {
        New          Template
        SignIn       Template
        // Add this line
        ForgotPassword Template
    }
    UserService     *models.UserService
    SessionService *models.SessionService
}
```

Next we define an HTTP handler to render the password reset form. This will be the first step in our password reset process where a user enters the email for the account they need to regain access to. Like the sign in form, we can parse any URL query params and use those to prefill the form. These are unlikely to get used in this particular page, but it is simple enough that it doesn't hurt to add it.

```
func (u Users) ForgotPassword(w http.ResponseWriter, r *http.Request) {
    var data struct {
        Email string
    }
    data.Email = r.FormValue("email")
    u.Templates.ForgotPassword.Execute(w, r, data)
}
```

This handler doesn't help us validate whether or not password reset service was designed well, but what will help us is the handler that processes this form. When a user submits their email address, we need to use the password reset service to create a password reset token. Let's create the handler to process this form and get the email address from the HTML form.

```
func (u Users) ProcessForgotPassword(w http.ResponseWriter, r *http.Request) {
    var data struct {
        Email string
    }
    data.Email = r.FormValue("email")
}
```

Once we have the email address we want to call the PasswordResetService's **Create** method, so we need access to this service. We give our handler access to this by adding it to the **Users** type as a new field.

```
type Users struct {
    Templates struct {
        New          Template
        SignIn       Template
        ForgotPassword Template
    }
    UserService      *models.UserService
    SessionService   *models.SessionService
    // Add this
    PasswordResetService *models.PasswordResetService
}
```

Now we can call the function to create our password reset token. If an error occurs, we will use an internal server error for now. We should add a note

to handle specific error cases in the future though, as we may want to handle specific errors differently.

```
func (u Users) ProcessForgotPassword(w http.ResponseWriter, r *http.Request) {
    // ...
    pwReset, err := u.PasswordResetService.Create(data.Email)
    if err != nil {
        // TODO: Handle other cases in the future. For instance,
        // if a user doesn't exist with the email address.
        fmt.Println(err)
        http.Error(w, "Something went wrong.", http.StatusInternalServerError)
        return
    }
}
```

Aside 19.1. No account found errors

Many websites will render a vague response when submitting the form that says, “If an account is found with this email address, we have sent instructions to reset the password.” The theory is that this doesn’t reveal what email addresses have accounts on the app, but in reality this information is discoverable simply by trying to create a new account with that same email address. As a result, I find this type of precaution to be unnecessary, and opt to go the opposite direction. When a user submits the form and their email address does not match an account, I prefer to tell them that no account was found with that email address. This helps prevent confusion when an email address has a typo, and can also remind people that they never signed up for an account and need to create one.

Regardless of which route you go, we would need to handle that specific error to render the correct message.

Next we need to utilize the email service to send our reset password email. That means we need to add the email service to our **Users** type as a field.

```

type Users struct {
    Templates struct {
        New          Template
        SignIn       Template
        ForgotPassword Template
    }
    UserService      *models.UserService
    SessionService   *models.SessionService
    PasswordResetService *models.PasswordResetService
    // Add this
    EmailService     *models.EmailService
}

```

Now we can finish up our HTTP handler function for processing the forgotten password form by sending the user an email. To do this we need to construct a URL that the user can click to reset their password. We will use the `url.Values` type to URL encode our reset token, and we will hard-code the base URL we are using for now.

```

func (u Users) ProcessForgotPassword(w http.ResponseWriter, r *http.Request) {
    // ...
    vals := url.Values{
        "token": {pwReset.Token},
    }
    // TODO: Make the URL here configurable
    resetURL := "https://www.lenslocked.com/reset-pw?" + vals.Encode()
    err = u.EmailService.ForgotPassword(data.Email, resetURL)
    if err != nil {
        fmt.Println(err)
        http.Error(w, "Something went wrong.", http.StatusInternalServerError)
        return
    }
}

```

The `url.Values` type may seem a bit odd at first. This is because it takes a slice of strings as the value for each key.

```

type Values map[string][]string

```

While this can be useful in some very specific circumstances, most of the time we will only have a single value assigned to a key.

When creating the full URL, we can't use a relative path like `/reset-pw` because this will be clicked from inside a mail client. If it were a link on our website the relative path would work because our browser knows to use the same domain, but in an email client the user won't be in a browser, or will be on another domain like `gmail.com`, neither of which will work. Later we may want to make this domain configurable - possibly as a field on our `Users` type - but for now a hard-coded URL with a TODO comment will suffice.

Lastly, we need to add a template telling the user to check their email and then render it.

```
type Users struct {
    Templates struct {
        New           Template
        SignIn        Template
        ForgotPassword Template
        // Add this
        CheckYourEmail Template
    }
    UserService      *models.UserService
    SessionService   *models.SessionService
    PasswordResetService *models.PasswordResetService
    EmailService     *models.EmailService
}

func (u Users) ProcessForgotPassword(w http.ResponseWriter, r *http.Request) {
    // ...

    // Don't render the token here! We need them to confirm they have access to
    // their email to get the token. Sharing it here would be a massive security
    // hole.
    u.Templates.CheckYourEmail.Execute(w, r, data)
}
```

It may be tempting to render the reset token in our template, but be sure that you do NOT do this! If we did, it would mean that any user could enter an email address like `jon@calhoun.io` and then reset the password for that account without having access to the email account. By sending an email and waiting

for the user to click on a link in the email we are ensuring that they have access to the email address associated with the account.

Our error handling isn't perfect, and we haven't actually created several of the templates we use, but this gives us a strong sense that our design works well for creating new password resets tokens. It also helps demonstrate how using this type of code structure allows us to code our handlers before creating all the templates and other necessary parts. We could even use an interface to design our PasswordResetService without creating the method stubs.

```
// Do not code this. It is just an example.
type Users struct {
    Templates struct {
        New           Template
        SignIn        Template
        ForgotPassword Template
        CheckYourEmail Template
    }
    UserService      *models.UserService
    SessionService   *models.SessionService
    // Here we see how it is possible to use an interface to define what we expect
    // our service to look like without actually creating the service. We could
    // even keep this code permanently, and assign a *models.PasswordResetService
    // to this field since it matches the interface.
    PasswordResetService interface {
        Create(email string) (*models.PasswordReset, error)
    }
    EmailService     *models.EmailService
}
```

Note: We won't be using this interface approach in the course, so if you opt to use it you will need to manually update the interface as we use additional methods provided by the PasswordResetService. The example was meant for illustration purposes only.

Our code won't work since we haven't created all our templates or fully implemented our password reset service, but it helps us verify that our design will work fairly well.

Source Code

- [Source Code](#) - see the final source code for this lesson.
- [Diff](#) - see the changes made in this lesson.

19.4 Asynchronous Emails

In the previous lesson we updated our HTTP handler to process the forgotten password form, send a user an email with instructions to reset their password, then we render a page letting the user know to check their email. We still have a few details left to implement - like the HTML template letting the user know to check their email - but before we do that we are going to discuss why our code doesn't send the password reset email asynchronously.

Let's first look at the code we wrote. Specifically, the section where we send the email.

```
func (u Users) ProcessForgotPassword(w http.ResponseWriter, r *http.Request) {
    // ...

    // TODO: Make the URL here configurable
    resetURL := "https://www.lenslocked.com/reset-pw?" + vals.Encode()
    err = u.EmailService.ForgotPassword(data.Email, resetURL)

    // ...
}
```

In this code we send an email and our code waits for this process to finish before continuing. In many cases, tutorials and courses will teach developers to send emails asynchronously. This can be done with a background job system of some sort or using a goroutine - the actual details of how aren't important right now. The main goal is to send the email in a way that doesn't block the HTTP handler from responding to the user. When done correctly, this can make an application feel faster and more responsive since the user isn't forced to wait for our email server.

We are very intentionally not doing this in our forgotten password handler. Instead, we are opting to block until the email is sent, and only then do we respond to the user’s web request.

When sending an email to a user, we need to ask ourselves whether the user will be blocked until they receive that email. For instance, if a user signs up for a new account, we might send them a welcome email. If this email has a confirmation code that they need to continue accessing their account, then we should likely wait for that email to send before telling the user to check their inbox. On the other hand, if that email just welcomes the user and isn’t necessary to continue using their new account, there is no reason to block the HTTP handler. We can instead send the email `async` and the user can read it whenever it shows up.

When a user submits the forgotten password form, they are telling us that they can no longer access their account. They are blocked until we give them a link to reset their password. If we were to render a page that says, “Check your email for instructions on resetting your password” when the email hasn’t actually sent, this can cause all sorts of confusion. The user may end up refreshing their inbox, checking their spam folder, and submitting the forgotten password form again. This can get even worse if they submit the form a second time and then receive the original password reset email, only to discover the reset token in that email is no longer valid because submitting the form a second time creates a new password reset token.

Password reset emails are an example of a situation where following “best practices” can result in a worse user experience (UX). In most cases, the better UX is to have an HTTP handler that is slightly slower to respond to the web request, but when it does respond we know for certain that the email has been sent to the user. They may still need to refresh their inbox, but we have done everything we can on our end to make the process clear for the user.

For other emails, using a background job or a goroutine to send the email will often work well, but it is worth remembering that Go’s standard library already handles each incoming web request in a goroutine, so having an HTTP handler

wait on an email to send isn't a server bottleneck like it can be in a language like Python or Ruby where most web servers process a single request at a time.

19.5 Forgot Password HTML Template

The forgotten password page is going to be very similar to the signin page. We will adjust the form a bit, but not too much. We start by creating the source file.

```
code templates/forgot-pw.gohtml
```

We can then add the following HTML to the template.

```
{
{{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">
      Forgot your password?
    </h1>
    <p class="text-sm text-gray-600 pb-4">No problem. Enter your email address and we'll send you a link to reset it.
    <form action="/forgot-pw" method="post">
      <div class="hidden">
        {{csrfField}}
      </div>
      <div class="py-2">
        <label for="email" class="text-sm font-semibold text-gray-800">
          Email Address</label>
        <br>
        <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
          ">
      </div>
    </form>
  </div>
</div>
```

```

        text-gray-800
        rounded
    "
    value="{{ .Email }}"
    autofocus
/>
</div>
<div class="py-4">
<button
    type="submit"
    class=""
    w-full
    py-4
    px-2
    bg-indigo-600
    hover:bg-indigo-700
    text-white
    rounded
    font-bold
    text-lg
    "
    >
    Reset password
</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="/signin" class="underline">Remember your password?</a>
    </p>
</div>
</form>
</div>
</div>
{{template "footer" .}}

```

We can use the **diff** tool to compare this with the **signin.gohtml** template to compare how they differ.

```
diff templates/signin.gohtml \
templates/forgot-pw.gohtml
```

The **diff** command may not work in windows, but there are free tools online

for diffing two source files. Feel free to explore them and find one to compare the two files.

Next we need to update `main.go` to parse this template.

```
code main.go
```

Find the section in the `main()` function where other templates are being parsed and add the following.

```
usersC.Templates.ForgotPassword = views.Must(views.ParseFS(
    templates.FS,
    "forgot-pw.gohtml", "tailwind.gohtml",
))
```

Next, head to where our routes are defined and add routes for our forgot password page. One will render the form while another processes it.

```
r.Get("/forgot-pw", usersC.ForgotPassword)
r.Post("/forgot-pw", usersC.ProcessForgotPassword)
```

The handler for processing forgotten passwords won't work quite yet because we haven't finished all of the services code, but we can set this part up anyway.

At this point we have a very subtle bug. If we open the `signin.gohtml` and `signup.gohtml` templates, these both send the user to the `/reset-pw` path to reset their password, but just defined our form and our routes using the `/forgot-pw` path. We will be using `/reset-pw` for the second step of the process, once they have a password reset token, and we need to update the links to send the user to the `/forgot-pw` page when they forget their password.

Let's update the source files.

```
code templates/signin.gohtml
```

Find the “Forgot your password?” link and update the link with the following path.

```
<p class="text-xs text-gray-500">
  <!-- Replace <a href="/forgot-pw" ... > with the updated link -->
  <a href="/forgot-pw" class="underline">Forgot your password?</a>
</p>
```

We need to repeat this with the signup template.

```
code templates/signup.gohtml
```

Find the “Forgot your password?” link and update the path in the link tag.

```
<p class="text-xs text-gray-500">
  <!-- Replace <a href="/forgot-pw" ... > with the updated link -->
  <a href="/forgot-pw" class="underline">Forgot your password?</a>
</p>
```

With our template created and being parsed in `main.go` we can restart the server and verify that the page renders correctly at `/forgot-pw`. We can also click the “Forgot your password?” links now that we have updated their `href` and verify that they direct us to the correct page.

Note: Submitting the forgot password form will not work until we finish up some additional code.

Source Code

- [Source Code](#) - see the final source code for this lesson.
- [Diff](#) - see the changes made in this lesson.

19.6 Initializing Services with ENV Vars

Our `Users` type in the `controllers` package has both the `EmailService` and the `PasswordResetService` fields declared, but these are never assigned a value when we instantiate the `Users` type in `main.go`. In this lesson we are going to focus on initializing these services using ENV variables that we learned about earlier. While doing this, we can perform some of the prep-work to read all of our configuration from ENV variables.

Our first step will be defining all of the data our application needs to start up in a `config` type. After that we will create a function to set this type up using our hard-coded values, and then we can gradually replace those with values read from ENV variables.

We will be coding this in our `main.go` source file.

```
code main.go
```

Near the top of the source file add the new `config` type. This needs to have config information for:

- Postgres
- SMTP
- CSRF
- Our web server

We can reuse any existing config types we have for our first pass at the code. For others we can create an inline struct to get started.

```
type config struct {
    PSQL models.PostgresConfig
    SMTP models.SMTPConfig
    CSRF struct {
        Key     string
        Secure bool
    }
    Server struct {
        Address string
    }
}
```

Next we need a way to load a config. It is generally a good idea to have a function like this return an error as well just in case our config loading process encounters an error of any sort.

```
func loadEnvConfig() (config, error) {
    var cfg config
    err := godotenv.Load()
    if err != nil {
        return cfg, err
    }
    // TODO: Setup PSQL config
    // TODO: Setup SMTP config
    // TODO: Setup CSRF config
    // TODO: Setup Server config
    return cfg, nil
}
```

Long term this will use ENV variables for every config value, but for now we will use some hard-coded values until we verify it is working, and we can add TODOs to our code to remind us to update it later. This allows us to change fewer things in our code at a time, helping avoid bugs that are hard to debug.

Replace the TODOs in our `loadEnvConfig` function with the code below.

```
// TODO: Read the PSQL values from an ENV variable
cfg.PSQL = models.DefaultPostgresConfig()

cfg.SMTP.Host = os.Getenv("SMTP_HOST")
```

```

portStr := os.Getenv("SMTP_PORT")
cfg.SMTP.Port, err = strconv.Atoi(portStr)
if err != nil {
    return cfg, err
}
cfg.SMTP.Username = os.Getenv("SMTP_USERNAME")
cfg.SMTP.Password = os.Getenv("SMTP_PASSWORD")

// TODO: Read the CSRF values from an ENV variable
cfg.CSRF.Key = "gFvi45R4fy5xNBlnEeZtQbfAVCYEIAUX"
cfg.CSRF.Secure = false

// TODO: Read the server values from an ENV variable
cfg.Server.Address = ":3000"

```

Most of this code is assigning hard-coded values, but we do have a bit of code that reads the SMTP config from ENV variables. We will discuss the naming schema used for ENV variables later in this lesson.

Now we can go back to our `main()` function and update it to utilize this config. Let's start by loading the config.

```

func main() {
    cfg, err := loadEnvConfig()
    if err != nil {
        panic(err)
    }
    // ...
}

```

Next up is our Postgres connection. We want to read it form the config file we loaded.

```

func main() {
    // ...

    // Setup the database
    db, err := models.Open(cfg.PSQL)
    if err != nil {
        panic(err)
    }
}

```

```
    defer db.Close()

    // ...
}
```

After that we can update our CSRF middleware to read from the config.

```
csrfMw := csrf.Protect(
    []byte(cfg.CSRF.Key),
    csrf.Secure(cfg.CSRF.Secure),
)
```

The last config change we need to make is updating the code that starts our web server. This should use the address provided in the config. If we haven't already, we can also handle the error from the ListenAndServe function call.

```
// Start the server
fmt.Printf("Starting the server on %s...\n", cfg.Server.Address)
err = http.ListenAndServe(cfg.Server.Address, r)
if err != nil {
    panic(err)
}
```

Now we can add code to instantiate our password reset service and the email service. The latter will use the SMTP config we are reading from ENV variables.

```
// Setup services
userService := models.UserService{
    DB: db,
}
sessionService := models.SessionService{
    DB: db,
}
pwResetService := models.PasswordResetService{
    DB: db,
}
emailService := models.NewEmailService(cfg.SMTP)
```

Once these services are initialized we can assign them to the correct fields in the users controller.

```
usersC := controllers.Users{
    UserService:           &userService,
    SessionService:        &sessionService,
    PasswordResetService: &pwResetService,
    EmailService:          emailService,
}
```

Our controllers expect pointers for every service, so previously we have been using the `&` to get a pointer to each of our services. The `NewEmailService` function returns a pointer already, so this service stands out from the rest because it doesn't have the `&` in front of it. We can make this more uniform by adding the `&` in front of our services as we declare them. This will make the variables pointers from the start, which is how we intend to use our service types anyway.

Head back up to where we declare our services and update the code. We are adding the `&` in front of the `models` word for the user service, the session service, and the password reset service.

```
// Setup services
userService := &models.UserService{
    DB: db,
}
sessionService := &models.SessionService{
    DB: db,
}
pwResetService := &models.PasswordResetService{
    DB: db,
}
emailService := models.NewEmailService(cfg.SMTP)
```

If using an editor that shows errors, we should see quite a few errors in our code. This can help us find the spots in our code that need updated do to this change.

We then need to head back down to where we declare the user middleware and update the code to reflect our changes.

```
// Setup middleware
umw := controllers.UserMiddleware{
    SessionService: sessionService,
}
```

Next are the controllers updates. We can now assign variables to all of our services without needing the & prefix in front of any of them.

```
// Setup controllers
usersC := controllers.Users{
    UserService:           userService,
    SessionService:        sessionService,
    PasswordResetService: pwResetService,
    EmailService:          emailService,
}
```

Our code will now compile, but it is attempting to read the SMTP config via ENV variables, and we haven't set that up. Long term, our goal is to have all of the config read from ENV variables. The naming of the ENV variables will be in the format **NAME_OF_KEY**, where we use uppercase letters, break up words with underscores. We will also try to have these match the structs and nesting of the config with shared prefixes. As an example, our SMTP port can be provided via the ENV variable **SMTP_PORT**. Our server address would use the key SERVER_ADDRESS since this matches our config.

ENV variable naming isn't a strict rule, but it is a very common format and a good guideline for consistency.

For SMTP we will need the following variables:

```
# SMTP connection info
SMTP_HOST=
SMTP_PORT=
SMTP_USERNAME=
SMTP_PASSWORD=
```

If we wanted to provide all of our application's configuration via ENV variables, we might add the following keys to both our `.env` file and our ENV template.

```
# Postgres database connection info.  
PSQL_HOST=  
PSQL_PORT=  
PSQL_USER=  
PSQL_PASSWORD=  
PSQL_DATABASE=  
PSQL_SSL_MODE=  
  
# CSRF configs  
CSRF_KEY=  
CSRF_SECURE=  
  
# Server configs  
SERVER_ADDRESS=
```

We will read all the cfg fields from the `.env` file in the future before deploying. Our primary goal for now was to use it for the SMTP credentials, and to limit the number of changes otherwise to help avoid bugs while transitioning to using this new config type.

Source Code

- [Source Code](#) - see the final source code for this lesson.
- [Diff](#) - see the changes made in this lesson.

19.7 Check Your Email HTML Template

We need a template to render after the forgot password form has been submitted. It doesn't have to be anything fancy, we just need something that tells the user we sent them an email with password reset instructions. Let's create a file for the template.

```
code templates/check-your-email.gohtml
```

Add the following code to the file.

```
{{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">
      Check your email
    </h1>
    <p class="text-sm text-gray-600 pb-4">An email has been sent to the email address {{.Email}}
    </div>
  </div>
{{template "footer" .}}
```

We then need to update `main.go` to ensure this template gets parsed.

```
code main.go
```

Find the section where templates are being parsed and add the following:

```
usersC.Templates.CheckYourEmail = views.Must(views.ParseFS(
  templates.FS,
  "check-your-email.gohtml", "tailwind.gohtml",
))
```

Aside 19.2. Naming inconsistency

Some of you may notice that I have the templates on the users controller named `SignIn` and `SignUp`, but those files are named `signin.gohtml` and `signup.gothml`. On these new files I opted to put a dash between words in

the template filename. That is, we have `forgot-pw` and `check-your-email` for the templates named ForgotPassword and CheckYourEmail.

I apologize that this isn't 100% consistent. I have a habit of fixing these things up as they become apparent in my code, but it felt like a waste of time to have everyone read a lesson on renaming source files, so I'll leave them be for now. Feel free to do that on your own time if it bugs you.

We haven't finished initializing our services, so we cannot test this template as part of its normal workflow. What we can do is temporarily render it in an HTTP handler, then undo those changes. Let's do this using the ForgotPassword HTTP handler.

`code controllers/users.go`

Find the ForgotPassword function and update it to render our CheckYourEmail template.

```
func (u *Users) ForgotPassword(w http.ResponseWriter, r *http.Request) {
    var data struct {
        Email string
    }
    data.Email = r.FormValue("email")
    // Be sure to revert this back to the ForgotPassword template
    // before proceeding to the next lesson.
    u.Templates.CheckYourEmail.Execute(w, r, data)
}
```

Now we can visit <http://localhost:3000/forgot-pw?email=user%40example.com> and view the template with an email address. Be sure to include the email URL query param, otherwise this won't work.

Once the template has been verified, undo the change to the ForgotPassword function. We want that page to render the ForgotPassword template.

Source Code

- [Source Code](#) - see the final source code for this lesson.
- [Diff](#) - see the changes made in this lesson.

19.8 Reset Password HTTP Handlers

Now we need to verify that the second half of our password reset process is designed well. We can do this by creating the HTTP handlers and HTML pages used to reset a user's password once a reset token has been sent to them.

```
code controllers/users.go
```

First we need to add the new `ResetPassword` template we will be using to render the form. This will be added to the users controller.

```
type Users struct {
    Templates struct {
        New           Template
        SignIn        Template
        ForgotPassword Template
        CheckYourEmail Template
        ResetPassword Template
    }
    UserService      *models.UserService
    SessionService   *models.SessionService
    ResetTokenService *models.ResetService
    EmailService     *models.EmailService
}
```

Next we add a handler to render the form. This is similar to our forgotten password handler, except this time we are parsing a token from the URL query parameters, not an email address. We will be inserting this token into the form as a hidden value, so it is important to make sure we are reading it correctly.

```
func (u Users) ResetPassword(w http.ResponseWriter, r *http.Request) {
    var data struct {
        Token string
    }
    data.Token = r.FormValue("token")
    u.Templates.ResetPassword.Execute(w, r, data)
}
```

Next, we need a handler to process this form. The form will have two fields:

- A token field
- A password field for the new password

Let's start by processing those fields.

```
func (u Users) ProcessResetPassword(w http.ResponseWriter, r *http.Request) {
    var data struct {
        Token     string
        Password string
    }
    data.Token = r.FormValue("token")
    data.Password = r.FormValue("password")
}
```

With the token and password parsed we can proceed with the password reset process. To do this we will:

1. Attempt to consume the token.
2. Update the user's password.
3. Create a new session.
4. Sign the user in.

5. Redirect them to the /users/me page.

That gives us the following code, minus a few TODOs that we will want to fill in later.

```
func (u Users) ProcessResetPassword(w http.ResponseWriter, r *http.Request) {
    var data struct {
        Token     string
        Password string
    }
    data.Token = r.FormValue("token")
    data.Password = r.FormValue("password")

    user, err := u.PasswordResetService.Consume(data.Token)
    if err != nil {
        fmt.Println(err)
        // TODO: Distinguish between server errors and invalid token errors.
        http.Error(w, "Something went wrong.", http.StatusInternalServerError)
        return
    }
    // TODO: Update the user's password.

    // Sign the user in now that they have reset their password.
    // Any errors from this point onward should redirect to the sign in page.
    session, err := u.SessionService.Create(user.ID)
    if err != nil {
        fmt.Println(err)
        http.Redirect(w, r, "/signin", http.StatusFound)
        return
    }
    setCookie(w, CookieSession, session.Token)
    http.Redirect(w, r, "/users/me", http.StatusFound)
}
```

If we are able to create a session, we will sign the user in after they update their password. This avoids the unnecessary step of having them log in again with the password they just set. If creating that session fails for any reason, we will redirect the user to the sign in page to allow them to try signing in with the updated password.

It looks like our password reset service is working well as designed. We don't have a way to reset a user's password yet, which we could do in two ways:

1. Pass the password into the **Consume** function as second argument.
2. Add a function to the UserService that will allow us to update a user's password and call that.

Either option would work, but we are going to go with the second. We will get to that in a future lesson. Next we will focus on creating the template for the password reset process, then we can implement our password reset service methods, and finally add a function to the user service that allows us to reset a user's password.

Source Code

- [Source Code](#) - see the final source code for this lesson.
- [Diff](#) - see the changes made in this lesson.

19.9 Reset Password HTML Template

Our next task is to create a form for resetting a user's password. We start by creating the source file for the template.

```
code templates/reset-pw.gohtml
```

Like our other forms, this will be very similar to the sign in page, but it will have different text, and the input for the reset token will be hidden.

```
{{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">
      Reset your password
    </h1>
  </div>
</div>
```

```
</h1>
<form action="/reset-pw" method="post">
  <div class="hidden">
    {{csrfField}}
  </div>
  <div class="py-2">
    <label for="password" class="text-sm font-semibold text-gray-800">
      New 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
    >
    autofocus
  </>
  </div>
  <div class="hidden">
    <input type="hidden" id="token" name="token" value="{{.Token}}"/>
  </div>
  <div class="py-4">
    <button
      type="submit"
      class=""
      w-full
      py-4
      px-2
      bg-indigo-600
      hover:bg-indigo-700
      text-white
      rounded
      font-bold
      text-lg
    >
      Update password
    </button>
  </div>
  <div class="py-2 w-full flex justify-between">
    <p class="text-xs text-gray-500">
      <a href="/signup" class="underline">Sign up</a>
```

```

    </p>
    <p class="text-xs text-gray-500">
        <a href="/signin" class="underline">Sign in</a>
    </p>
    </div>
    </form>
    </div>
</div>
{{template "footer" .}}

```

For the most part this code should work, but there is one edge case to discuss. What happens if the token isn't provided as a URL query parameter?

If we rendered the HTML template we have, the token input will remain hidden, and there will be no way to submit the form correctly. It will always have an invalid token input. This sounds very much like a bug that we should handle.

There are two possible ways to handle this case:

1. Render an error message stating that the token was not read from the URL.
2. Render an input field requesting the password reset token.

We are going to go with the latter, as this will allow a user to reset their password as long as they have the reset token and can input it manually. To do this, we need to replace the token input in our HTML template with an if/else statement that checks whether the token is present. If it is present, we use the code we had. If it is not present, we render a label and a visible input for the password reset token.

```

{{if .Token}}
    <div class="hidden">
        <input type="hidden" id="token" name="token" value="{{.Token}}" />
    </div>
{{else}}
    <div class="py-2">

```

```
<label for="token" class="text-sm font-semibold text-gray-800">
  Password Reset Token</label>
>
<input
  name="token"
  id="token"
  type="text"
  placeholder="Check your email"
  required
  class="
    w-full
    px-3
    py-2
    border border-gray-300
    placeholder-gray-500
    text-gray-800
    rounded
  "
/
>
</div>
{{end}}
```

As long as we generate our reset password URLs correctly and the user clicks on the link, we should pretty much always render the hidden input field, but just in case this doesn't work we have a backup plan in our code.

Next we will head to `main.go` to make sure this template is being parsed.

```
code main.go
```

Find the code where templates are being parsed and add the following.

```
usersC.Templates.ResetPassword = views.Must(views.ParseFS(
  templates.FS,
  "reset-pw.gohtml", "tailwind.gohtml",
))
```

After that, navigate to where routes are declared in our `main()` function and add the following routes:

```
r.Get("/reset-pw", usersC.ResetPassword)
r.Post("/reset-pw", usersC.ProcessResetPassword)
```

Finally, restart your server and verify it works. To do this we want to visit the following urls:

- <http://localhost:3000/reset-pw?token=abc123> - This should render the password reset form with the reset token input hidden.
- <http://localhost:3000/reset-pw> - This should render the password reset form with the reset token input visible.

We can also inspect the source code for the hidden token input to verify it is being set correctly.

Source Code

- [Source Code](#) - see the final source code for this lesson.
- [Diff](#) - see the changes made in this lesson.

19.10 Update Password Function

Next we are going to add a function that allows us to update a user's password. This needs to part of a service, and we are going to add it to the UserService.

```
code models/user.go
```

We are going to add a new method named **UpdatePassword** to the UserService. In this method we are going to hashed password using **bcrypt**, then we will write a SQL query to update the user's password hash.

```
func (us *UserService) UpdatePassword(userID int, password string) error {
    hashedBytes, err := bcrypt.GenerateFromPassword([]byte(password), bcrypt.DefaultCost)
    if err != nil {
        return fmt.Errorf("update password: %w", err)
    }
    passwordHash := string(hashedBytes)
    _, err = us.DB.Exec(`UPDATE users
        SET password_hash = $2
        WHERE id = $1;`, userID, passwordHash)
    if err != nil {
        return fmt.Errorf("update password: %w", err)
    }
    return nil
}
```

We can now utilize the UpdatePassword method in the password reset handler. This is in the users controller, so open that file.

code controllers/users.go

Find the function to process reset passwords and add the following where we had our TODO.

```
func (u Users) ProcessResetPassword(w http.ResponseWriter, r *http.Request) {
    // ...

    err = u.UserService.UpdatePassword(user.ID, data.Password)
    if err != nil {
        fmt.Println(err)
        http.Error(w, "Something went wrong.", http.StatusInternalServerError)
        return
    }

    // ...
}
```

We now have everything except for the PasswordResetService implemented, so it is a pretty safe bet the methods we stubbed will work well as designed.

We will focus on implementing the Create and Consume methods for password reset tokens to finalize the password reset feature.

Source Code

- [Source Code](#) - see the final source code for this lesson.
- [Diff](#) - see the changes made in this lesson.

19.11 PasswordReset Creation

We are finally ready to implement the ResetService that we stubbed out earlier in this section. Our design appears to work well, and this is the final step in finishing the password reset feature.

Implementing the **Create** function is going to involve using a lot of what we learned previously in the course, so most of it should look familiar. In many cases it might feel like we are duplicating some code, which is a perfectly valid thing to do early on. If it turns out we are duplicating the same code all the time then we can consider ways to extract that logic into something more reusable, like a helper function.

Open up the source file for our password reset service and head to the **Create** function.

```
code models/password_reset.go
```

The first thing we want to do in the **Create** function is verify that the email address is associated with a user, and lookup who that user is. We do this by normalizing the email address (making it lowercase), and then searching for the ID of that user. If we find an ID, we have a valid email address. If we cannot find an ID for that email, then it isn't a valid email address in our system.

Not only will this help us ensure that a user with the email address provided exists, but we will use the ID later when we create the password reset token.

```
func (service *PasswordResetService) Create(email string) (*PasswordReset, error) {
    // Verify we have a valid email address for a user
    email = strings.ToLower(email)
    var userID int
    row := service.DB.QueryRow(`SELECT id FROM users WHERE email = $1;`, email)
    err := row.Scan(&userID)
    if err != nil {
        // TODO: Consider returning a specific error when the user does not exist.
        return nil, fmt.Errorf("create: %w", err)
    }
}
```

When we encounter an error from the database query, we aren't checking to see what type of error occurred. It could be a database connection issue, or it could be an error about no records being found. Long term we may want to return a user-friendly error depending on what went wrong. A user not being found is very different than if our database being unreachable in this context. Since we know this is something we may want to address later, we can add a TODO to remind us.

If there isn't an error we know we have a valid email address. We can then make sure we have a valid number of bytes per token and create a token. This is pretty much identical to the session service code.

```
func (service *PasswordResetService) Create(email string) (*PasswordReset, error) {
    // ...

    // Build the PasswordReset
    bytesPerToken := service.BytesPerToken
    if bytesPerToken == 0 {
        bytesPerToken = MinBytesPerToken
    }
    token, err := rand.String(bytesPerToken)
    if err != nil {
        return nil, fmt.Errorf("create: %w", err)
    }
}
```

The next thing we need to do is hash the token. Again, we can use code nearly identical to what we wrote in our session service, except we will be adding a method to the PasswordResetService.

```
func (service *PasswordResetService) hash(token string) string {
    tokenHash := sha256.Sum256([]byte(token))
    return base64.URLEncoding.EncodeToString(tokenHash[:])
}
```

Note: Long term we might opt to extract the hashing functionality into another function, or a method on a more general type, but for now duplicating the code once isn't bad. It becomes more concerning when we start duplicating it 3+ times.

The last field we need for a **PasswordReset** is a valid **ExpiresAt** timestamp. We create this by using the current time and adding the duration for the password reset token to that time. This gives us a new time that represents when the token expires. Eg “1pm plus 1 hour = 2pm”.

We have more default values to check, so we will write code similar to what we wrote with the **bytesPerToken** to get the duration.

```
func (service *PasswordResetService) Create(email string) (*PasswordReset, error) {
    // ...

    duration := service.Duration
    if duration == 0 {
        duration = DefaultResetDuration
    }
}
```

After that we are ready to construct expiration time and the rest of our PasswordReset.

```
func (service *PasswordResetService) Create(email string) (*PasswordReset, error) {
    // ...

    pwReset := PasswordReset{
        UserID:     userID,
        Token:      token,
        TokenHash:  service.hash(token),
        ExpiresAt: time.Now().Add(duration),
    }
}
```

In our code we are using the `time.Now` function. While this works well for our use case, many larger applications will inject this function into services. If we wanted to do this, it might look like the code below. (*Note: We are not doing this, it is only shown for demonstration purposes.*)

```
type PasswordResetService struct {
    // ...

    // We are NOT doing this, but it is for demonstration purposes
    Now func() time.Time
}
```

We could then treat the `Now` field similar to how we treat the `Duration` field. If it is `nil`, we use the `time.Now` function as a default value, but if it is set we use whatever function was provided. Getting the current time this way ends up being more code, but it can make testing easier when we need to test edge cases. For instance, if we wanted to verify that an expired reset token could not be used, we could provide a `Now` function that gives us a time far enough in the past that our reset token will be expired. This works significantly better than trying to have our code wait for the token to expire, as each machine running tests might be slightly faster or slower than others, and it can lead to flakey tests. This entire topic is covered more in [Test with Go](#), but for now we will continue using the `time.Now` function in our code.

Now that our `PasswordReset` is setup, we can focus on inserting it into our database and returning the results. Again, this will be similar to code in the

session service. We are going to use an `ON CONFLICT` here to overwrite an existing password reset token, and we are going to insert roughly the same fields. The primary difference is the extra code to account for the `expires_at` field.

```
func (rs *ResetService) Create(email string) (*PasswordReset, error) {
    // ...

    // Insert the PasswordReset into the DB
    row = service.DB.QueryRow(`

        INSERT INTO password_resets (user_id, token_hash, expires_at)
        VALUES ($1, $2, $3) ON CONFLICT (user_id) DO
        UPDATE
        SET token_hash = $2, expires_at = $3
        RETURNING id;`, pwReset.UserID, pwReset.TokenHash, pwReset.ExpiresAt)
    err = row.Scan(&pwReset.ID)
    if err != nil {
        return nil, fmt.Errorf("create: %w", err)
    }
    return &pwReset, nil
}
```

We now have a working function to create password reset tokens. There might be a few things we could optimize in our code and our SQL queries, but what we have should work sufficiently. We can also optimize things later if they actually prove to be a bottleneck, rather than spending time on optimizations that might not ever matter.

We can restart our web application and create a new password reset token by submitting a valid email address to the forgot password form. We should see an email in mailtrap (or whatever service you used) that has a URL with the token as a URL query parameter.

Source Code

- [Source Code](#) - see the final source code for this lesson.
- [Diff](#) - see the changes made in this lesson.

19.12 Implementing Consume

The last step in finishing up password resets is to implement the **Consume** function for the password reset service. In this function we need to do three main things:

1. Validate that that the token is valid and hasn't expired.
2. Lookup the user associated with the token so we can return that information.
3. Delete the token so that it cannot be used again.

A token is considered valid if we can find it in the database, so we can handle that with a SQL query. Once we find a password reset we can validate whether it has expired or not by checking the **expires_at** field and verifying that the timestamp isn't in the past.

Open up the password reset source file in our models package.

```
code models/password_reset.go
```

Head down to the **Consume** function, and add the following code to query for both a reset token and the associated user. We could technically use the **UserService** to lookup the user information, but doing it with a SQL JOIN reduces the number of queries we need to make.

```
func (service *PasswordResetService) Consume(token string) (*User, error) {
    tokenHash := service.hash(token)
    var user User
    var pwReset PasswordReset
    row := service.DB.QueryRow(`
```

SELECT password_resets.id, password_resets.expires_at,

```
        SELECT * FROM users  
        WHERE id = ?`)
```

```

        users.id,
        users.email,
        users.password_hash
    FROM password_resets
        JOIN users ON users.id = password_resets.user_id
    WHERE password_resets.token_hash = $1;`, tokenHash)
err := row.Scan(
    &pwReset.ID, &pwReset.ExpiresAt,
    &user.ID, &user.Email, &user.PasswordHash)
if err != nil {
    return nil, fmt.Errorf("consume: %w", err)
}
}

```

After performing our query we scan the results into a PasswordReset and a User so we have access to both moving forward.

Now that we have all the information we need, we need to validate that the reset token hasn't expired. We do this by taking the expiration time for the reset token and checking if the current time is after that time. If it is, then it is a date in the past and the password reset has expired.

```

if time.Now().After(pwReset.ExpiresAt) {
    return nil, fmt.Errorf("token expired: %v", token)
}

```

This is another example of a situation where we may eventually want a more specific error so that our frontend can render a better error message. For instance, we might tell the user, "That password reset token has expired. Please request another by submitting the forgotten password form." For now we are going to focus on just getting the password reset process working, but keep this in mind as something we can incrementally improve on later.

If our password reset is valid, we need to delete it in order to consume it. We don't currently have a way to delete password resets, so let's add a function that will delete a password reset via its ID.

```
func (service *PasswordResetService) delete(id int) error {
    _, err := service.DB.Exec(`DELETE FROM password_resets
                                WHERE id = $1;`, id)
    if err != nil {
        return fmt.Errorf("delete: %w", err)
    }
    return nil
}
```

We don't have any specific need to export this function, so we will keep it unexported for now.

Head back to the **Consume** function and finish it up by calling our **delete** function, handling the error, and finally returning the user associated with the password reset token we just consumed.

```
func (service *PasswordResetService) Consume(token string) (*User, error) {
    // ...

    err = service.delete(pwReset.ID)
    if err != nil {
        return nil, fmt.Errorf("consume: %w", err)
    }
    return &user, nil
}
```

With our code finished, we need to test everything. Restart the application and test the password reset process. Submit a valid email address into the forgot password form, verify that an email is sent, then copy the URL from the email. Update it to point to <http://localhost:3000> instead of **lenslocked.com**, as we want to test this in development. Be sure to keep the path and the token in the URL query parameters.

Once the password reset page is rendered, try entering a new password. If successful, we should be redirected to the **/users/me** page. We can then sign out and verify that the new password works for signing in, and the old password no longer works.

Aside 19.3. Bug in the screencasts

In the screencasts, I forgot to update the path in the password reset form template when we created it. In the written version of the course I caught and updated the code to address this.

If you have been following the screencasts, open the `reset-pw.gohtml` template and update the HTML form to submit to the `/reset-pw` path instead of `/signin`. This is covered in the videos, but I am mentioning it here in case someone is following the written lesson after using the videos previously.

Source Code

- [Source Code](#) - see the final source code for this lesson.
- [Diff](#) - see the changes made in this lesson.

19.13 Password Reset Exercises

19.13.1 Ex1 - Implement passwordless sign-in

In the first lesson of the section 18 we discussed how the password reset process works. We also talked about how passwordless sign-in (sometimes called magic links) uses a similar process.

For this exercise, create an option for users to sign into the application without entering a password, but instead by using a link that gets emailed to them.

Also, think about why this process is still secure compared to using a password. If you have questions or concerns, feel free to discuss them in Slack.

19.13.2 Ex2 - Add an option to update a user's email address

A user can now update their password via the lost password form, but they have no way to update their email address. Take what we learned so far in the course and add a page where the user can see their account's current email address and can update it with a new email address.

Chapter 20

Better Error Handling

20.1 Inspecting Errors

In this section we are going to explore ways to improve our error handling. While we will be looking at code, the goal in this section is to understand the various ways of handling errors so that we have the proper tools to handle a variety of situations in the future.

What does that mean exactly?

You have probably seen code like below:

```
if err != nil {  
    return err  
}
```

Or perhaps you have seen the error being wrapped with a bit of context prior to being returned.

```
if err != nil {
    return fmt.Errorf("creating user: %w", err)
}
```

In both of these cases we aren't really handling the error. We may add some context to it, which is a great start, but we aren't looking at the error and deciding if we can do anything about it. We are simply returning the error so that it can be handled elsewhere.

Sometimes this is exactly what we want to do. We cannot always know how to properly handle an error without additional context, and sometimes we cannot do anything about an error. In those cases, returning the error is an appropriate response. In other circumstances this may not be true.

Imagine that we were writing code to send an email to a user. During our first attempt at writing the code, we might do some error checking, and if there is an error we return it with some context.

```
err := emailService.Send(welcomeEmail)
if err != nil {
    return fmt.Errorf("sending welcome email: %w", err)
}
```

This is a completely valid way of approaching the code at first, but over time we should start to ask ourselves, “Can I handle any of the errors that I might encounter here?” For instance, if we can check if an error is a connection error, could we add some code to retry after a short delay?

There are many circumstances where handling the error could still fail. For instance, if our email service provider is having a major outage, retrying after a short delay isn't likely to fix the issue. In those cases we may still end up returning an error, but if it was a temporary outage this type of retry could avoid the need to return an error at all, and our code could continue to operate as desired.

Another example is an error we will see when attempting to create a user with an existing email address.

```
ERROR: duplicate key value violates unique constraint "users_email_key" (SQLSTATE 23505)
```

While this error might be useful in the SQL context, it is not very useful for an end user who is trying to sign up for an account. In this case handling the error might simply be translating it into a user-friendly error that will help them decide how to proceed. For instance, we might tell the user, “This email address is already in use, please try another email address or sign in to your existing account.”.

Handling an error like this requires the correct context, as well as knowledge of what error has occurred. In this particular instance, we would need to handle the error somewhere in our code where we know that a user is being created, and then we would need to parse some additional information from the error to see if we could render a more helpful message to our end user.

There are a few common ways to determine additional information from an error. Let’s look at examples starting with checking for equality.

The `database/sql` package has the `ErrNoRows` variable exported. This is an error variable that is returned whenever a `QueryRow` function call returns no records. If we encounter an error when calling `QueryRow`, we can check for this specific error and handle that case accordingly.

For instance, when a user is signing in we might want to query for their account before verifying the password they typed is correct. If we were to query for a user with a specific email address and the `ErrNoRows` error was returned, this would imply that no user account exists with that email address and we could tell the user that they typed their email address incorrectly.

```
err := db.QueryRow("SELECT * FROM users WHERE email=$1;", email)
if err == sql.ErrNoRows {
    // Render the signin page with an error telling the user
```

```
// that no account exists with that email address
}
```

We can also check to see if an error has additional methods. We achieve this using Go's [type assertion](#), along with an interface defining the methods we are looking for. In the example below we are looking to see if our error has the **Temporary** method which might indicate that the error is temporary - like a temporary server outage.

```
type temporary interface {
    Temporary() bool
}

err := doSomething()
if err != nil {
    te, ok := err.(temporary)
    if !ok {
        return err
    }
    if te.Temporary() {
        // Do something special if it is a temporary error
    }
}
```

With methods like **Temporary** we can gain additional information that might help us decide how to handle the error. In the case of a temporary outage, we might opt to use our pause and retry strategy since there is a chance the temporary outage will be resolved.

Aside 20.1. How do I know what interfaces to test?

If a package returns errors with additional information like this **Temporary** method, there will typically be an error type or something else noted in the docs to indicate what methods should be checked for.

We can use a similar technique to see if an error has a specific type. The [io/fs](#) package can return the [PathError](#) type.

```
type PathError struct {
    Op   string
    Path string
    Err  error
}
```

When using the [io/fs](#) package, we can check to see if this was the type of error we received and then use the additional fields offered by that type.

```
err := fs.ReadDir(...)
if err != nil {
    pe, ok := err.(*fs.PathError)
    if !ok {
        return err
    }
    // We can use things like pe.Path now
}
```

Finally, if all other techniques fail to determine what the error message is about and we need additional information, we can look at the error message and try to parse the information we need from the error string.

```
err := someFunction()
if err != nil {
    if strings.Contains("invalid filename", err.Error()) {
        // We have an invalid filename error and can handle
        // it in a different way if we need to.
    }
}
```

Generally speaking, parsing an error message as a string is the least preferable option, but it will work when no other options are viable.

20.2 Inspecting Wrapped Errors

While all of the techniques we just learned worked in the past, error wrapping has changed things a bit.

Wrapping occurs when an error is wrapped around an existing error to add context or some additional information, and it provides an `Unwrap()` method to obtain the original error. When we call `fmt.ErrorOf` and pass in the original error with a message, we are wrapping that error. We are also changing the value of that error variable, which means that if we were to try to compare a wrapped error to a known one, it wouldn't match. Below is an example.

```
var wrappedErr error = fmt.Errorf("signing in: %w", sql.ErrNoRows)
if wrappedErr == sql.ErrNoRows {
    // This will never be true.
}
```

Similarly, using a type assertion on an error can fail for the same reason. An error may be wrapped, and the new error created may not have the methods we are testing for with our assertion.

To handle both of these situations we need a way to unwrap the error. This will allow us to check any of the wrapped errors to see if any of them match the error qualities we are looking for. We can do this via the `Unwrap` function in the errors package.

```
func Unwrap(err error) error
```

Behind the scenes this function is doing a type assertion on the error to see if it implements the following interface.

```
interface {
    Unwrap() error
}
```

If the error implements the `Unwrap` method, the result of calling `Unwrap` is returned. Otherwise if the error cannot be unwrapped `nil` is returned.

Using the `Unwrap` function we can manually check to see if an error matches a specific error type.

```
err := demo()
tmpErr := err
for tmpErr != nil {
    if tmpErr == sql.ErrNoRows {
        // Do something special in this case
    }
    tmpErr = errors.Unwrap(tmpErr)
}
```

Doing this every time we needed to check an error would be pretty annoying, so Go's `errors` package also provides us with the `errors.Is` and `errors.As` functions. These are helpers that do a lot of the nested unwrapping work for us.

`Is` will check if an error or any error it wraps matches a specific error value. We can see this by wrapping the `sql.ErrNoRows` error and calling the `Is` function.

```
err := fmt.Errorf("wrapped demo: %w", sql.ErrNoRows)
if errors.Is(err, sql.ErrNoRows) {
    // This will be true!
}
```

`As` works similarly, but it will check if the error or any of the wrapped errors match a specific type. Below is an example using an interface, but we can also test to see if the error matches a struct type.

```

type temporary interface {
    Temporary() bool
}

err := doSomething()
var te temporary
if errors.As(err, &te) {
    if te.Temporary() {
        // Do something special if it is a temporary error
    }
}
return err

```

If any of the wrapped errors successfully match the type we are looking for, it will be assigned to the second argument passed into the `As` function call. This means we need to provide a pointer to our variable as the second argument, which can be a bit confusing if the type we are trying to match is also a pointer. In these cases we end up passing in a pointer to a pointer.

```

err := fs.ReadDir(...)
if err != nil {
    var pe *fs.PathError
    if errors.As(err, &pe) {
        // use path error here
    }
}
return err

```

Rather than using any custom code to test for error equality, it is best to use `errors.As` and `errors.Is`. This will ensure that our code continues to work with wrapped errors.

20.3 Designing the Alert Banner

Now that we know how to inspect our errors, we can begin to look at our errors and decide either how we want to react. Sometimes we might be able to try

to fix an error via code - like retrying something - but in many cases we will encounter errors that we cannot fix. In these cases, we will frequently want to translate the errors into a user-friendly message explaining what went wrong to our end user. This is especially true when it is an error caused by invalid input from the user, such as providing an invalid email address.

The first step we will take towards our goal of displaying more user-friendly errors is to update our HTML template to include an error field that we can populate with error information. We want this to be available on any page we render, so we are going to update our Tailwind layout template.

```
code templates/tailwind.gohtml
```

We are going to add our alerts below the `<header>` tag, and we are going to include the HTML for two alerts to start just to verify that it looks okay with two errors.

```
{{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 class="h-screen bg-gray-100">
    <header class="bg-gradient-to-r from-blue-800 to-indigo-800 text-white">
        <!-- ... -->
    </header>
    <!-- Alerts -->
    <div class="py-4 px-2">
        <div class="flex bg-red-100 rounded px-2 py-2 text-red-800 mb-2">
            <div class="flex-grow">
                The email address you provided is already associated with an account.
            </div>
            <a href="#">
                <svg xmlns="http://www.w3.org/2000/svg" fill="none" viewBox="0 0 24 24" stroke-width="1px">
                    <path stroke-linecap="round" stroke-linejoin="round" d="M9.75 9.75l4.5 4.5m0-4.5l-4.5-4.5" />
                </svg>
            </a>
        </div>
    </div>
</body>
</html>
```

```

</div>

<div class="flex bg-red-100 rounded px-2 py-2 text-red-800 mb-2">
  <div class="flex-grow">
    Something went wrong.
  </div>
  <a href="#">
    <svg xmlns="http://www.w3.org/2000/svg" fill="none" viewBox="0 0 24 24" stroke-width="1"
      <path stroke-linecap="round" stroke-linejoin="round" d="M9.75 9.75l4.5 4.5m0-4.5l-4.5"/>
    </svg>
  </a>
</div>
</div>
{{end}}

```

The HTML uses the `x` SVG icon from <https://heroicons.com/>, which will later be used to close the error message. While these are statically coded now, we will later start to render these dynamically when we have an error to show to a user.

Source Code

- [Source Code](#) - see the final source code for this lesson.
- [Diff](#) - see the changes made in this lesson.

20.4 Dynamic Alerts

We want to use errors in our app, not hard coded values.

```

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() (template.HTML, error) {

```

```
        return "", fmt.Errorf("currentUser not implemented")
    },
    "errors": func() []string {
        return []string{
            "Don't do that!",
            "The email address you provided is already associated with another account.",
            "Something went wrong.",
        }
    },
}
// ...
}
```

```
{{if errors}}
<div class="py-4 px-2">
{{range errors}}
<div class="flex bg-red-100 rounded px-2 py-2 text-red-800 mb-2">
<div class="flex-grow">
{{.}}
</div>
<a href="#">
<svg xmlns="http://www.w3.org/2000/svg" fill="none" viewBox="0 0 24 24" stroke-width=1>
<path stroke-linecap="round" stroke-linejoin="round" d="M6 18L18 6M6 6L12 12" />
</svg>
</a>
</div>
{{end}}
</div>
{{end}}
```

Source Code

- [Source Code](#) - see the final source code for this lesson.
- [Diff](#) - see the changes made in this lesson.

20.5 Removing Alerts with JavaScript

It would be nice to remove the alerts if a user doesn't want to see them. Not necessary, but nice. If you don't care about learning the JS here, just skip the lesson. All we are doing is adding some JS to handle this case.

```
 {{if errors}}
  <div class="py-4 px-2">
    {{range errors}}
      <!-- Add the closeable class here -->
      <div class="closeable flex bg-red-100 rounded px-2 py-2 text-red-800 mb-2">
        <div class="flex-grow">
          {{.}}
        </div>
      <!-- Add the onclick here -->
      <a href="#" onclick="closeAlert(event)">
        <svg xmlns="http://www.w3.org/2000/svg" fill="none" viewBox="0 0 24 24" stroke-width=1>
          <path stroke-linecap="round" stroke-linejoin="round" d="M6 18L18 6M6 6L12 12" />
        </svg>
      </a>
    </div>
    {{end}}
  </div>
{{end}}
```

Now add a JS function to handle this.

```
 {{define "footer"}}
<script>
  function closeAlert(event) {
    let closeable = event.target.closest(".closeable");
    closeable.remove();
    // closeable.classList.add("hidden");
  }
</script>
</body>
</html>
{{end}}
```

Source Code

- [Source Code](#) - see the final source code for this lesson.
- [Diff](#) - see the changes made in this lesson.

20.6 Detecting Existing Emails

Let's start to apply everything we have learned and see if we can detect when a user signs up with an existing email address and show a relevant error message.

Head to the users model.

code models/user.go

In the `Create` method, we want to detect when a new account is trying to be created with an existing email address. We could try querying our DB before we do the insert, and this isn't a bad thing to do anyway, but there is always a small chance that the user is inserted *after* we do that query, but before we perform our own insert.

How? Imagine if the user's network or computer somehow submitted the same web form twice. Our app would be processing two web requests at the same time and the timing could lead to this type of occurrence. Using SQL transactions could help, but for now let's just look at how to examine the error message after an insert.

In an ideal world, the types of errors used in the `pgx` package would be documented. If we can't find that, or if we are unsure on how to proceed, we can print out information about the error, including the error message. We get the type using `Printf` and `%T`, which gives us the type of the variable.

```
func (us *UserService) Create(email, password string) (*User, error) {
    // ...
    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 {
    fmt.Printf("Type = %T\n", err)
    fmt.Printf("Error = %v\n", err)
    return nil, fmt.Errorf("create user: %w", err)
}
return &user, nil
}

```

This will give us output like below if we trigger the error by signing up with an existing email address.

```

Type = *pgconn.PgError
Error = ERROR: duplicate key value violates unique constraint "users_email_key" (SQLSTATE 23505)

```

We could use string matching with the error message to try to determine if this was the error, but let's look at the [pgx](#) driver we use for postgres to see if we can use this [PgError](#) type. With a little bit of searching, we can find the [PgError](#) type that is used in pgx v4.

Looking at the pgx docs we might also find the [pgerrcode](#) package, which looks like it will help us determine what type of error code we have.

Let's install these packages so we can use them.

```

go get github.com/jackc/pgconn
go get github.com/jackc/pgerrcode

```

Aside 20.2. You MUST use pgx v4 for this code to work

At this point in the course we are using pgx v4. While v5 has released, the code was restructured a bit and the [PgError](#) is imported from a different package for

v5. As a result, using pgx v5 with the `pgconn` package we just installed will not work.

Later in the course we will look at updating our dependencies and see how to move to v5, but for now I suggest that you continue to use v4 so your code works as expected.

Now we can update our code to use these packages and determine if we have an error because someone is signing up with an email address that already has an account. Let's first create the exported error variable we are going to return.

```
var (
    // A common pattern is to add the package as a prefix to the error for
    // context.
    ErrEmailTaken = errors.New("models: email address is already in use")
)
```

Next, let's update the error handling to see if we can use this as a PgError.

```
func (us *UserService) Create(email, password string) (*User, error) {
    // ...

    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 {
        // See if we can use this error as a PgError
        var pgError *pgconn.PgError
        if errors.As(err, &pgError) {
            // This is a PgError, so see if it matches a unique violation.
            if pgError.Code == pgerrcode.UniqueViolation {
                // If this is true, it has to be an email violation since this is the
                // only way to trigger this type of violation with our SQL.
                return nil, ErrEmailTaken
            }
        }
    }
    return nil, fmt.Errorf("create user: %w", err)
}
```

```
    }
    return &user, nil
}
```

At the moment this is only a partial upgrade to what we had, but it is a good starting point. We can look at how to improve this as we progress through our code.

Before we finish up, run a `go mod tidy` to tidy up our `go.mod` file.

```
go mod tidy
```

Source Code

- [Source Code](#) - see the final source code for this lesson.
- [Diff](#) - see the changes made in this lesson.

20.7 Accepting Errors in Templates

The next step for presenting more user-friendly errors is updating our templates to accept and render errors as alert messages.

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

update ParseFS and remove the errors

```
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() (template.HTML, error) {
                return "", fmt.Errorf("currentUser not implemented")
            },
            "errors": func() []string {
                return nil
            },
        },
    )
    // ...
}
```

Find the **Execute** method on the **Template** type and update it to take in errors as a variadic parameter. We can also update our comments to document the function we need on an error to use a public message.

```
func (t Template) Execute(w http.ResponseWriter, r *http.Request, data interface{}, errs ...error)
// ...
```

Next, update the FuncMap we create inside the Execute method to use the new arguments to generate a list of error messages.

```
tpl = tpl.Funcs(
    template.FuncMap{
        "csrfField": func() template.HTML {
            return csrf.TemplateField(r)
        },
        "currentUser": func() *models.User {
            return context.User(r.Context())
        },
        "errors": func() []string {
            var errorMessages []string
            for _, err := range errs {
                // TODO: Don't keep this long term - we will see why in a later lesson
                errorMessages = append(errorMessages, err.Error())
            }
        },
    },
)
```

```

        }
        return errorMessages
    },
},
)
}

```

Update the **Template** interface we defined in our controllers to reflect this change.

code controllers/template.go

```

type Template interface {
    Execute(w http.ResponseWriter, r *http.Request, data interface{}, errs ...error)
}

```

We won't have any compilation errors now, but we still need to update our controllers to pass errors into our templates.

code controllers/users.go

Update the **Create** function on the **Users** type to re-render the sign up template when we see the **ErrEmailTaken** error.

```

func (u Users) Create(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.Create(data.Email, data.Password)
    if err != nil {
        u.Templates.New.Execute(w, r, data, err)
        return
    }
    session, err := u.SessionService.Create(user.ID)
}
// ...
}

```

Now we can sign up with a taken email address and see an error message. We still have work to do, but this is a good start.

Source Code

- [Source Code](#) - see the final source code for this lesson.
- [Diff](#) - see the changes made in this lesson.

20.8 Public vs Internal Errors

We typically shouldn't share an error message with an end user without knowing what the error message says. When our application encounters an error, it often isn't intended to be displayed to end users. An example of this is an error when our database connection fails. In that case we might see an error message like below.

```
failed to connect to `host=localhost user=baloo database=lenslocked`:  
dial error (dial tcp 127.0.0.1:5432: connect: connection refused)
```

While this doesn't leak our password, it isn't a message meant for our users, and someone might take advantage of leaked information like this to try to hack our web server. None of this should be made public to our users.

One way to prevent this is to think of errors as being in two classifications: - Errors that we know can be shared with end users. We will call these "public errors". - If an error isn't clearly public, it should be considered internal and not shared with end users. Instead, we should show a more generic error.

There are a number of ways we could implement this in our code. We could have specific error variables, and know how to render each of those to our users. Or we could have custom error types with user-friendly messages, and only render those specific errors to end users.

Let's look at one way of implementing this. We are going to define what interface our error needs to implement in the `views` package. That way any error that provides the functionality we need will work.

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

Add the following type somewhere in the file.

```
// We will use this to determine if an error provides the Public method.
type public interface {
    Public() string
}
```

Now head up to the `Execute` method and update the logic there to look for this method.

```
func (t Template) Execute(w http.ResponseWriter, r *http.Request, data interface{}, errs ...error) {
    // ...
    tpl = tpl.Funcs(
        template.FuncMap{
            // ...
            "errors": func() []string {
                var errMessages []string
                for _, err := range errs {
                    var pubErr public
                    if errors.As(err, &pubErr) {
                        errMessages = append(errMessages, pubErr.Public)
                    } else {
                        fmt.Println(err)
                        errMessages = append(errMessages, "Something went wrong"))
                    }
                }
                return errMessages
            },
        },
    )
    // ...
}
```

Now our application will only render errors that have this `Public` method.

If we wanted to log errors that weren't printed out to the end user we can add that to our code.

```
"errors": func() []string {
    var errMessages []string
    for _, err := range errs {
        var pubErr public
        if errors.As(err, &pubErr) {
            errMessages = append(errMessages, pubErr.Public())
        } else {
            // Add this line
            fmt.Println(err)
            errMessages = append(errMessages, "Something went wrong.")
        }
    }
    return errMessages
},
```

The downside is that this is a closure, so every time we call the **errors** function inside our template, the error will be logged. Our current code calls it twice, so we see this logging twice. We can address this by parsing error messages outside the closure, then returning the results in the closure.

Add a new method to parse the error messages.

```
func errMessages(errs ...error) []string {
    var msgs []string
    for _, err := range errs {
        var pubErr public
        if errors.As(err, &pubErr) {
            msgs = append(msgs, pubErr.Public())
        } else {
            fmt.Println(err)
            msgs = append(msgs, "Something went wrong.")
        }
    }
    return msgs
}
```

Then use it in our **Execute** method.

```

func (t Template) Execute(w http.ResponseWriter, r *http.Request, data interface{}, errs ...error) {
    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.StatusInternalServerError)
        return
    }
    // Call the errMessages func before the closures.
    errMsgs := errMessages(errs...)
    tpl = tpl.Funcs(
        template.FuncMap{
            "csrfField": func() template.HTML {
                return csrf.TemplateField(r)
            },
            "currentUser": func() *models.User {
                return context.User(r.Context())
            },
            "errors": func() []string {
                // return the pre-processed err messages inside the closure.
                return errMsgs
            },
        },
    )
    // ...
}

```

Source Code

- [Source Code](#) - see the final source code for this lesson.
- [Diff](#) - see the changes made in this lesson.

20.9 Creating Public Errors

Let's add some code that makes it easier us to wrap any error with a public message.

```
mkdir errors  
code errors/public.go
```

By using the name `errors`, which overlaps with the `errors` package in the standard lib, we may need to use a few tricks to make it easier to use both in one package. We will look at that shortly.

```
package errors

// Public wraps the original error with a new error that has a
// `Public()` string` method that will return a message that is
// acceptable to display to the public. This error can also be
// unwrapped using the traditional `errors` package approach.
func Public(err error, msg string) error {
    return publicError{err, msg}
}

// We need an implementation to work with
type publicError struct {
    err error
    msg string
}

func (pe publicError) Error() string {
    return pe.err.Error()
}

func (pe publicError) Public() string {
    return pe.msg
}

func (pe publicError) Unwrap() error {
    return pe.err
}
```

We can now create a public error with the following Go code:

```
err := doSomething()  
// Wrap with a public message  
err = errors.Public(err, "This is my public message")
```

Source Code

- [Source Code](#) - see the final source code for this lesson.
- [Diff](#) - see the changes made in this lesson.

20.10 Using Public Errors

Next, let's add some logic to use our public errors. We are going to update our users controller to look for the `ErrEmailTaken` error.

code controllers/users.go

```
func (u Users) Create(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.Create(data.Email, data.Password)
    if err != nil {
        if errors.Is(err, models.ErrEmailTaken) {
            err = errors.Public(err, "That email address is already associated with an account.")
        }
        u.Templates.New.Execute(w, r, data, err)
    }
    // ...
}
```

Our code will error from the imports, because we are trying to use the `Is` function and the `Public` function, and each is from a different errors package. We need to do one of two things to fix this:

The first option is to import our app errors package as `apperrors`.

```

import (
    "errors"
    "fmt"
    "net/http"
    "net/url"

    "github.com/joncalhoun/lenslocked/context"
    apperrors "github.com/joncalhoun/lenslocked/errors"
    "github.com/joncalhoun/lenslocked/models"
)

```

Now we can use the following code in our users controller.

```

func (u Users) Create(w http.ResponseWriter, r *http.Request) {
    // ...
    if err != nil {
        if errors.Is(err, models.ErrEmailTaken) {
            err = apperrors.Public(err, "That email address is already associated with an account.")
        }
        u.Templates.New.Execute(w, r, data, err)
    }
    // ...
}

```

The second option is to add a way of accessing the **Is** and **As** functions inside of our own **errors** package.

code errors/wrappers.go

```

package errors

import "errors"

// These variables are used to give us access to existing
// functions in the std lib errors package. We can also
// wrap them in custom functionality as needed if we want,
// or mock them during testing.
var (
    As = errors.As
    Is = errors.Is
)

```

Now if we had back to our users controller, it will work with the original code as long as we import our own errors package, and not the one from the standard library.

```
package controllers

import (
    "fmt"
    "net/http"
    "net/url"

    "github.com/joncalhoun/lenslocked/context"
    "github.com/joncalhoun/lenslocked/errors"
    "github.com/joncalhoun/lenslocked/models"
)

// ...

func (u Users) Create(w http.ResponseWriter, r *http.Request) {
    // ...
    if err != nil {
        if errors.Is(err, models.ErrEmailTaken) {
            err = errors.Public(err, "That email address is already associated with another account")
        }
        u.Templates.New.Execute(w, r, data, err)
        return
    }
    // ...
}
```

By using this approach of wrapping an error with the **Public** function, we can now dictate exactly what error message should be displayed depending on what error we receive in our controllers.

If we restart our app and try to sign up with an existing email address we should now see a much better error message.

Source Code

- [Source Code](#) - see the final source code for this lesson.
- [Diff](#) - see the changes made in this lesson.

20.11 Better Error Handling Exercises

20.11.1 Ex1 - Update our controllers

Update the relevant controllers to redirect the user to the form they just submitted with an error message if the form couldn't be processed.

Start with just using the error provided by the errors package and a generic error message, then see if you can improve errors in any cases.

Ex2 - Explore status codes

Can you use your errors to use better status codes when rendering templates?

We could either include status codes in the error, or we could choose based on the type of error (public = `http.StatusBadRequest`, non-public - `http.StatusInternalServerError`)

Chapter 21

Galleries

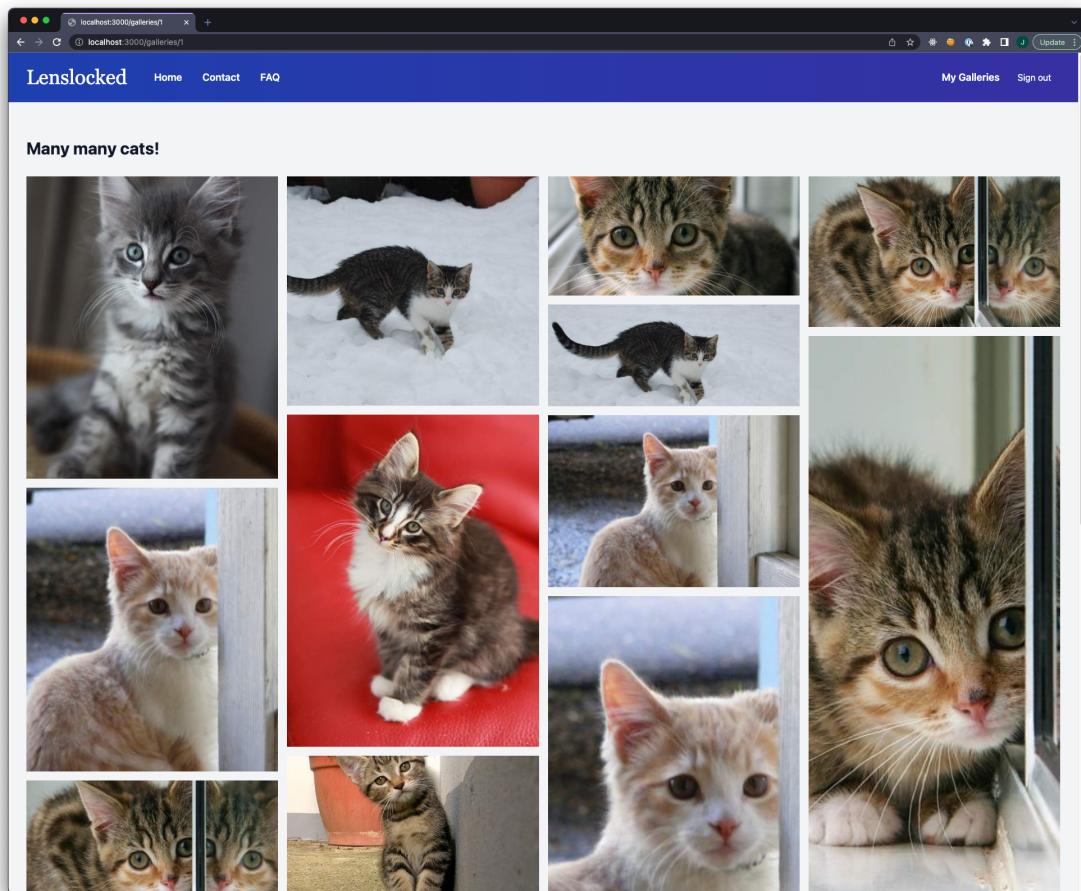
21.1 Galleries Overview

Thus far we have incrementally added code to our views, controllers, and models as we need to add each individual feature. We didn't write the entire `UserService` and `SessionService` all at once; we incrementally added code as we needed it and went back and forth between controllers.

While this approach can work, it can also be useful to step back and think about a feature or functionality being added to an application as a whole. Doing this allows us to think through everything upfront. We might get all the details right, but we will probably make fewer mistakes than if we didn't think about it at all and started coding.

Knowing this, let's take few minutes to talk about what functionality we expect to get out of our galleries feature.

Galleries are meant to give users a way to share photos with others. For instance, I might have a bunch of cat pictures and want to share them in a gallery titled, "Many many cats!"



A gallery will be a page with multiple images on it, and it may be shared with many different people.

Long term we might want to allow users to download images, click on thumbnails, and so on, but for now we are only going to focus on the core functionality which is viewing a set of images.

Knowing what we want our galleries to be, we can start to think about how a user might create a gallery. This will help us determine the functionality that we will need to implement in our application.

For instance, if a user wanted to create a new gallery, the process might be:

1. Enter the title of the gallery.
2. Upload images for the gallery.
3. Get a link to view the gallery.

After thinking about a few different ways a user might interact with our application, we could come up with the following actions we need to support:

1. Create a new gallery with a title.
2. Upload images to a gallery.
3. Delete images from a gallery.
4. Update the title of a gallery.
5. View a gallery (so we can share it with others).
6. Delete a gallery.
7. View a list of galleries we are allowed to edit.

We might also need to restrict some of these actions. For instance, we don't want just anyone adding images to a gallery.

Other actions might be available to everyone. We might want to allow any user, even someone not signed in, to view a gallery.

Knowing this, we can plan out an initial set of functionality that we will need to code in our application.

This might give us the following views we need to create: - Create a new gallery
- Edit a gallery - View an existing gallery - View a list of all of our galleries

We will call these new, edit, show, and index, and these are all very common **CRUD** operations that most resources will require.

We will also need controllers (aka HTTP handlers) to support these views: - New and Create to render and process a new gallery form. - Edit and Update to render and process a form to edit a gallery. - Show to render a gallery. - Delete to delete a gallery.

We might also need some extra handlers to process some additional functionality on pages like the edit gallery page: - An HTTP handler to process image uploads. - An HTTP handler to remove images from a gallery.

And finally, we need a way to persist data in our models package, and this will need to support the following: - Creating a gallery - Updating a gallery - Querying for a gallery by ID - Querying for all galleries with a user ID - Deleting a gallery - Creating an image for a gallery. - Deleting an image from a gallery.

In this section we are going to focus on everything related to the gallery, but we will hold off on things like uploading images for the next section of the course.

21.2 Gallery Model and Migration

Now that we have a rough idea of what will be required to implement our galleries feature, we can start to think about how to structure our data.

In the case of our galleries, the data we need to store in our database is pretty minimal.

Galleries will need: - ID - UserID - Title - Associated Images

A gallery is basically just a container for our images. We need a way to look up all images associated with a gallery, but that can be a separate model that we query via our gallery ID.

We can start by defining a migration.

If the app is running via `modd`, stop it so the migration isn't automatically run

until we are ready for it.

```
cd migrations  
goose create galleries sql
```

For simplicity, we are going to fix the migration number now because we know we are the only one working on the app as we learn.

Normally we would save this until we are getting ready to merge the code in case others are also making migrations.

```
goose fix # converts to 00004_galleries.sql instead of timestamp  
code 00004_galleries.sql
```

In the migration, add the following.

```
-- +goose Up  
-- +goose StatementBegin  
CREATE TABLE galleries (  
    id SERIAL PRIMARY KEY,  
    user_id INT REFERENCES users (id),  
    title TEXT  
);  
-- +goose StatementEnd  
  
-- +goose Down  
-- +goose StatementBegin  
DROP TABLE galleries;  
-- +goose StatementEnd
```

If we start our app up it will run this migration. If we had **modd** running, it may have run the migration before we had it finalized, which can be annoying. When working on migrations, it is a good idea to stop running the app with modd.

Next, let's setup a model that matches our database.

```
cd ..
code models/gallery.go
```

We can then define the model.

```
package models

type Gallery struct {
    ID      int
    UserID int
    Title  string
}
```

Source Code

- [Source Code](#) - see the final source code for this lesson.
- [Diff](#) - see the changes made in this lesson.

21.3 Creating Gallery Records

Now that we have a database table a model, we need a way to insert records into our database.

We will do that by creating a Gallery Service, much like we have in other cases.

```
code models/gallery.go
```

Define the service and the **Create** method.

```
import (
    "database/sql"
    "fmt"
)

type GalleryService struct {
    DB *sql.DB
}

func (service *GalleryService) Create(title string, userID int) (*Gallery, error) {
    gallery := Gallery{
        Title: title,
        UserID: userID,
    }
    row := service.DB.QueryRow(`  

        INSERT INTO galleries (title, user_id)  

        VALUES ($1, $2) RETURNING id;`, gallery.Title, gallery.UserID)
    err := row.Scan(&gallery.ID)
    if err != nil {
        return nil, fmt.Errorf("create gallery: %w", err)
    }
    return &gallery, nil
}
```

Source Code

- [Source Code](#) - see the final source code for this lesson.
- [Diff](#) - see the changes made in this lesson.

21.4 Querying for Galleries by ID

We are going to need to query for a single gallery by its ID so we can both show the gallery and render its edit page.

```
code models/gallery.go
```

```
func (service *GalleryService) ByID(id int) (*Gallery, error) {
    gallery := Gallery{
        ID: id,
    }
    row := service.DB.QueryRow(`SELECT title, user_id
                                FROM galleries
                                WHERE id = $1;`, gallery.ID)
    err := row.Scan(&gallery.Title, &gallery.UserID)
    if err != nil {
        return nil, fmt.Errorf("query gallery by id: %w", err)
    }
    return &gallery, nil
}
```

We can improve our error handling by checking to see if the error is a `sql.ErrNoRows` error, and returning our own error from the `models` package.

This helps because users of the models package don't need to know about sql being used behind the scenes.

Let's create a file to store errors we will be using and exporting. While we are at it, we can move the `ErrEmailTaken` error to this file.

code `models/errors.go`

```
package models

import "errors"

var (
    ErrNotFound = errors.New("models: resource could not be found")
    ErrEmailTaken = errors.New("models: email address is already in use")
)
```

Remove the original `ErrEmailTaken` variable.

```
code models/user.go
```

```
// Remove this code from the user.go source file
var (
    ErrEmailTaken = errors.New("models: email address is already in use")
)
```

Now update our gallery source file.

```
code models/gallery.go
```

```
func (service *GalleryService) ByID(id int) (*Gallery, error) {
// ...
if err != nil {
    if errors.Is(err, sql.ErrNoRows) {
        return nil, ErrNotFound
    }
    return nil, fmt.Errorf("query gallery by id: %w", err)
}
return &gallery, nil
}
```

Source Code

- [Source Code](#) - see the final source code for this lesson.
- [Diff](#) - see the changes made in this lesson.

21.5 Querying Galleries by UserID

We are also going to need to query for all of the galleries that a single user owns so we can show them a list of all their galleries.

We will do this by querying by UserID.

```
code models/gallery.go
```

```
func (service *GalleryService) ByUserID(userID int) ([]Gallery, error) {
    rows, err := service.DB.Query(`SELECT id, title
                                    FROM galleries
                                    WHERE user_id = $1;`, userID)
    if err != nil {
        return nil, fmt.Errorf("query galleries by user: %w", err)
    }
    var galleries []Gallery
    for rows.Next() {
        gallery := Gallery{
            UserID: userID,
        }
        err := rows.Scan(&gallery.ID, &gallery.Title)
        if err != nil {
            return nil, fmt.Errorf("query galleries by user: %w", err)
        }
        galleries = append(galleries, gallery)
    }
    if rows.Err() != nil {
        return nil, fmt.Errorf("query galleries by user: %w", err)
    }
    return galleries, nil
}
```

Source Code

- [Source Code](#) - see the final source code for this lesson.
- [Diff](#) - see the changes made in this lesson.

21.6 Updating Gallery Records

We will also need a way to update a gallery.

This will be pretty basic, as image uploading and deleting will be handled separately, so for now this only involves updating the title.

We could eventually expand our features to include things like rearranging photos, publishing a gallery, marking it as private, and more, but for now we won't have any of those features.

```
code models/gallery.go
```

```
func (service *GalleryService) Update(gallery *Gallery) error {
    _, err := service.DB.Exec(`UPDATE galleries
        SET title = $2
        WHERE id = $1;`, gallery.ID, gallery.Title)
    if err != nil {
        return fmt.Errorf("update gallery: %w", err)
    }
    return nil
}
```

Source Code

- [Source Code](#) - see the final source code for this lesson.
- [Diff](#) - see the changes made in this lesson.

21.7 Deleting Gallery Records

Finally, we need a way to delete a gallery.

```
code models/gallery.go
```

```
func (service *GalleryService) Delete(id int) error {
    _, err := service.DB.Exec(`DELETE FROM galleries
        WHERE id = $1;`, id)
```

```
    if err != nil {
        return fmt.Errorf("delete gallery by id: %w", err)
    }
    return nil
}
```

Source Code

- [Source Code](#) - see the final source code for this lesson.
- [Diff](#) - see the changes made in this lesson.

21.8 New Gallery Handler

With our models all prepared, we can start working on our controllers and views.

We will start with a handler for rendering the new gallery page, then add a template to render the view, and after that we can add a handler to our controller to process the form submission.

```
code controllers/galleries.go
```

```
package controllers

import (
    "fmt"
    "net/http"

    "github.com/joncalhoun/lenslocked/context"
    "github.com/joncalhoun/lenslocked/models"
)

type Galleries struct {
    Templates struct {
```

```
        New      Template
    }
    GalleryService *models.GalleryService
}

func (g Galleries) New(w http.ResponseWriter, r *http.Request) {
    var data struct {
        Title string
    }
    data.Title = r.FormValue("title")
    g.Templates.New.Execute(w, r, data)
}
```

Source Code

- [Source Code](#) - see the final source code for this lesson.
- [Diff](#) - see the changes made in this lesson.

21.9 views.Template Name Bug

Aside 21.1. Quick summary

In this lesson we deep-dive into a bug that can be a bit confusing, and requires a somewhat detailed understanding of how Go templates work.

If everything in the lesson doesn't make sense, don't let that stop you from proceeding. Make the code updates and move forward with the course, then come back to this later after you have used templates more.

We are about to add a new template for our new gallery page.

A common way to do this is to nest the `.gohtml` files in a directory with the name of the resource.

For instance, in our `templates` directory, we might create a new folder with the name `galleries` and then add a file named `new.gohtml` to that directory.

```
mkdir templates/galleries
code templates/galleries/new.gohtml
```

We could then add some contents to our template file.

```
<h1>Create a new Gallery</h1>
```

While there is nothing wrong with this approach, it will lead to a bug in our current code. If we try to use a nested file like this, our template execution will result in an error.

```
executing template: template: "galleries/new.gohtml" is an incomplete or empty template
```

Let's test it.

```
code main.go
```

```
func main() {
    // Add this where the other services are created
    galleryService := &models.GalleryService{
        DB: db,
    }

    // Add this where the other controllers are created
    galleriesC := controllers.Galleries{
        GalleryService: galleryService,
    }
    galleriesC.Templates.New = views.Must(views.ParseFS(
```

```

    templates.FS,
    "galleries/new.gohtml", "tailwind.gohtml",
))

// Finally, add a route so we can try to use the new template.
// Do this near the end of the other routes.
r.Get("/galleries/new", galleriesC.New)
}

```

Let's figure out what causes this bug, and how to fix it. We will start at our `views` package's `ParseFS` function.

`code views/template.go`

In this code we name our template using the provided pattern.

```

func ParseFS(fs fs.FS, patterns ...string) (Template, error) {
    tpl := template.New(patterns[0])
    // ...
}

```

In the case of the `galleries/new` template, the name will be `galleries/new.gohtml`.

The `html/template` package uses the name `new.gohtml` when it parses this file. The `html/template` package does not include the path in a template name.

When we go to execute our template, we execute the `galleries/new.gohtml` template, but it has nothing in it because its contents were placed in a template named `new.gohtml` by the `html/template` package.

If we wanted, we could see this all in the `html/template` source code.

It is also in the docs. `ParseFS` states that it behaves mostly like `ParseFiles`, except it reads from an `fs.FS`.

The [ParseFiles docs](#) state the following (*emphasis added by me*):

ParseFiles creates a new Template and parses the template definitions from the named files. **The returned template's name will have the (base) name** and (parsed) contents of the first file. There must be at least one file. If an error occurs, parsing stops and the returned *Template is nil.

When parsing multiple files with the same name in different directories, the last one mentioned will be the one that results. For instance, ParseFiles("a/foo", "b/foo") stores "b/foo" as the template named "foo", while "a/foo" is unavailable.

21.9.1 How do we fix it?

The easiest way to fix this issue is to use the base filename for our template name, much like the standard library does.

code views/template.go

```
func ParseFS(fs fs.FS, patterns ...string) (Template, error) {
    tpl := template.New(filepath.Base(patterns[0]))
    // ...
}
```

With that update made, we should now be able to visit the [/galleries/new](#) page and see the rendered template.

Source Code

- [Source Code](#) - see the final source code for this lesson.
- [Diff](#) - see the changes made in this lesson.

21.10 New Gallery Template

As we saw in the last lesson, we are going to store our templates in a nested directory.

```
templates/
  galleries/
    new.gohtml
    --> We will add new templates here for galleries

  signup.gohtml
  ...
```

We will start with the header and footer tempaltes, as well as a div to contain our heading.

```
{{template "header" .}}
<div class="p-8 w-full">
  <h1 class="pt-4 pb-8 text-3xl font-bold text-gray-800">
    Create a new Gallery
  </h1>
</div>
{{template "footer" .}}
```

Once we have verified this is working we can add the form to our page. This will go inside the div we created and after the **h1** inside of it.

```
<form action="/galleries" method="post">
  <div class="hidden">
    {{csrfField}}
  </div>
  <div class="py-2">
    <label for="title" class="text-sm font-semibold text-gray-800">
      Title
    </label>
    <input
      name="title"
      id="title"
      type="text"
```

```
placeholder="Gallery Title"
required
class=""
  w-full
  px-3
  py-2
  border border-gray-300
  placeholder-gray-500
  text-gray-800
  rounded
"
  value="{{.Title}}"
  autofocus
/>
</div>

<div class="py-4">
  <button
    type="submit"
    class=""
    py-2
    px-8
    bg-indigo-600
    hover:bg-indigo-700
    text-white
    rounded
    font-bold
    text-lg
  "
  >
    Create
  </button>
</div>
</form>
```

Aside 21.2. Styling HTML

Typically when writing my HTML and classes I won't break it into many lines like we see here with the **button**. Instead, I will let the editor word-wrap as necessary.

I break the classes into new lines in the book because it helps when presenting the code in the book format where my tooling doesn't always word-wrap correctly.

Feel free to write it in whatever format you prefer.

Source Code

- [Source Code](#) - see the final source code for this lesson.
- [Diff](#) - see the changes made in this lesson.

21.11 Gallery Routing and CSRF Bug Fixes

We need to address two “bugs” in our code.

The first is something we did temporarily, so it isn’t really a bug.

The second is a bug related to our CSRF protection.

If we head to our [/galleries/new](#) page, we will see a form to create a new gallery. Unfortunately, anyone who visits this URL on our website can see this form, even if they aren’t signed in! We want to restrict access to this page to signed in users, so we need to update our routing to use middleware to do this.

```
code main.go
```

Head to the code where we define our routes. We are going to remove the existing route and add the following.

```
func main() {
    // ...

    // Replace the existing /galleries/new route with the following.
    r.Route("/galleries", func(r chi.Router) {
        r.Group(func(r chi.Router) {
            r.Use(umw.RequireUser)
            r.Get("/new", galleriesC.New)
        })
    })
}

// ...
```

`r.Route` will cause any routes defined in the closure to be subroutes. Eg `/new` ends up being `/galleries/new`.

`r.Group` is similar, but it doesn't change the URL by adding a prefix. This is useful when we want to apply the same middleware to a group of routes, but not to all routes. In our case, pages like the new gallery form will require a user to be signed in, but pages like viewing a gallery will be accessible to anyone. We don't have a route to view a gallery, but when we add one later this will likely make a bit more sense.

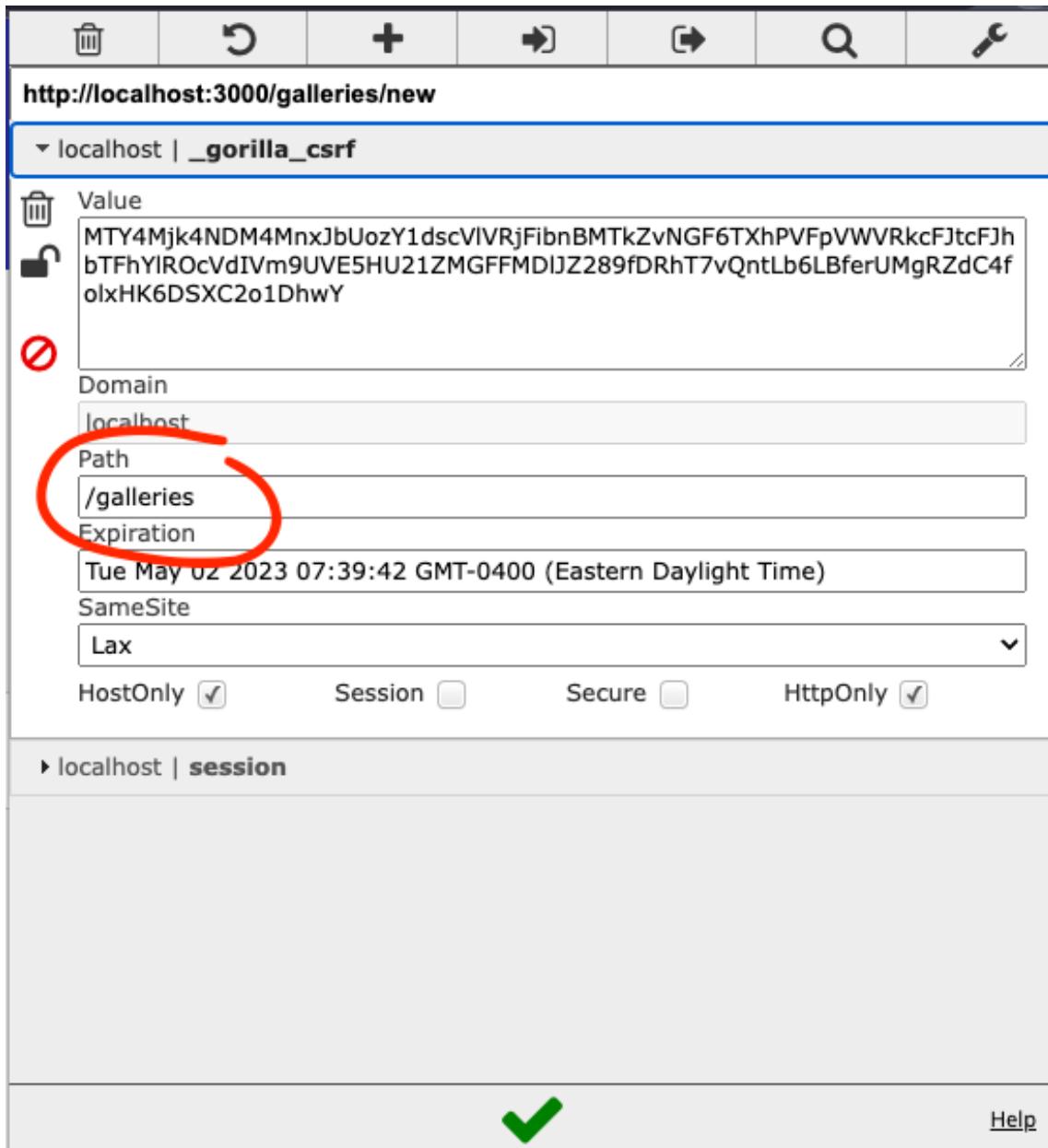
21.11.1 CSRF Bug

The CSRF bug we have is that our CSRF middleware is storing the CSRF cookie with an invalid path attribute.

We can see this bug by navigating to the `/galleries/new` page and trying to use the “Sign out” link on our navbar.

By default, the CSRF library we use sets the path attribute based on the current URL.

On pages like `/signup` and `/signin` the path used was `/`, so our CSRF tokens always worked. In the case of our new gallery page, our path is `/galleries/new` and the CSRF middleware uses the path `/galleries` in the CSRF cookie.



If we try to submit a form to a URL with the `/galleries` path prefix, the CSRF token will work fine.

BUT, if we try to submit a form to `/signout`, like we do with the sign out link, this doesn't have the `/galleries` prefix so the CSRF cookie isn't included in

the request.

To fix this, we need to tell our CSRF middleware to always use the path `/`, which will make the CSRF cookie work on every page regardless of what the current path is.

```
code main.go
```

Find the code where we declare the CSRF middleware in the `main()` function and update it to add the `csrf.Path` option.

```
csrfMw := csrf.Protect(
    []byte(cfg.CSRF.Key),
    csrf.Secure(cfg.CSRF.Secure),
    csrf.Path("/"),
)
```

Due to the way the CSRF package works, we may also need to clear our cookies before this change will fully work.

You can do this in almost all browsers by clearing your browsing data. You can also use an extension like [EditThisCookie](#) to delete the cookie on the [/galleries/new](#) page.

Source Code

- [Source Code](#) - see the final source code for this lesson.
- [Diff](#) - see the changes made in this lesson.

21.12 Create Gallery Handler

Our next task is to process the new gallery form and use the data to create a gallery in our database.

```
code controllers/galleries.go
```

We will add the new code in a method named **Create**.

```
func (g Galleries) Create(w http.ResponseWriter, r *http.Request) {
    var data struct {
        UserID int
        Title  string
    }
    data.UserID = context.User(r.Context()).ID
    data.Title = r.FormValue("title")

    gallery, err := g.GalleryService.Create(data.Title, data.UserID)
    if err != nil {
        g.Templates.New.Execute(w, r, data, err)
        return
    }
    // This page doesn't exist, but we will want to redirect here eventually.
    editPath := fmt.Sprintf("/galleries/%d/edit", gallery.ID)
    http.Redirect(w, r, editPath, http.StatusFound)
}
```

Next we need to update our routes with this new page.

```
code main.go
```

Our create gallery page accessed via a POST to the **/galleries** path, and a user is required to access this route.

```
func main() {
    /**
     r.Route("/galleries", func(r chi.Router) {
        r.Group(func(r chi.Router) {
            r.Use(umw.RequireUser)
            r.Get("/new", galleriesC.New)
            r.Post("/", galleriesC.Create)
        })
    })
    // ...
}
```

Optional - use `docker compose exec` with `psql` to verify galleries are being created correctly.

Source Code

- [Source Code](#) - see the final source code for this lesson.
- [Diff](#) - see the changes made in this lesson.

21.13 Edit Gallery Handler

We need to create a handler for editing a gallery.

```
code controllers/galleries.go
```

The first thing we need to do in our handler function is get the ID of the gallery we are editing.

```
func (g Galleries) Edit(w http.ResponseWriter, r *http.Request) {
    id, err := strconv.Atoi(chi.URLParam(r, "id"))
    if err != nil {
        // 404 error - page isn't found.
        http.Error(w, "Invalid ID", http.StatusNotFound)
        return
    }
}
```

Aside 21.3. Chi's URLParam

Chi allows us to add parameters to the path using the `{` and `}` syntax. For instance, the following code would setup a path where the `id` is a dynamic parameter.

```
r.Get("/dogs/{id}/edit", ...)
```

To access the `id` part of the URL, we need to call `chi.URLParam` and pass in the name given when we setup our route.

A good general rule to follow when determining if something should be a URL parameter is to think about what data is required versus what information is optional for a path to work. Anything required should be part of the path as a URL parameter, and anything extra should be a URL query parameter (the ones after the `?` in a url).

With our gallery we need the `id` of a gallery to render it, and optionally we might allow the `title` to be provided as part of the URL. As a result, we would place the `id` in the URL, and allow the title to be provided via query parameters like in the example below.

```
/galleries/123/edit?title=Fancy%20New%20Title
```

Once we have a gallery ID we can query for it using our gallery service.

```
gallery, err := g.GalleryService.ByID(id)
if err != nil {
    if err == models.ErrNotFound {
        http.Error(w, "Gallery not found", http.StatusNotFound)
        return
    }
    http.Error(w, "Something went wrong", http.StatusInternalServerError)
    return
}
```

Before we can render the gallery information, we need to verify that the user actually owns this gallery.

```
user := context.User(r.Context())
if gallery.UserID != user.ID {
    http.Error(w, "You are not authorized to edit this gallery", http.StatusForbidden)
    return
}
```

Finally, we can construct the data we need to render the edit page and add the **Edit** template to our **Galleries** type.

```
type Galleries struct {
    Templates struct {
        New   Template
        Edit  Template
    }
    GalleryService *models.GalleryService
}

// ...

func (g Galleries) Edit(w http.ResponseWriter, r *http.Request) {
    // ...

    data := struct {
        ID   int
        Title string
    }{
        ID:   gallery.ID,
        Title: gallery.Title,
    }
    g.Templates.Edit.Execute(w, r, data)
}
```

This page won't render until we create the template in the next lesson.

Source Code

- [Source Code](#) - see the final source code for this lesson.
- [Diff](#) - see the changes made in this lesson.

21.14 Edit Gallery Template

Create the template file

```
code templates/galleries/edit.gohtml
```

Add the basics to verify it is rendering.

```
{{template "header" .}}
<div class="p-8 w-full">
  <h1 class="pt-4 pb-8 text-3xl font-bold text-gray-800">
    Edit your Gallery
  </h1>
</div>
{{template "footer" .}}
```

Make sure this page works by assigning it to the galleries controller **Edit** template.

```
code main.go
```

Find the section where we assign templates to our galleries controller and add new template.

```
func main() {
  // ...
  galleriesC.Templates.Edit = views.Must(views.ParseFS(
    templates.FS,
    "galleries/edit.gohtml", "tailwind.gohtml",
  ))
  // ...
}
```

We also need to add the route. This will use a URL parameter, and the syntax for that in the chi router is to use the **{** and **}** brackets.

```
func main() {
    // ...
    r.Route("/galleries", func(r chi.Router) {
        r.Group(func(r chi.Router) {
            r.Use(umw.RequireUser)
                r.Get("/new", galleriesC.New)
                r.Post("/", galleriesC.Create)
                r.Get("/{id}/edit", galleriesC.Edit)
            })
        })
    // ...
}
```

Restart the app and verify the edit page renders for galleries you have created. It should error for galleries that don't exist, or galleries your current account didn't create. Test these cases as well.

Once all is working we can add the form to the page. This is roughly the same as the new gallery page, except we updated the action path for the form and the button text.

code `templates/galleries/edit.gohtml`

```
<form action="/galleries/{{.ID}}" method="post">
    <div class="hidden">
        {{csrfField}}
    </div>
    <div class="py-2">
        <label for="title" class="text-sm font-semibold text-gray-800">
            Title
        </label>
        <input
            name="title"
            id="title"
            type="text"
            placeholder="Gallery Title"
            required
            class="
                w-full
                px-3
                py-2
                border border-gray-300
            " />
    </div>
</form>
```

```
placeholder-gray-500
text-gray-800
rounded
"
value="{{.Title}}"
autofocus
/>
</div>
<div class="py-4">
  <button
    type="submit"
    class=""
    py-2
    px-8
    bg-indigo-600
    hover:bg-indigo-700
    text-white
    rounded
    font-bold
    text-lg
  "
  >
    Update
  </button>
</div>
</form>
```

We don't have a handler for processing this form yet, so submitting it will not work.

We can also add a button for deleting the gallery while we are here. Add the following below the edit gallery form inside the `edit.gohtml` template file.

```
<div class="py-4">
  <h2>Dangerous actions</h2>
  <form action="/galleries/{{.ID}}/delete" method="post" onsubmit="return confirm('Do you really want to delete this gallery?')"
    <div class="hidden">
      {{csrfField}}
    </div>
    <button
      type="submit"
      class=""
      py-2
      px-8
      bg-red-600
      hover:bg-red-700
    >
      Delete
    </button>
  </form>
</div>
```

```
    text-white
    rounded
    font-bold
    text-lg
  "
>
  Delete
</button>
</form>
</div>
```

Again, this form won't work until we add a handler for deleting galleries.

Aside 21.4. HTML onsubmit

The `onsubmit` attribute in HTML allows us to run some JavaScript prior to submitting an HTML form. In this case, we are calling the `confirm` function, which will produce a popup asking the user to confirm that they want to delete the gallery. We then return the result from this confirm function, which will determine whether our form is actually submitted or not. This is a handy way of quickly adding confirmations to dangerous tasks like deleting a resource.

Source Code

- [Source Code](#) - see the final source code for this lesson.
- [Diff](#) - see the changes made in this lesson.

21.15 Update Gallery Handler

Our next handler is the one used to process the edit gallery form. Once again we will be working in the galleries controller source file.

```
code controllers/galleries.go
```

Our code starts off very similar to the edit page: 1. Get the ID 2. Query for the gallery 3. Verify the user has access to it.

```
func (g Galleries) Update(w http.ResponseWriter, r *http.Request) {
    id, err := strconv.Atoi(chi.URLParam(r, "id"))
    if err != nil {
        http.Error(w, "Invalid ID", http.StatusNotFound)
        return
    }
    gallery, err := g.GalleryService.ByID(id)
    if err != nil {
        if errors.Is(err, models.ErrNotFound) {
            http.Error(w, "Gallery not found", http.StatusNotFound)
            return
        }
        http.Error(w, "Something went wrong", http.StatusInternalServerError)
        return
    }
    user := context.User(r.Context())
    if gallery.UserID != user.ID {
        http.Error(w, "You are not authorized to edit this gallery", http.StatusForbidden)
        return
    }
}
```

Once we have all of that out of the way we can parse the title from the form and update our gallery.

```
func (g Galleries) Update(w http.ResponseWriter, r *http.Request) {
    // ...

    title := r.FormValue("title")
    gallery.Title = title
    err = g.GalleryService.Update(gallery)
    if err != nil {
        http.Error(w, "Something went wrong", http.StatusInternalServerError)
        return
    }
    editPath := fmt.Sprintf("/galleries/%d/edit", gallery.ID)
    http.Redirect(w, r, editPath, http.StatusFound)
}
```

After we successfully update a gallery we can either re-render the edit page, or redirect to the index page.

We will redirect back to the edit page for now, as this will make more sense when we start to add images and want to see the gallery with them added.

Lastly, we want to update our routes with this new handler.

```
code main.go
```

Find the routes and update the galleries routes to include the update path.

```
r.Route("/galleries", func(r chi.Router) {
    r.Group(func(r chi.Router) {
        r.Use(umw.RequireUser)
        r.Get("/new", galleriesC.New)
        r.Post("/", galleriesC.Create)
        r.Get("/{id}/edit", galleriesC.Edit)
        // Add this line
        r.Post("/{id}", galleriesC.Update)
    })
})
```

After restarting the application we can verify that everything works by attempting to update the title of an existing gallery.

Source Code

- [Source Code](#) - see the final source code for this lesson.
- [Diff](#) - see the changes made in this lesson.

21.16 Gallery Index Handler

```
code controllers/galleries.go
```

We are going ot need a template to render, so we can start there.

```
type Galleries struct {
    Templates struct {
        New   Template
        Edit  Template
        Index Template
    }
    GalleryService *models.GalleryService
}
```

Next we can define the handler function and any data we are going to use to render the template. It is often a good idea to create a new data type for our views that is different from the models as we will often want different data in our views than what we store directly in our database.

```
func (g Galleries) Index(w http.ResponseWriter, r *http.Request) {
    type Gallery struct {
        ID      int
        Title  string
    }
    var data struct {
        Galleries []Gallery
    }
    // TODO: Lookup the galleries we are going to render
    g.Templates.Index.Execute(w, r, data)
}
```

Once we have the outline of our function in place we can think about how we need to get the data we want. In this case, we want to query for all of the galleries a specific user owns and then convert them to the **Gallery** type we just defined.

```

user := context.User(r.Context())
galleries, err := g.GalleryService.ByUserID(user.ID)
if err != nil {
    http.Error(w, "Something went wrong", http.StatusInternalServerError)
    return
}
for _, gallery := range galleries {
    data.Galleries = append(data.Galleries, Gallery{
        ID: gallery.ID,
        Title: gallery.Title,
    })
}

```

We can then setup the route for this handler.

code main.go

Find the existing galleries routes and add the index page.

```

r.Route("/galleries", func(r chi.Router) {
    r.Group(func(r chi.Router) {
        r.Use(umw.RequireUser)
        // Add this line
        r.Get("/", galleriesC.Index)
        r.Get("/new", galleriesC.New)
        r.Post("/", galleriesC.Create)
        r.Get("/{id}/edit", galleriesC.Edit)
        r.Post("/{id}", galleriesC.Update)
    })
})

```

It is a bit annoying typing all these paths in manually. Let's add a link to the navbar.

code templates/tailwind.gohtml

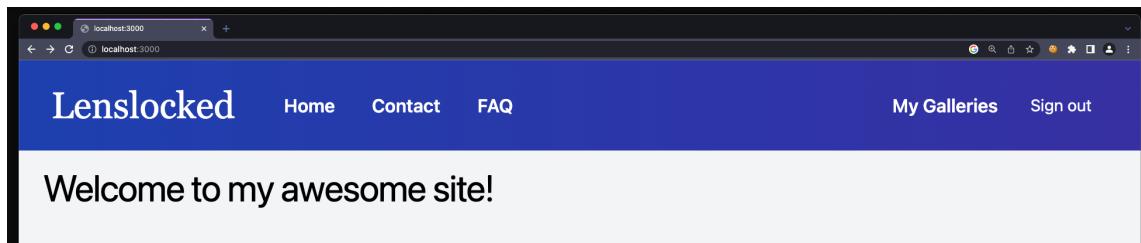
First, remove the **flex-grow** from the **div** that contains our Home, Contact, and FAQ links.

```
<!-- Remove the flex-grow class from the following div -->
<div class="">
  <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>
```

Just after the div with our Home, Contact, and FAQ links, add the following HTML.

```
{{if currentUser}}
  <div class="flex-grow flex flex-row-reverse">
    <a class="text-lg font-semibold hover:text-blue-100 pr-8" href="/galleries">My Galleries</a>
  </div>
{{else}}
  <div class="flex-grow"></div>
{{end}}
```

This will add a div that grows and places its contents on the right side if we are signed in. The result is a navbar with a link to our galleries near the signout button.



If we are not signed in, we will see a navbar that looks identical to what we saw before, as the div with **flex-grow** is an empty div.

Source Code

- [Source Code](#) - see the final source code for this lesson.
- [Diff](#) - see the changes made in this lesson.

21.17 Discovering and Fixing a Gallery Index Bug

We need a template to render all of our galleries.

```
code templates/galleries/index.gohtml
```

As always, start with a title and then we can slowly add more content.

```
{{template "header" .}}
<div class="p-8 w-full">
  <h1 class="pt-4 pb-8 text-3xl font-bold text-gray-800">
    My Galleries
  </h1>
  <!-- TODO: Add code to render galleries -->
</div>
{{template "footer" .}}
```

Next we want to connect this to our handler so we can test any UI changes we make.

```
code main.go
```

Find the section of the `main()` function where we parse other templates and add the following.

```
galleriesC.Templates.Index = views.Must(views.ParseFS(
  templates.FS,
  "galleries/index.gohtml", "tailwind.gohtml",
))
```

Restart the application and try to view the index page. We will see an error! Ugh.

Let's figure out what is going on. Head to the gallery controller.

```
code controllers/galleries.go
```

Add a print statement to the error case inside the `Index` method so we can determine what is causing the error. In many cases like this where we are using `http.Error`, we likely want to either print out the error, or use some other way of logging what went wrong in case something goes wrong in production and isn't easy to replicate.

```
func (g Galleries) Index(w http.ResponseWriter, r *http.Request) {
    // ...
    if err != nil {
        fmt.Println(err)
        http.Error(w, "Something went wrong", http.StatusInternalServerError)
        return
    }
    // ...
}
```

Now try to visit the `/galleries` page again. Upon looking our terminal logs, we will see the following error.

```
query galleries by user: sql: Scan error on column index 1, name "title": destination not a poi
```

This is saying that when we query our SQL table and then try to call `Scan`, the value we passed in to assign the value to is not a pointer.

That suggests our bug is in the gallery model.

```
code models/gallery.go
```

Looking at the `ByUserID` method, we can see that we forgot the `&` sign when calling `rows.Scan`. Add it and we should be good to go.

```
err = rows.Scan(&gallery.ID, &gallery.Title)
```

Restart the application and verify that everything is working and we can see the start of our index page.

Source Code

- [Source Code](#) - see the final source code for this lesson.
- [Diff](#) - see the changes made in this lesson.

21.18 Gallery Index Template Continued

Now that our bug is resolved, we are ready to finish up the index template.

```
code templates/galleries/index.gohtml
```

We want to render all of our galleries, so an HTML table is a good fit. We can start with some headings based on what we want to render. In our case, we will want to show the ID of the gallery, the title, and actions we can perform.

```
<table class="w-full table-fixed">
  <thead>
    <tr>
      <th class="p-2 text-left w-24">ID</th>
      <th class="p-2 text-left">Title</th>
      <th class="p-2 text-left w-96">Actions</th>
    </tr>
  </thead>
  <tbody>
    <!-- TODO: Add code to render each gallery in the table -->
  </tbody>
</table>
```

Aside 21.5. Why are we showing the gallery ID?

In most applications, showing the ID of each gallery won't help users very much, so adding it to our table isn't necessary. We could have a much cleaner table with just the title of each gallery and a list of actions.

When learning to develop web applications, I find that it helps to have the ID present so that we can more easily debug and perform manual SQL queries if necessary, so I am opting to include it in our table.

Next we can use a **range** to fill in the rows of our table.

```
<!-- Add this inside the tbody -->
{{range .Galleries}}
  <tr class="border">
    <td class="p-2 border">{{.ID}}</td>
    <td class="p-2 border">{{.Title}}</td>
    <td class="p-2 border">
      <a href="/galleries/{{.ID}}">View</a>
      <a href="/galleries/{{.ID}}/edit">Edit</a>
    </td>
  </tr>
{{end}}
```

We also need a way to create a new gallery. We will offer this via a button after our table.

```
<!-- Place this after the </table> closing -->


<a href="/galleries/new"
    class=""
    py-2 px-8
    bg-indigo-600 hover:bg-indigo-700
    text-lg text-white font-bold
    rounded">


```

```
New Gallery
</a>
</div>
```

This all gives us a working page, but we can improve it a bit.

For starters, we can improve the style of the actions by making them small buttons with varying colors. While doing this, we can make add the `flex` and `space-x-2` classes to their enclosing container to ensure they are spaced out and on the same row.

```
<td class="p-2 border flex space-x-2">
  <a class=""
    py-1 px-2
    bg-blue-100 hover:bg-blue-200
    rounded border border-blue-600
    text-xs text-blue-600"
    href="/galleries/{{.ID}}"
  >
  View
</a>
  <a class=""
    py-1 px-2
    bg-yellow-100 hover:bg-yellow-200
    rounded border border-yellow-600
    text-xs text-yellow-600"
    href="/galleries/{{.ID}}/edit"
  >
  Edit
</a>
</td>
```

Another thing we can do is add a button to delete each gallery right from the index page. We can add this next to the other action buttons in the table. It will be a bit more code because we need a form to POST to an endpoint.

```
<form action="/galleries/{{.ID}}/delete" method="post"
  onsubmit="return confirm('Do you really want to delete this gallery?');">
  <div class="hidden">{{csrfField}}</div>
  <button type="submit"
    class=""

  </button>
</form>
```

```
py-1 px-2  
bg-red-100 hover:bg-red-200  
rounded border border-red-600  
text-xs text-red-600"  
>  
Delete  
</button>  
</form>
```

Before moving on, we can test that everything we added works. - The new button sends us to the new gallery page - The edit button for each gallery goes to that gallery's edit page - The view button for each gallery should send us to a path like `/galleries/{id}`, though this won't work yet. - The delete button for each gallery should confirm whether we want to delete or not, then attempt to POST to `/galleries/{id}/delete`.

Source Code

- [Source Code](#) - see the final source code for this lesson.
- [Diff](#) - see the changes made in this lesson.

21.19 Show Gallery Handler

We aren't going to restrict access to this page like we have with other gallery pages. Anyone with a link to a gallery will be able to view it.

We will start in the galleries controller source file.

```
code controllers/galleries.go
```

In this source file we will add the `Show` method, and then in it we are going to:

1. Get the ID from the URL param
2. Lookup the gallery

This sounds eerily familiar. We have done this exact same thing in the **Edit** and **Update** methods. For now we can copy that source code, but later we can look at options for removing some of this code duplication if we want to.

```
func (g Galleries) Show(w http.ResponseWriter, r *http.Request) {
    id, err := strconv.Atoi(chi.URLParam(r, "id"))
    if err != nil {
        http.Error(w, "Invalid ID", http.StatusNotFound)
        return
    }
    gallery, err := g.GalleryService.ByID(id)
    if err != nil {
        if errors.Is(err, models.ErrNotFound) {
            http.Error(w, "Gallery not found", http.StatusNotFound)
            return
        }
        http.Error(w, "Something went wrong", http.StatusInternalServerError)
        return
    }
    // TODO: Render the gallery
}
```

Be sure to not copy over the logic that requires a user to own the gallery. We aren't going to add that restriction to our show page, and instead we are going to allow anyone, even users who are not signed in, to view a gallery.

We need a template to render our gallery, so we can add that next.

```
type Galleries struct {
    Templates struct {
        Show   Template
        New    Template
        Edit   Template
        Index  Template
    }
    GalleryService *models.GalleryService
}
```

Heading back to the **Show** method, we can use this to render our template.

```
func (g Galleries) Show(w http.ResponseWriter, r *http.Request) {
    // ...
    g.Templates.Show.Execute(w, r, gallery)
}
```

Like in previous handler functions, we likely want to define a new type rather than passing our model directly into our template.

We could reuse the **Gallery** type from our **Index**, but there is a good chance that we will need different data when viewing a single gallery than what we need when viewing a table listing all of our galleries. One example of this is a gallery's images. On the index page we won't be rendering these, but when showing a gallery we will. Update the **Show** handler to use the following code to setup the data struct and then execute the correct template with the data.

```
var data struct {
    ID      int
    Title   string
    Images  []string
}
data.ID = gallery.ID
data.Title = gallery.Title
g.Templates.Show.Execute(w, r, data)
```

We can also add some code to generate some random images until we properly support uploading images to a gallery. This will help us when designing the show gallery template.

```
// We are going to psuedo-randomly come up with 20 images to render for our
// gallery until we actually support uploading images. These images will use
// placekitten.com, which gives us cat images.
for i := 0; i < 20; i++ {
    // width and height are random values between 200 and 700
    w, h := rand.Intn(500)+200, rand.Intn(500)+200
    // using the width and height, we generate a URL
    catImageURL := fmt.Sprintf("https://placekitten.com/%d/%d", w, h)
```

```
// Then we add the URL to our images.
data.Images = append(data.Images, catImageURL)
}
```

Putting this all together, we get the following handler function.

```
func (g Galleries) Show(w http.ResponseWriter, r *http.Request) {
    id, err := strconv.Atoi(chi.URLParam(r, "id"))
    if err != nil {
        http.Error(w, "Invalid ID", http.StatusNotFound)
        return
    }
    gallery, err := g.GalleryService.ByID(id)
    if err != nil {
        if errors.Is(err, models.ErrNotFound) {
            http.Error(w, "Gallery not found", http.StatusNotFound)
            return
        }
        http.Error(w, "Something went wrong", http.StatusInternalServerError)
        return
    }
    var data struct {
        ID      int
        Title   string
        Images  []string
    }
    data.ID = gallery.ID
    data.Title = gallery.Title
    for i := 0; i < 20; i++ {
        w, h := rand.Intn(500)+200, rand.Intn(500)+200
        catImageURL := fmt.Sprintf("https://placekitten.com/%d/%d", w, h)
        data.Images = append(data.Images, catImageURL)
    }
    g.Templates.Show.Execute(w, r, data)
}
```

Our last step is to update `main.go` with the new handler and its route.

```
code main.go
```

This route will be outside the `Group` because we don't want it to require a user.

```
r.Route("/galleries", func(r chi.Router) {
    r.Get("/{id}", galleriesC.Show) // <-- Add this
    r.Group(func(r chi.Router) {
        r.Use(umw.RequireUser)
        r.Get("/", galleriesC.Index)
        r.Get("/new", galleriesC.New)
        r.Post("/", galleriesC.Create)
        r.Get("/{id}/edit", galleriesC.Edit)
        r.Post("/{id}", galleriesC.Update)
    })
})
```

Source Code

- [Source Code](#) - see the final source code for this lesson.
- [Diff](#) - see the changes made in this lesson.

21.20 Show Gallery Template and a Tailwind Update

We need a template for our page that renders a gallery, so let's add that to our code.

```
code templates/galleries/show.gohtml
```

This template is fairly short, so we can define the whole thing before adding it in our `main.go` source file.

```
{{template "header" .}}
<div class="px-8 py-12 w-full">
    <h1 class="pt-4 pb-8 text-3xl font-bold text-gray-900">
        {{.Title}}
```

```
</h1>
<div class="grid grid-cols-4 gap-4 space-y-4">
  {{range .Images}}
    <div class="h-min w-full">
      <a href="{{.}}>
        
  <meta charset="UTF-8" />
  <meta name="viewport" content="width=device-width, initial-scale=1.0" />
  <!-- Replace the <link> tag we have with this -->
  <script src="https://cdn.tailwindcss.com"></script>
</head>
```

This shouldn't break any of the styling we have, which is why we didn't rush to update it earlier in the course. It will cause our app to use the newest version of Tailwind CSS. We can worry about minifying the source when we prepare for production.

Now let's update our view to use the new class.

```
code templates/galleries/show.gohtml
```

```
<!-- Update the following line to use columns-4 instead of the grid -->


{{range .Images}}
    <div class="h-min w-full">
      <a href="{{.}}>
        /gallery-<id>/`.

Our GalleryService will need to know what the images directory is so it can query for images, so let's start by adding that.

```
code models/gallery.go
```

```
type GalleryService struct {
 DB *sql.DB

 // ImagesDir is used to tell the GalleryService where to store and locate
```

```
// images. If not set, the GalleryService will default to using the "images"
// directory.
ImagesDir string
}
```

We can also add a helper function that helps us generate the filepath to a specific gallery. This will also include the logic for handling the zero value of the **ImagesDir** field.

```
func (service GalleryService) galleryDir(id int) string {
 imagesDir := service.ImagesDir
 if imagesDir == "" {
 imagesDir = "images"
 }
 return filepath.Join(imagesDir, fmt.Sprintf("gallery-%d", id))
}
```

*Note: We use the **filepath** package to account for slashes going the opposite direction in some operating systems.*

With this default value we don't need to ever set the **ImagesDir** field, but if we wanted to customize it we could add something like **IMAGES\_DIR** to the **.env** file, then update our config to read this in and set it on the gallery service when setting it up. We won't be doing this, but feel free to experiment with it.

### Aside 22.1. Be Cautious of Images Getting Into Git

If the **ImagesDir** gets changed in a dev environment, our code will likely create that new directory and add images to it during local development. While this isn't an issue, it is likely that this directory won't be in the **.gitignore** file, so it could get added to the git repository and cause that to grow in size quite a bit with images that shouldn't be checked in. Be sure to add any directory you use to the **.gitignore** file, or make sure this is happening in an environment where

changes aren't checked into the source code, like in production where we will only deploying using our code, not push changes back to git.

## Source Code

- [Source Code](#) - see the final source code for this lesson.
- [Diff](#) - see the changes made in this lesson.

## 22.4 Globbing Image Files

First, we are going to create a type to return for our images and a method to query for all of a gallery's images. We will stub both of these to start, then fill them in as we go.

```
type Image struct {
 // TODO: Add fields to this type
}

func (service *GalleryService) Images(galleryID int) ([]Image, error) {
 // TODO: Implement this
}
```

We could opt to use `id` as the field name, but it is less obvious for this particular function what the ID is of, so it can be nice to use `galleryID` in cases like this to make it more clear.

Inside the `Images` function we are going to use `filepath.Glob` to get a list of all files in our image directory. We will then filter those files for images with supported extensions. The `Glob` function uses a `glob pattern` to find files. This is a pattern like `*.jpg` or `images/*`.

We can use the following code to create our pattern.

```
globPattern := filepath.Join(service.galleryDir(galleryID), "*)")
```

With the default images directory, a gallery with the ID **2**, and MacOS or Linux as our operating, this would give us a glob pattern of:

```
images/gallery-2/*
```

If we changed any of those details the pattern might vary, but hopefully an example helps illustrate what we are doing. We can then glob for all of the files using this pattern.

```
allFiles, err := filepath.Glob(globPattern)
if err != nil {
 return nil, fmt.Errorf("retrieving gallery images: %w", err)
}
```

We will want to filter for files with specific extensions, so let's add a function to help with that.

```
func hasExtension(file string, extensions []string) bool {
 for _, ext := range extensions {
 file = strings.ToLower(file)
 ext = strings.ToLower(ext)
 if filepath.Ext(file) == ext {
 return true
 }
 }
 return false
}
```

We need a set of extensions that we support when calling the **hasExtension** function. We can provide these via a reusable function for now, then later as an

exercise we can look at how to move this to a field on the `GalleryService` type similar to how the `ImagesDir` works.

```
func (service *GalleryService) extensions() []string {
 return []string{".png", ".jpg", ".jpeg", ".gif"}
}
```

This gives us a way to reuse the same extensions later when we limit what files can be uploaded for a gallery.

With that we can iterate through our files and only keep the ones with our desired extensions.

```
var imagePaths []string
for _, file := range allFiles {
 if hasExtension(file, service.extensions()) {
 imagePaths = append(imagePaths, file)
 }
}
```

Finally, we need to return something. Glob only returns the path to a file, but that is pretty useful and sounds like something we can add to our `Image` type for now.

```
type Image struct {
 Path string
}
```

Later if we decide other information would be more useful, like the filename, or the gallery ID, we can add that to the `Image` type.

We can now finalize the `Images` function by creating a slice of images and returning it.

```
func (service *GalleryService) Images(galleryID int) ([]Image, error) {
 globPattern := filepath.Join(service.galleryDir(galleryID), "*")
 allFiles, err := filepath.Glob(globPattern)
 if err != nil {
 return nil, fmt.Errorf("retrieving gallery images: %w", err)
 }
 var images []Image
 for _, file := range allFiles {
 if hasExtension(file, service.extensions()) {
 images = append(images, Image{
 Path: file,
 })
 }
 }
 return images, nil
}
```

We can even test this by updating our experimental code.

```
code cmd/exp/exp.go
```

We won't be using a database connection, so we only need to instantiate the `GalleryService` and call the `Images` function with the ID we used for our test images.

```
package main

import (
 "fmt"

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

func main() {
 gs := models.GalleryService{}
 fmt.Println(gs.Images(2))
}
```

## Source Code

- [Source Code](#) - see the final source code for this lesson.
- [Diff](#) - see the changes made in this lesson.

## 22.5 Adding Filename and GalleryID to the Image Type

We have a way to query for all of a gallery's images. Now we need a way of converting each image into a URL or path to the image in our app.

We are going to serve images at the path:

```
/galleries/{id}/images/{filename}
```

This will make it clear what gallery we are looking at, so we can do things like filter out unpublished galleries if we ever support those. It will also allow us to distinguish between two images with the same filename, but different galleries.

To do this, we need to update the `Image` type in the `models` package to include some additional information we need. Most notably, we can add the `GalleryID` and the `Filename` for each image.

```
code models/gallery.go
```

```
type Image struct {
 GalleryID int
 Path string
 Filename string
}
```

Next we can update the `Images` function to add this information.

```
func (service *GalleryService) Images(galleryID int) ([]Image, error) {
 // ...
 var images []Image
 for _, file := range allFiles {
 if hasExtension(file, service.extensions()) {
 images = append(images, Image{
 GalleryID: galleryID,
 Path: file,
 Filename: filepath.Base(file),
 })
 }
 }
 return images, nil
}
```

## Source Code

- [Source Code](#) - see the final source code for this lesson.
- [Diff](#) - see the changes made in this lesson.

## 22.6 Adding Images to the Show Gallery Page

Now that we have access to this information, we can use it in our **Show** handler for galleries.

```
code controllers/galleries.go
```

```
func (g Galleries) Show(w http.ResponseWriter, r *http.Request) {
 gallery, err := g.galleryByID(w, r)
 if err != nil {
 return
 }
 type Image struct {
 GalleryID int
 Filename string
 }
```

```

 }
 var data struct {
 ID int
 Title string
 Images []Image
 }
 data.ID = gallery.ID
 data.Title = gallery.Title
 images, err := g.GalleryService.Images(gallery.ID)
 if err != nil {
 fmt.Println(err)
 http.Error(w, "Something went wrong", http.StatusInternalServerError)
 return
 }
 for _, image := range images {
 data.Images = append(data.Images, Image{
 GalleryID: image.GalleryID,
 Filename: image.Filename,
 })
 }
 g.Templates.Show.Execute(w, r, data)
}

```

Then we can update the template.

```
code templates/galleries/show.gohtml
```

```

{{template "header" .}}
<div class="p-8 w-full">
 <h1 class="pt-4 pb-8 text-3xl font-bold text-gray-800">
 {{.Title}}
 </h1>
 <div class="columns-4 gap-4 space-y-4">
 {{range .Images}}
 <div class="h-min w-full">

 </div>
 {{end}}
 </div>
</div>
{{template "footer" .}}

```

*Note: Images won't render yet because we haven't setup that HTTP handler yet.*

## Source Code

- [Source Code](#) - see the final source code for this lesson.
- [Diff](#) - see the changes made in this lesson.

## 22.7 Show Image Handler

We have worked our way from our models to our views adding functionality to query all of a gallery's images and add them to the templates. We now need a way to render those individual images.

We need an HTTP handler for the path:

```
/galleries/:id/images/:filename
```

Let's work on adding a handler for this route.

```
code controllers/galleries.go
```

We will first parse the filename and the gallery ID.

```
func (g Galleries) Image(w http.ResponseWriter, r *http.Request) {
 filename := chi.URLParam(r, "filename")
 galleryID, err := strconv.Atoi(chi.URLParam(r, "id"))
 if err != nil {
 http.Error(w, "Invalid ID", http.StatusNotFound)
 return
 }
}
```

Next we can get all of the images for the gallery and look for our specific image. Long term we will want a way to query our GalleryService for a single image, but short term this works.

```
images, err := g.GalleryService.Images(galleryID)
if err != nil {
 fmt.Println(err)
 http.Error(w, "Something went wrong", http.StatusInternalServerError)
 return
}
var requestedImage models.Image
imageFound := false
for _, image := range images {
 if image.Filename == filename {
 requestedImage = image
 imageFound = true
 break
 }
}
```

Finally, we need to decide what to render. If the image isn't found, we want a not found page. Otherwise, we can use the `http.ServeFile` function to serve our image.

```
if !imageFound {
 http.Error(w, "Image not found", http.StatusNotFound)
 return
}
http.ServeFile(w, r, requestedImage.Path)
```

Putting this all together we get:

```
func (g Galleries) Image(w http.ResponseWriter, r *http.Request) {
 filename := chi.URLParam(r, "filename")
 galleryID, err := strconv.Atoi(chi.URLParam(r, "id"))
 if err != nil {
 http.Error(w, "Invalid ID", http.StatusNotFound)
 return
 }
 images, err := g.GalleryService.Images(galleryID)
 if err != nil {
```

```

 fmt.Println(err)
 http.Error(w, "Something went wrong", http.StatusInternalServerError)
 return
 }
 var requestedImage models.Image
 imageFound := false
 for _, image := range images {
 if image.Filename == filename {
 requestedImage = image
 imageFound = true
 break
 }
 }
 if !imageFound {
 http.Error(w, "Image not found", http.StatusNotFound)
 return
 }
 http.ServeFile(w, r, requestedImage.Path)
}

```

Next we need to update our routes.

code main.go

Find the section of the `main` function where we declare our gallery routes and add the following:

```

r.Route("/galleries", func(r chi.Router) {
 r.Get("/{id}", galleriesC.Show)
 r.Get("/{id}/images/{filename}", galleriesC.Image)
 // ...
}

```

We need to make sure this path is not inside the group that uses the require user middleware. We want our images to render for anyone, not just signed in users.

Now we can restart the server and verify that we can see images.

## Source Code

- [Source Code](#) - see the final source code for this lesson.
- [Diff](#) - see the changes made in this lesson.

## 22.8 Querying for a Single Image

Let's add a function to query for a single image and verify it exists.

```
code models/gallery.go
```

Define the function and create the image path.

```
func (service *GalleryService) Image(galleryID int, filename string) (Image, error) {
 imagePath := filepath.Join(service.galleryDir(galleryID), filename)
}
```

Next, use `os.Stat` to determine if the file exists. We don't need the file info, so we can just check the error to decide how to proceed. If there are no errors, we can return an image.

```
, err := os.Stat(imagePath)
if err != nil {
 return Image{}, fmt.Errorf("querying for image: %w", err)
}
return Image{
 Filename: filename,
 GalleryID: galleryID,
 Path: imagePath,
}, nil
```

Lastly, we can improve our error handling by checking to see if this error is because a file doesn't exist, or some thing else going wrong. If the file doesn't exist, we can use our `ErrNotFound` error in our `models` package.

```
if errors.Is(err, fs.ErrNotExist) {
 return Image{}, ErrNotFound
}
```

### Aside 22.2. Don't use os.IsNotExist

In the `os` package there is a function called `IsNotExist`. Historically, this was used to determine if the error meant that the file didn't exist. As the docs state, this predates the `errors.Is` function, and the approach we are using with `errors.Is` should be used instead.

This gives us the complete code below.

```
func (service *GalleryService) Image(galleryID int, filename string) (Image, error) {
 imagePath := filepath.Join(service.galleryDir(galleryID), filename)
 _, err := os.Stat(imagePath)
 if err != nil {
 if errors.Is(err, fs.ErrNotExist) {
 return Image{}, ErrNotFound
 }
 return Image{}, fmt.Errorf("querying for image: %w", err)
 }
 return Image{
 Filename: filename,
 GalleryID: galleryID,
 Path: imagePath,
 }, nil
}
```

Let's update the HTTP handler for images to use this new function.

```
code controllers/galleries.go
```

```
func (g Galleries) Image(w http.ResponseWriter, r *http.Request) {
 filename := chi.URLParam(r, "filename")
 galleryID, err := strconv.Atoi(chi.URLParam(r, "id"))
 if err != nil {
 http.Error(w, "Invalid ID", http.StatusNotFound)
 return
 }
 image, err := g.GalleryService.Image(galleryID, filename)
 if err != nil {
 if errors.Is(err, models.ErrNotFound) {
 http.Error(w, "Image not found", http.StatusNotFound)
 return
 }
 fmt.Println(err)
 http.Error(w, "Something went wrong", http.StatusInternalServerError)
 return
 }
 http.ServeFile(w, r, image.Path)
}
```

## Source Code

- [Source Code](#) - see the final source code for this lesson.
- [Diff](#) - see the changes made in this lesson.

## 22.9 URL Path Escaping Image Filenames

We are generating our paths for images in our templates using code similar to the example below.

```

```

Unfortunately, the `html/template` package doesn't escape the variables in a URL like this. That means if someone uploads a file with a character that isn't safe in a URL, it wouldn't work. An example would be the file `hello?.png`

The question mark character means something special in URLs, but is allowed in many file systems.

### Aside 22.3. Pups.zip Included a Test Image

In the [pups.zip](#) file that we used earlier to create testing images, one of the files has a filename with a question mark in it. It may not be easy to see, but if you look closely at the show gallery page, you should see an image that doesn't load at [/galleries/2/images/IMG\\_4?89.png](#).

The [url.PathEscape](#) function will help us fix this. We could use this in a number of places including:

1. Adding a [pathEscape](#) function to our templates.
2. Adding a [FilenameEscaped](#) field to our [Image](#) that we provide to the template.

We are going to go with the second option.

code controllers/galleries.go

Add the [FilenameEscaped](#) field, and assign it using the PathEscape function.

```
func (g Galleries) Show(w http.ResponseWriter, r *http.Request) {
 // ...
 type Image struct {
 GalleryID int
 Filename string
 FilenameEscaped string
 }
 // ...
 for _, image := range images {
 data.Images = append(data.Images, Image{
 GalleryID: image.GalleryID,
 Filename: image.Filename,
 FilenameEscaped: url.PathEscape(image.Filename),
 })
 }
}
```

```
 })
 }
 g.Templates.Show.Execute(w, r, data)
}
```

Now we can update our template to use this in URLs.

```
code templates/galleries/show.gohtml
```

We are going to find every place where we use `{{{.Filename}}}` in a URL path, and we are going to replace it with:

```
{{{.FilenameEscaped}}}
```

Looking at our template, that gives us the following:

```
{{template "header" .}}

{{{.Title}}}

{{range .Images}}
 <div class="h-min w-full">

 </div>
 {{end}}
 </div>

{{template "footer" .}}


```

## Source Code

- [Source Code](#) - see the final source code for this lesson.
- [Diff](#) - see the changes made in this lesson.

## 22.10 Adding Images to the Edit Gallery Page

Another page where we want to view our images is the edit gallery page.

```
code controllers/galleries.go
```

Update the `Edit` HTTP handler to add images much like the `Show` handler does.

```
func (g Galleries) Edit(w http.ResponseWriter, r *http.Request) {
 gallery, err := g.galleryByID(w, r, userMustOwnGallery)
 if err != nil {
 return
 }
 type Image struct {
 GalleryID int
 Filename string
 FilenameEscaped string
 }
 var data struct {
 ID int
 Title string
 Images []Image
 }
 data.ID = gallery.ID
 data.Title = gallery.Title
 images, err := g.GalleryService.Images(gallery.ID)
 if err != nil {
 fmt.Println(err)
 http.Error(w, "Something went wrong", http.StatusInternalServerError)
 return
 }
 for _, image := range images {
 data.Images = append(data.Images, Image{
 GalleryID: image.GalleryID,
 Filename: image.Filename,
 FilenameEscaped: url.PathEscape(image.Filename),
 })
 }
 g.Templates.Edit.Execute(w, r, data)
}
```

After that we can update our template.

```
code templates/galleries/edit.gohtml
```

Find the spot where the update form ends and the dangerous actions start. We will add our images in between these two sections of the page.

```
 {{template "header" .}}
<div class="p-8 w-full">
 <h1 class="pt-4 pb-8 text-3xl font-bold text-gray-800">
 Edit your Gallery
 </h1>
 <form action="/galleries/{{.ID}}" method="post">
 <!-- ... -->
 </form>
 <!-- Images -->
 <div class="py-4">
 <h2 class="pb-2 text-sm font-semibold text-gray-800">Current Images</h2>
 <div class="py-2 grid grid-cols-8 gap-2">
 {{range .Images}}
 <div class="h-min w-full relative">

 </div>
 {{end}}
 </div>
 <!-- Danger Actions -->
 <!-- ... -->
 </div>
{{template "footer" .}}
```

Restart the server and verify the changes are working as expected. Later we will add buttons to delete images and upload new images.

## Source Code

- [Source Code](#) - see the final source code for this lesson.
- [Diff](#) - see the changes made in this lesson.

## 22.11 Delete Image Form

In addition to showing each image when editing a gallery, we also need a way to delete images we no longer want.

We will start by adding the HTML form for this, then we can work on implementing things on our server.

```
code templates/galleries/edit.gohtml
```

Our HTML is getting larger, so it can help to break the forms into their own HTML templates.

```
{{define "delete_image_form"}}
<form action="/galleries/{{.GalleryID}}/images/{{.FilenameEscaped}}/delete"
method="post"
onsubmit="return confirm('Do you really want to delete this image?');">
{{csrfField}}
<button
type="submit"
class="
 p-1
 text-xs text-red-800
 bg-red-100
 border border-red-400
 rounded
">
 Delete
</button>
</form>
{{end}}
```

We can now add the form to our page using the template. This approach makes it easier to get the gist of the page without getting bogged down with all of the HTML.

Be aware that doing this too much might make tracking down some HTML harder. Digging 5 templates deep can get annoying, so it is a balancing act.

```
<!-- ... -->
<!-- Find where we have our current images -->

<h2 class="pb-2 text-sm font-semibold text-gray-800">Current Images</h2>
 <div class="py-2 grid grid-cols-8 gap-2">
 {{range .Images}}
 <div class="h-min w-full relative">
 <!-- Add this part -->
 <div class="absolute top-2 right-2">
 {{template "delete_image_form" .}}
 </div>

{{define "upload_image_form"}}
<form action="/galleries/{.ID}/images"
 method="post"
 enctype="multipart/form-data">
</form>
{{end}}
```

There is a good bit of new stuff here, so let's pause to talk about what is going on.

We are going to assume the path for our form to post to for now. We haven't set up a route for our path, but we are going to post to the following path to add images:

```
/galleries/:id/images
```

Our form also has an **enctype** for the first time. By default this will be:

```
application/x-www-form-urlencoded
```

We haven't set it before because this default is what we have wanted.

When uploading files, another encoding type is used:

```
multipart/form-data
```

The more general rule is that for binary data (files, images etc), or very large amounts of data, we should use **multipart/form-data**, otherwise we want to use the default of form urlencoded.

With that all covered we can finish up the form with a label, an input button for selecting multiple files, and the submit button.

```
{%define "upload_image_form"%}
<form action="/galleries/{{.ID}}/images"
 method="post"
 enctype="multipart/form-data">
 {{csrfField}}
 <div class="py-2">
 <label for="images" class="block mb-2 text-sm font-semibold text-gray-800">
 Add Images
 <p class="py-2 text-xs text-gray-600 font-normal">
 Please only upload jpg, png, and gif files.
 </p>
 </label>
 <input type="file" multiple
 accept="image/png, image/jpeg, image/gif"
 id="images" name="images" />
</div>
<button
 type="submit"
 class="py-2 px-8
 bg-indigo-600 hover:bg-indigo-700
 text-white text-lg font-bold
 rounded
 ">
 Upload
</button>
</form>
{{end}}
```

The HTML labels are likely familiar.

Our `<input>` tag has a type of `file`. The type being file allows us to upload a file, like an image, to add to our gallery. The `multiple` tag allows users to upload multiple files at once. The name is the key we will use to parse it on our server.

We also have the `accept` attribute. This limits out form to image files of the formats provided. While this is useful, it is worth noting that someone could alter the HTML in their browser to bypass this, so it isn't a reliable solution and we will need to validate things on our server as well.

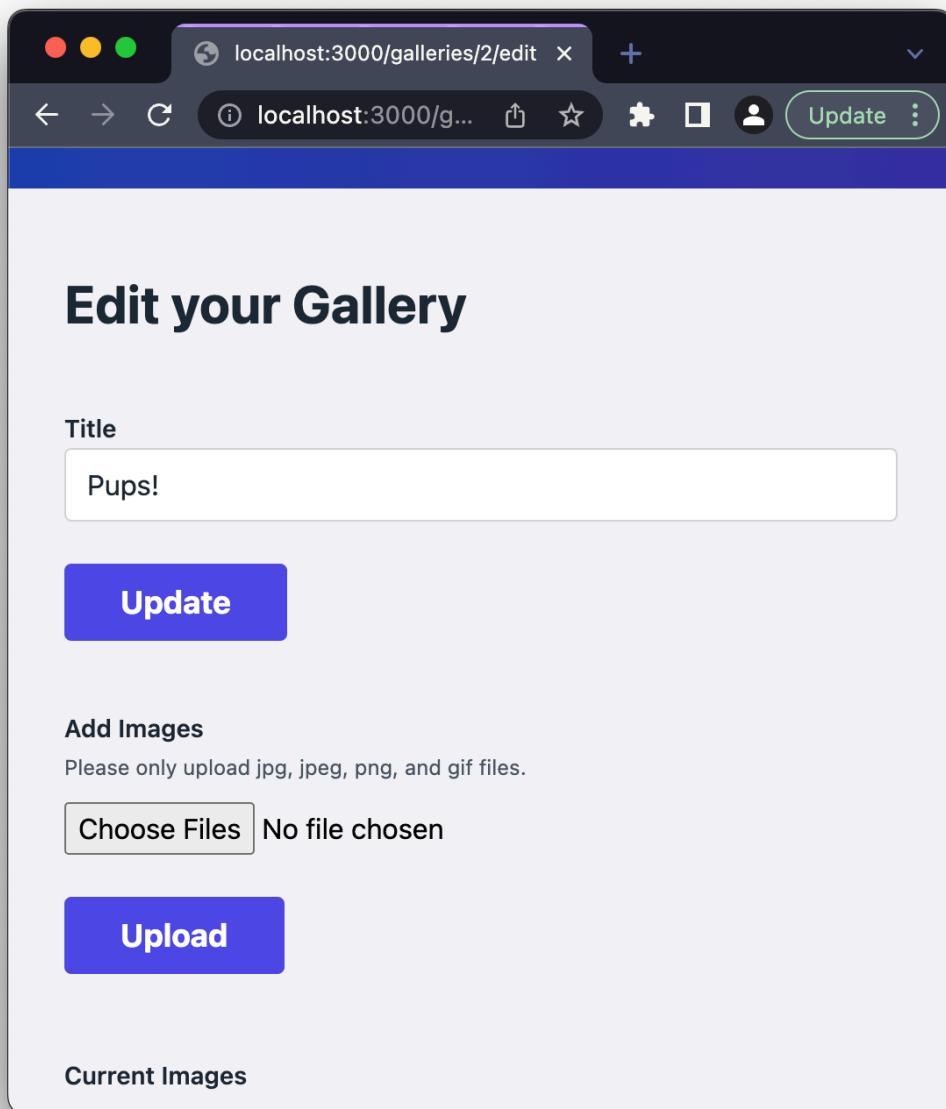
The rest of this is pretty much the same as things we have done already.

We defined this in a named template with the name `upload_image_form`, so

we need to tell our code where to render the form. Find the section of our code where we have the current images, and add the upload form just before it.

```
<!-- ... existing code for the update form -->
</form>
<!-- Add the following few lines -->
<div class="py-4">
 {{template "upload_image_form" .}}
</div>
<!-- Current Images section -->
<div class="py-4">
<!-- ... existing code with the current images -->
```

Reload the server and verify that the new form renders on the edit gallery page.



## Source Code

- [Source Code](#) - see the final source code for this lesson.

- [Diff](#) - see the changes made in this lesson.

## 22.16 Image Upload Handler

```
code controllers/galleries.go
```

We can start by verifying that the current user has access to this gallery. We always want to make sure users aren't altering galleries they do not have access to.

```
func (g Galleries) UploadImage(w http.ResponseWriter, r *http.Request) {
 gallery, err := g.galleryByID(w, r, userMustOwnGallery)
 if err != nil {
 return
 }
}
```

After that we need to parse our multipart form. To do this we use [ParseMultipartForm](#), which needs a maximum amount of memory to use. *Worth noting that it will still accept files larger than the max memory, it will just store the excess on disk rather than in memory.*

```
err = r.ParseMultipartForm(5 << 20) // 5mb
if err != nil {
 http.Error(w, "Something went wrong", http.StatusInternalServerError)
 return
}
```

### Aside 22.4. Bit Shifts

In computers, integers are stored as binary data using base 2. The numbers 1 to 5 in binary are:

```
001 // 1
010 // 2
011 // 3
100 // 4
101 // 5
```

We can see this using the following Go code to print out numbers in their binary representation:

```
package main

import "fmt"

func main() {
 for i := 0; i < 10; i++ {
 fmt.Printf("%4b\n", i)
 }
}
```

`%b` says to print a number in binary. The `%4b` tells our code to left-pad the numbers so they are 4 characters wide. This is helpful to print them out right-aligned.

The `<<` and `>>` operators are used to shift bits. When we write `1 << 20` we are saying that we want to shift the bits from the number `1` left 20 times, giving us the binary number:

```
10000000000000000000000000
```

This is equivalent to 1,048,576, which is 1 megabyte. `1 << 10` is 1024, which is one kilobyte. `1 << 20` is just a short way of getting one mb.

When we write `5 << 20` we are taking the bits from 5, which are `101`, and then shifting those left 20 times giving us:

10100000000000000000000000

If we convert this to a decimal number we get 5242880, which is  $5 * 1048576$ , or 5mb.

Bit shifts aren't incredibly common, but can be seen in code and used for various things, so it is useful to become familiar with what they are on the off chance you encounter them.

We could have also just written `5 * 1024 * 1024` for 5mb, so feel free to do that if it is clearer to you.

Once our multipart form is parsed we can access the files in it through the `MultipartForm` field on the `http.Request`. This has the type `*multipart.Form`. If we dig through the `File` field of this type, we will see that it is a map of `FileHeaders`. While the field is called `File`, it maps to multiple file headers. With these file headers we can open each file uploaded from a form to read the data and store our image wherever we need.

```
fileHeaders := r.MultipartForm.File["images"]
for _, fileHeader := range fileHeaders {
 file, err := fileHeader.Open()
 if err != nil {
 http.Error(w, "Something went wrong", http.StatusInternalServerError)
 return
 }
 defer file.Close()
 // ... Do something with the file
}
```

We can also access the original filename through the **Filename** field on the **FileHeader**.

```
fmt.Printf("Attempting to upload %v for gallery %d.\n",
 fileHeader.Filename, gallery.ID)
```

Let's use all of this to make a temporary handler that we can test out. We will take the first image uploaded, and copy it to the response. This should render the image as the response of our web request.

```
func (g Galleries) UploadImage(w http.ResponseWriter, r *http.Request) {
 gallery, err := g.galleryByID(w, r, userMustOwnGallery)
 if err != nil {
 return
 }
 err = r.ParseMultipartForm(5 << 20) // 5mb
 if err != nil {
 http.Error(w, "Something went wrong", http.StatusInternalServerError)
 return
 }
 fileHeaders := r.MultipartForm.File["images"]
 for _, fileHeader := range fileHeaders {
 file, err := fileHeader.Open()
 if err != nil {
 http.Error(w, "Something went wrong", http.StatusInternalServerError)
 return
 }
 defer file.Close()
 fmt.Printf("Attempting to upload %v for gallery %d.\n",
 fileHeader.Filename, gallery.ID)
 io.Copy(w, file)
 return
 }
}
```

Add this to our routes so we can verify it works.

```
code main.go
```

```
r.Route("/galleries", func(r chi.Router) {
 r.Get("/{id}", galleriesC.Show)
 r.Group(func(r chi.Router) {
 r.Use(umw.RequireUser)
```

```
// ...
// Add this line inside the group that requires a user
r.Post("/{id}/images", galleriesC.UploadImage)
})
})
```

Now test it out. If we go to the edit gallery page and attempt to upload an image, we should see it in our browser window after submitting the form. We should also see a log message similar to the one below.

```
Attempting to upload test_file.jpg for gallery 2.
```

That should be enough insight into how the form works to start thinking about how the gallery service needs updated to handle image creation.

## Source Code

- [Source Code](#) - see the final source code for this lesson.
- [Diff](#) - see the changes made in this lesson.

## 22.17 Creating Images in the GalleryService

```
code models/gallery.go
```

```
func (service *GalleryService) CreateImage(galleryID int, filename string, contents io.Reader)
 galleryDir := service.galleryDir(galleryID)
 err := os.MkdirAll(galleryDir, 0755)
 if err != nil {
 return fmt.Errorf("creating gallery-%d images directory: %w", galleryID, err)
 }
 imagePath := filepath.Join(galleryDir, filename)
```

```

 dst, err := os.Create(imagePath)
 if err != nil {
 return fmt.Errorf("creating image file: %w", err)
 }
 defer dst.Close()

 _, err = io.Copy(dst, contents)
 if err != nil {
 return fmt.Errorf("copying contents to image: %w", err)
 }
 return nil
 }
}

```

Update the controller

```

func (g Galleries) UploadImage(w http.ResponseWriter, r *http.Request) {
 // ... unchanged
 fileHeaders := r.MultipartForm.File["images"]
 for _, fileHeader := range fileHeaders {
 file, err := fileHeader.Open()
 if err != nil {
 http.Error(w, "Something went wrong", http.StatusInternalServerError)
 return
 }
 defer file.Close()

 err = g.GalleryService.CreateImage(gallery.ID, fileHeader.Filename, file)
 if err != nil {
 http.Error(w, "Something went wrong", http.StatusInternalServerError)
 return
 }
 }
 editPath := fmt.Sprintf("/galleries/%d/edit", gallery.ID)
 http.Redirect(w, r, editPath, http.StatusFound)
}

```

## Source Code

- [Source Code](#) - see the final source code for this lesson.
- [Diff](#) - see the changes made in this lesson.

## 22.18 Detecting Content Type

While we have taken some precautions in our HTML form, we need to do server-side validation of the files being uploaded.

The `net/http` package has a `DetectContentType` function that can be useful to determine the content type of a file.

We are going to utilize it to add some validation to our `models` package. This will be a function that can check if any file matches a specific content type, and if not returns a custom error.

code `models/errors.go`

First, we start with the an error type. We will use a more generic file error, so we can reuse it later if we have other files being uploaded. The goal here is to have a way to know if there is an issue with the file provided.

```
type FileError struct {
 Issue string
}

func (fe FileError) Error() string {
 return fmt.Sprintf("invalid file: %v", fe.Issue)
}
```

Next we write the code to check content types. We start by defining the function. We need to accept an `io.ReadSeeker` so that we can reset the file after reading some of it. Our file submitted via the HTML form meets this criteria, and we will just need to update a few function arguments later to make it work.

```
func checkContentType(r io.ReadSeeker, allowedTypes []string) error {
 return nil
}
```

Next we can start implementing it. `http.DetectContentType` We can use the `ReadSeeker` passed into our function to read these bytes, but we first need a byte slice.

```
testBytes := make([]byte, 512)
_, err := r.Read(testBytes)
if err != nil {
 return fmt.Errorf("checking content type: %w", err)
}
```

We are creating a slice with 512 bytes. It will read less if the file is smaller. This is then passed into the `DetectContentType` function to determine the content type.

After that we reset the reader.

```
, err = r.Seek(0, 0)
if err != nil {
 return fmt.Errorf("checking content type: %w", err)
}
```

Then we check the content type of our bytes. Return `nil` if it is an allowed content type. Otherwise we need to return a `FileError`

```
contentType := http.DetectContentType(testBytes)
for _, t := range allowedTypes {
 if contentType == t {
 return nil
 }
}
return FileError{
 Issue: fmt.Sprintf("invalid content type: %v", contentType),
}
```

Putting this all together we get the following function:

```
func checkContentType(r io.ReadSeeker, allowedTypes []string) error {
 testBytes := make([]byte, 512)
 _, err := r.Read(testBytes)
 if err != nil {
 return fmt.Errorf("checking content type: %w", err)
 }
 _, err = r.Seek(0, 0)
 if err != nil {
 return fmt.Errorf("checking content type: %w", err)
 }

 contentType := http.DetectContentType(testBytes)
 for _, t := range allowedTypes {
 if contentType == t {
 return nil
 }
 }
 return FileError{
 Issue: fmt.Sprintf("invalid content type: %v", contentType),
 }
}
```

We can also add a function to check for valid extensions. This is much shorter because we already code the logic to check if a file has a specific extension.

```
func checkExtension(filename string, allowedExtensions []string) error {
 if !hasExtension(filename, allowedExtensions) {
 return FileError{
 Issue: fmt.Sprintf("invalid extension: %v", filepath.Ext(filename)),
 }
 }
 return nil
}
```

## Source Code

- [Source Code](#) - see the final source code for this lesson.
- [Diff](#) - see the changes made in this lesson.

## 22.19 Rendering Content Type Errors

```
code models/gallery.go
```

First, update the `CreateImage` function to accept an `io.ReadSeeker` instead of an `io.Reader`. This works because our file we open from the HTML form also implements the `io.Seeker` interface.

```
// Change to io.ReadSeeker & add error check at start.
func (service *GalleryService) CreateImage(galleryID int, filename string, contents io.ReadSeeker
 // ...
}
```

We need to specify which image content types we support. We will hard-code this in a function for now.

```
func (service *GalleryService) imageContentTypes() []string {
 return []string{"image/png", "image/jpg", "image/gif"}
}
```

Next, add the code to check our file's contents and return an error. We also want to verify that it has a valid extension. Add the following to the start of the function.

```
err := checkContentType(contents, service.imageContentTypes())
if err != nil {
 return fmt.Errorf("creating image %v: %w", filename, err)
}
if !hasExtension(filename, service.extensions()) {
 return fmt.Errorf("creating image %v: %w", filename, err)
}
```

Now we need to update our controller to render this error differently.

```
code controllers/galleries.go
```

```
func (g Galleries) UploadImage(w http.ResponseWriter, r *http.Request) {
 // ...
 fileHeaders := r.MultipartForm.File["images"]
 for _, fileHeader := range fileHeaders {
 // ...

 err = g.GalleryService.CreateImage(gallery.ID, fileHeader.Filename, file)
 if err != nil {
 // Add some extra error handling.
 var fileErr models.FileError
 if errors.As(err, &fileErr) {
 msg := fmt.Sprintf("%v has an invalid content type or extension", fileHeader.Filename)
 http.Error(w, msg, http.StatusBadRequest)
 return
 }
 fmt.Println(err)
 http.Error(w, "Something went wrong", http.StatusInternalServerError)
 return
 }
 editPath := fmt.Sprintf("/galleries/%d/edit", gallery.ID)
 http.Redirect(w, r, editPath, http.StatusFound)
 }
}
```

To test this, remove the `accept="..."` part of our upload image form in the `edit.gohtml` template. Be sure to add this back in before moving forward.

## Source Code

- [Source Code](#) - see the final source code for this lesson.
- [Diff](#) - see the changes made in this lesson.

## 22.20 Deleting Images on Gallery Deletion

```
code models/gallery.go
```

```
func (service *GalleryService) Delete(id int) error {
 _, err := service.DB.Exec(`DELETE FROM galleries
 WHERE id = $1;`, id)
 if err != nil {
 return fmt.Errorf("delete gallery: %w", err)
 }
 err = os.RemoveAll(service.galleryDir(id))
 if err != nil {
 return fmt.Errorf("delete gallery images: %w", err)
 }
 return nil
}
```

## Source Code

- [Source Code](#) - see the final source code for this lesson.
- [Diff](#) - see the changes made in this lesson.

## 22.21 Redirect to Galleries After Auth

```
code controllers/users.go
```

```
func (u Users) Create(w http.ResponseWriter, r *http.Request) {
 // ...
 http.Redirect(w, r, "/galleries", http.StatusFound)
}
```

```
func (u Users) ProcessSignIn(w http.ResponseWriter, r *http.Request) {
 // ...
 http.Redirect(w, r, "/galleries", http.StatusFound)
}
```

## Source Code

- [Source Code](#) - see the final source code for this lesson.
- [Diff](#) - see the changes made in this lesson.

## 22.22 Image Exercises

### 22.22.1 Ex1 - Make the image extensions customizable

The image extensions allowed are currently hard-coded in our **GalleryService** type.

```
func (service *GalleryService) extensions() []string {
 return []string{".png", ".jpg", ".jpeg", ".gif"}
}
```

Add a new field to the **GalleryService** type that allows these to be set when we setup our server, and make sure the default values work if this field isn't set.

### 22.22.2 Ex2 - Make the image content types customizable

The image content types allowed are currently hard-coded in our **GalleryService** type.

```
func (service *GalleryService) imageContentTypes() []string {
 return []string{"image/png", "image/jpeg", "image/gif"}
}
```

Add a new field to the `GalleryService` type that allows these to be set when we setup our server, and make sure the default values work if this field isn't set.

### 22.22.3 Ex3 - Create an ImageService

We currently stick all of the image logic inside our `GalleryService`. Create an `ImageService` that contains all of this logic and see how that compares.

Do you like having more services, or does the extra setup make it more frustrating?

Determining how to organize our code is always about tradeoffs, so it is a good idea to explore various techniques so we better understand those tradeoffs.

# Chapter 23

## Preparing for Production

### 23.1 Loading All Config via ENV

At the moment, our web server starts up via `main.go`, while the rest of our commands are in the `cmd` directory.

Let's move our web server to the cmd directory while refactoring it a bit to make it easier to test if we want to in the future.

```
mkdir cmd/server
mv main.go cmd/server/server.go
code cmd/server/server.go
```

Now we would start our server with:

```
go run cmd/server/server.go
```

That means we need to update `modd` if we are using it.

```
code modd.conf
```

The go build command needs updated.

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

/*.go !/*_test.go **/*.gohtml {
 prep: go build -o lenslocked ./cmd/server/
 daemon +sigterm: ./lenslocked
}
```

Now we are going to refactor a bit. Our two goals are:

1. Extract everything configurable or sensitive into an config variable. Eg the CSRF key.
2. Move most of our server code into a function that is easier to test.

We will focus on the first goal for now - extracting our config fully. We need to do this so we don't have any real live secrets committed to our source. That means if we had anything like the CSRF key in our source code, we need to use a new value when we go to production.

```
func loadEnvConfig() (config, error) {
 var cfg config
 err := godotenv.Load()
 if err != nil {
 return cfg, err
 }
 cfg.PSQL = models.PostgresConfig{
 Host: os.Getenv("PSQL_HOST"),
 Port: os.Getenv("PSQL_PORT"),
 User: os.Getenv("PSQL_USER"),
 Password: os.Getenv("PSQL_PASSWORD"),
 Database: os.Getenv("PSQL_DATABASE"),
 }
}
```

```

 SSLMode: os.Getenv("PSQL_SSLMODE"),
 }
 if cfg.PSQL.Host == "" && cfg.PSQL.Port == "" {
 return cfg, fmt.Errorf("No PSQL Config provided.")
 }

 cfg.SMTP.Host = os.Getenv("SMTP_HOST")
 portStr := os.Getenv("SMTP_PORT")
 cfg.SMTP.Port, err = strconv.Atoi(portStr)
 if err != nil {
 return cfg, err
 }
 cfg.SMTP.Username = os.Getenv("SMTP_USERNAME")
 cfg.SMTP.Password = os.Getenv("SMTP_PASSWORD")

 cfg.CSRF.Key = os.Getenv("CSRF_KEY")
 cfg.CSRF.Secure = os.Getenv("CSRF_SECURE") == "true"

 cfg.Server.Address = os.Getenv("SERVER_ADDRESS")

 return cfg, nil
}

```

Next, update the `.env` file.

```
code .env
```

Be sure to fill in your own values for things like the SMTP username and password, and the CSRF key. This is test

```

SMTP_HOST=sandbox.smtp.mailtrap.io
SMTP_PORT=587
SMTP_USERNAME=<your username>
SMTP_PASSWORD=<your password>

PSQL_HOST=localhost
PSQL_PORT=5432
PSQL_USER=baloo
PSQL_PASSWORD=junglebook
PSQL_DATABASE=lenslocked
PSQL_SSLMODE=disable

CSRF_KEY=<32 byte string>

```

```
CSRF_SECURE=false
SERVER_ADDRESS=localhost:3000
```

It is also a good idea to update the template.

```
code .env.template
```

This should mostly just have default values and empty keys so a developer knows what to fill in for their own `.env` file.

```
SMTP_HOST=sandbox.smtp.mailtrap.io
SMTP_PORT=587
SMTP_USERNAME=<your username>
SMTP_PASSWORD=<your password>

PSQL_HOST=localhost
PSQL_PORT=5432
PSQL_USER=baloo
PSQL_PASSWORD=junglebook
PSQL_DATABASE=lenslocked
PSQL_SSLMODE=disable

CSRF_KEY=<32 byte string>
CSRF_SECURE=false

SERVER_ADDRESS=localhost:3000
```

## Source Code

- [Source Code](#) - see the final source code for this lesson.
- [Diff](#) - see the changes made in this lesson.

## 23.2 Docker Compose Overrides

All of our docker config is in a single `docker-compose.yml` file.

Works well when getting started, but if we want to use docker to deploy, we will need different settings there.

We can approach this a few ways:

1. Use override files
2. Use a separate dockerfile for production

The way the first works, is we can define a base dockerfile for everything we want in ALL docker configs, then we can use overrides to add environment specific stuff.

We would likely have three files defining our docker services: - the base - the dev override - the prod override

Upside to this is that we can define the def override with the following name:

```
code docker-compose.override.yml
```

And it will automatically be used as an override.

In prod we have to use the `-f` flag to specify which docker compose files we want to use.

Let's start by creating our base file

```
code docker-compose.yml
```

In this file we only want things that will be present in ALL environments. We can override specific settings, but we can't remove services in an override.

We are going to start by stripping out everything, then we can slowly add things back in via overrides as necessary.

```
Only put things in this file that we want in *every* environment
version: "3.9"

services:

 db:
 image: postgres
 restart: always
 environment:
 POSTGRES_USER: ${SQL_USER}
 POSTGRES_PASSWORD: ${SQL_PASSWORD}
 POSTGRES_DB: ${SQL_DATABASE}
```

We won't want adminer in prod.

We also don't want to expose the DB ports in prod. Instead, if our Go app runs in the same docker network it can access the DB while it isn't exposed elsewhere. This helps keep our DB a bit more secure.

We also are going to use the ENV variables for the user name, password, and DB name. This works b/c docker reads from a `.env` file automatically.

We can now create a dev override that adds all the things we previously had.

```
code docker-compose.override.yml
```

```
override defines changes to services and new services that we want to use in development.
version: "3.9"

services:
 [REDACTED] # All settings from the base docker-compose.yml will be used and we can change or add new ones
 db:
 ports:
```

```
[REDACTED] # We expose the DB ports so that apps not running via docker-compose can
- 5432:5432

Adminer provides a nice little web UI to connect to databases
adminer:
 image: adminer
 restart: always
 environment:
 ADMINER_DESIGN: dracula
 ports:
 - 3333:8080
```

Verify this all worked by recreating everything.

```
The --remove-orphans flag removes containers we may not have defined in our
docker-compose, but are still present from old configs.
docker compose down --remove-orphans
docker compose up -d
```

Verify the DB is there (easiest way - run the app and verify it works)

Verify Adminer is accessible at :3333

Later we can provide a docker compose file for our prod setup.

## Source Code

- [Source Code](#) - see the final source code for this lesson.
- [Diff](#) - see the changes made in this lesson.

## 23.3 Building Tailwind Locally

We are going to install Tailwind and build our own stylesheet. The main goals here are to:

1. Reduce the size of our CSS file that we serve.
2. Enable the easy use of custom CSS with Tailwind.

The first is achieved by removing all Tailwind CSS classes that our application doesn't use. We will set up the Tailwind CLI to look over all of our `.gohtml` templates and find CSS classes we use. Any others will be stripped from the resulting CSS. By default, Tailwind has something like 9MB of CSS generated, but any normal install is likely going to be 10-20KB.

The second goal is achieved by writing our own `styles.css` file and providing it to Tailwind when we build.

In this lesson we will walk through the Tailwind build process using the Tailwind CLI and installing everything locally.

In the next lesson we will use Docker to build with Tailwind. We do this to make sure this works for everyone regardless of OS or currently installed software. The downside is that it is harder to customize with plugins and can be confusing for beginners. It also isn't clear what needs done to minify.

*My preference is a locally installed CLI, but both options are here, so don't let this lesson stop you if you cannot get it installed correctly for some reason.*

First, you will need `node` and `npm` first. Instructions can be found [in the npm docs](#).

Verify they are working and return a valid version.

```
node -v
npm -v
```

After that you will need to follow the [Tailwind Docs](#) for installing and setting up via the CLI in a directory named `tailwind`.

```
mkdir tailwind
cd tailwind
```

Install Tailwind and open the generated config.

```
npm install -D tailwindcss
npx tailwindcss init
code tailwind.config.js
```

Update the config with the location of our templates.

```
/** @type {import('tailwindcss').Config} */
module.exports = {
 content: [
 "../templates/**/*.{gohtml,html}",
],
 theme: {
 extend: {},
 },
 plugins: [],
}
```

Create a stylesheet as an input, and tell it to include the base Tailwind components.

```
code styles.css
```

```
@tailwind base;
@tailwind components;
@tailwind utilities;

.test {
 font-size: 10rem;
}

.another-test {
 font-size: 2rem;
}
```

Build Tailwind.

```
npx tailwindcss -i ./styles.css -o ../assets/styles.css --watch
```

This will generate an output CSS file for our app in the assets dir.

Probably a good idea to make sure the `node_modules` dir isn't committed to source code.

```
code .gitignore
```

Add the following.

```
Node stuff
node_modules/
```

We will see how to render our pages using 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.

## 23.4 Tailwind Via Docker

### Aside 23.1. Performance Warning

In this section we look at building Tailwind via Docker. In my experience, this can slow the performance of your computer a good bit due to how we are polling for changes, so I would recommend running Tailwind locally most of the time and not using this approach unless you have issues getting Tailwind to run.

Rather than installing the Tailwind CLI locally, we will be setting it up via Docker in this lesson.

We will store all the files we use for setup in the `tailwind` directory. We will put our output files in `/assets`, so we can make that as well if it wasn't created last lesson

```
Make these if you didn't create them last lesson
mkdir tailwind
mkdir assets
```

The first thing we need is a `Dockerfile` defining our docker image.

```
code tailwind/Dockerfile
```

Add the following contents. They might not all make sense, but we will walk over how this works in a bit.

```
Start with an image with node installed
FROM node:latest

Create the directories we need
RUN mkdir /tailwind

Set /tailwind as the workdir.
A workdir is required for npm to work correctly.
WORKDIR /tailwind
```

```
Install tailwindcss and initialize
RUN npm init -y && \
 npm install tailwindcss && \
 npx tailwindcss init

Run tailwindcss. This will watch for changes in /src/styles.css and output to /dst/styles.css
CMD npx tailwindcss -c /src/tailwind.config.js -i /src/styles.css -o /dst/styles.css --watch --
```

At a high level, all this file does is define a docker image with node, npm, and Tailwind installed. It then runs this and watches for changes to our input CSS file.

To use this docker image, we are going to need to mount a few directories. We need to do this so we can update files on our system and Tailwind can react to those changes. We also need to do this so we can get the output from Tailwind running inside the docker container in our local directory.

The directories we will use are:

- **/src** - This directory provides input files for running Tailwind. This includes our custom styles, and our config.
- **/templates** - This directory has all of our HTML files. In our case, **.gohtml** files. We provide this so that Tailwind can remove CSS that we don't need and trim down the size of our stylesheet.
- **/dst** - where the final css will be written.

Next, let's set up our Tailwind config.

```
code tailwind/tailwind.config.js
```

This will be similar to a config Tailwind initializes when setting it up locally, except it will point to the template location we plan to use in our container.

```
/** @type {import('tailwindcss').Config} */
module.exports = {
 content: [
 "/templates/**/*.{gohtml, html}",
],
 theme: {
 extend: {},
 },
 plugins: [
],
}
```

*Note: If setting up Tailwind locally and not using Docker, the content directory needs updated to our templates directory which is a relative path from where you are running Tailwind.*

After that we need an input stylesheet.

```
code tailwind/styles.css
```

Again, this will mimic what is used for a new Tailwind install.

```
@tailwind base;
@tailwind components;
@tailwind utilities;

/* Add custom CSS here if you need it. */
```

We now have all of our input files, so we can proceed with setting up Docker compose. This will handle mounting those volumes we need.

```
code docker-compose.yml
```

Add the following service.

```
tailwind:
 build:
 context: ./tailwind
 dockerfile: Dockerfile
 tty: true
 restart: always
 volumes:
 - ./tailwind:/src
 - ./assets:/dst
 - ./templates:/templates
```

Finally, start up the new service with docker compose.

```
docker compose up -d
```

We should now have a new CSS file in our **assets** directory with all of the Tailwind CSS we need!

We can also build manually

```
docker compose run tailwind npx tailwindcss -c /src/tailwind.config.js -i /src/styles.css -o /d
```

We can even add the **-minify** tag to minify our output.

## Source Code

- [Source Code](#) - see the final source code for this lesson.
- [Diff](#) - see the changes made in this lesson.

## 23.5 Serving Static Assets

Our next step is to setup our app to serve our **styles.css** file.

We will setup our app to serve static files from the assets directory. To do this, we need to add a new route.

```
code cmd/server/server.go
```

Add the following to our routes, then we can talk about how it works.

```
assetsHandler := http.FileServer(http.Dir("assets"))
r.Get("/assets/*", http.StripPrefix("/assets", assetsHandler).ServeHTTP)
```

We first have the FileServer and `http.Dir`. `http.Dir` is a type in the `net/http` package that is used to signify a directory. If we look at the `http.Dir` type, it is really a string.

```
type Dir string
```

When we write `http.Dir(cfg.ImagesDir)` we are converting our string into the `http.Dir` type, not calling a function. What makes `http.Dir` useful to us now is that it has the `Open` method.

Surrounding `http.Dir` is the `http.FileServer` function call. This function takes in an `http.FileSystem` and returns an HTTP handler that serves files from that file system.

A `FileSystem` is an interface that defines a type with the `Open` method:

```
type FileSystem interface {
 Open(name string) (File, error)
}
```

By converting our directory from a string into an `http.Dir`, we are turning it into an `http.FileSystem` as well because `http.Dir` implements that interface.

After this we have a call to the `http.StripPrefix` function. This is an HTTP handler that works like the middleware we explored earlier. It accepts an `http.Handler` as an argument, and then returns a new `http.Handler` that will remove a prefix from the URL Path prior to calling the `http.Handler` being wrapped. In our case, it removes the `/assets` portion of the path before our `http.FileServer` handler runs.

We remove the `/assets` prefix because the HTTP FileServer looks for a file using the entire URL Path. In other words, if we visited “/assets/styles.css” and did not strip the prefix, our FileServer would attempt to find a file at “/assets/assets/styles.css” (**notice the word “assets” is repeated in the path**). By stripping out the `/assets` prefix, we can include it as part of our routing path, but remove it so the FileServer still works as expected.

Finally, we use `.ServeHTTP` at the end of all of this because `r.Get` expects an `http.HandlerFunc`, not an `http.Handler`, and `ServeHTTP` matches the definition of a `HandlerFunc`.

With this all in place we can update our templates to use our own CSS file rather than the Tailwind JS.

```
code templates/tailwind.gohtml
```

Remove the `<script>` tag that uses the Tailwind CDN and add the following in its place.

```
<!-- Remove this line -->
<!-- <script src="https://cdn.tailwindcss.com"></script> -->
<!-- Replace it with this line -->
<link rel="stylesheet" href="/assets/styles.css" />
```

Restart the app and verify the style sheets are working. Also visit `/assets/styles.css` to verify the CSS file is returned by our server.

## Source Code

- [Source Code](#) - see the final source code for this lesson.
- [Diff](#) - see the changes made in this lesson.

## 23.6 Making main Easier to Test

We are going to move our code into a `run()` function. If it stays in main, we can only test the server with those configs. We use ENV, so that isn't awful, but ideally we could just use a function or type we pass a config into, then we can more easily test. See [this article](#) for some context on why this should be considered.

code cmd/server.go

We can start by just splitting `main()` into two functions. This means we barely change any code.

```
func main() {
 cfg, err := loadEnvConfig()
 if err != nil {
 panic(err)
 }
 err = run(cfg)
 if err != nil {
 panic(err)
 }
}

func run(cfg config) error {
 // Setup the database
 db, err := models.Open(cfg.PSQL)
 if err != nil {
 panic(err)
 }
 defer db.Close()

 // ... the rest of the code that was in `main()` is in the run func now
}
```

Now we can slowly refactor `run` to return an error anywhere one occurs.

```
func run(cfg config) error {
 // Setup the database
 db, err := models.Open(cfg.PSQL)
 if err != nil {
 return err
 }
 defer db.Close()

 err = models.MigrateFS(db, migrations.FS, ".")
 if err != nil {
 return err
 }

 // ...

 // Start the server
 fmt.Printf("Starting the server on %s...\n", cfg.Server.Address)
 return http.ListenAndServe(cfg.Server.Address, r)
}
```

More refactoring could be done if desired. Much of this code could be moved into an isolated `Server` type, or we could move the routes to a function of its own to make them easier to find. This all boils down to what works best for each team and project for managing things.

## Source Code

- [Source Code](#) - see the final source code for this lesson.
- [Diff](#) - see the changes made in this lesson.

## 23.7 Running our Go Server via Docker

```
code Dockerfile
```

```
FROM golang
WORKDIR /app
COPY go.mod go.sum .
RUN go mod download
COPY .
RUN go build -v -o ./server ./cmd/server/
CMD ./server
```

```
code docker-compose.production.yml
```

```
version: "3.9"

services:
 server:
 build:
 context: ./
 dockerfile: Dockerfile
 restart: always
 volumes:
 - ./images:/app/images
 ports:
 - 3000:3000
 depends_on:
 - db
```

Stop your server if you have it running.

```
docker compose \
-f docker-compose.yml \
-f docker-compose.production.yml \
up
```

This will error because our PSQL\_HOST is localhost. We need this to be **db** now because that is the name of our service inside docker compose.

```
code .env
```

```
PSQL_HOST=db
```

It is also worth ensuring that your **SERVER\_ADDRESS** is set to **:3000** and NOT **localhost:3000**, as the latter will not work when running our Go code via docker.

Restart your server with docker.

```
docker compose \
-f docker-compose.yml \
-f docker-compose.production.yml \
up
```

Cleanup

```
docker compose -f docker-compose.yml -f docker-compose.production.yml rm server
```

If we plan to use **modd** we should also change the **PSQL\_HOST** to be **localhost** as well. The **db** part will only be necessary in production.

```
code .env
```

```
PSQL_HOST=localhost
```

## Source Code

- [Source Code](#) - see the final source code for this lesson.
- [Diff](#) - see the changes made in this lesson.

## 23.8 Multi-Stage Docker Builds

```
FROM golang AS builder
WORKDIR /app
COPY go.mod go.sum .
RUN go mod download
COPY .
RUN go build -v -o ./server ./cmd/server/

FROM ubuntu
WORKDIR /
COPY ./assets ./assets
COPY .env .env
COPY --from=builder /app/server ./server
CMD ["/server"]
```

docker-compose.production.yml:

```
version: "3.9"

services:
 server:
 build:
 context: ./go.prod.Dockerfile
 restart: always
 volumes:
 - ./assets:/assets
 - ./images:/images
 - .env:/.env
 ports:
 - 3000:3000
 depends_on:
 - db
```

Try running it:

```
docker compose -f docker-compose.yml -f docker-compose.production.yml up --build
```

*Sidenote: We could use docker to provide the ENV for prod.*

## Source Code

- [Source Code](#) - see the final source code for this lesson.
- [Diff](#) - see the changes made in this lesson.

## 23.9 Tailwind Production Build

```
FROM node:latest AS tailwind-builder
WORKDIR /tailwind
RUN npm init -y && \
 npm install tailwindcss && \
 npx tailwindcss init
COPY ./templates /templates
COPY ./tailwind/tailwind.config.js /src/tailwind.config.js
COPY ./tailwind/styles.css /src/styles.css
RUN npx tailwindcss -c /src/tailwind.config.js -i /src/styles.css -o /styles.css

FROM golang AS builder
WORKDIR /app
COPY go.mod go.sum ./
RUN go mod download
COPY . .
RUN go build -v -o ./server ./cmd/server/

FROM ubuntu
WORKDIR /
COPY ./assets ./assets
COPY --from=builder /app/server ./server
COPY --from=tailwind-builder /styles.css ./assets/styles.css
CMD ["./server"]
```

## Source Code

- [Source Code](#) - see the final source code for this lesson.
- [Diff](#) - see the changes made in this lesson.

## 23.10 Caddy Server via Docker

<https://caddyserver.com/docs/>

code Caddyfile

```
TODO: Replace this with a domain before deploying.
:80

reverse_proxy server:3000
```

code docker-compose.production.yml

```
caddy:
 image: caddy
 restart: always
 ports:
 - 80:80
 - 443:443
 # - "443:443/udp"
 volumes:
 - ./Caddyfile:/etc/caddy/Caddyfile
```

Remove the ports from the server service:

```
server:
 ...
 # Delete this bit
 # ports:
 # - 3000:3000
```

Verify it works.

```
docker compose -f docker-compose.yml -f docker-compose.production.yml up --build
```

Visit :80 to see the app. Remove May need to set `PSQL_HOST=db` in `.env` for testing.

Be sure to set the `PSQL_HOST` back to `localhost` after testing, as we will continue using modd for local development.

## Source Code

- [Source Code](#) - see the final source code for this lesson.
- [Diff](#) - see the changes made in this lesson.

## 23.11 Production Setup Exercises

### 23.11.1 Ex1 - Experiment with a new app port

Imagine we wanted our Go application to run on the port `3001` instead of `3000`. What changes would need to be made to make this work? What would we need to do in our docker files? In our Caddyfile?

Attempt to make the changes and update the port to 3001.

# Chapter 24

## Deploying

### 24.1 Creating a Digital Ocean Droplet

In this section we are going to discuss setting up a server on Digital Ocean (called a droplet on DO). We could use almost any hosting provider - Amazon AWS, Google's GCP, Microsoft Azure, etc. We could even set this up using a computer we have in our garage or home. The only real requirement is that the server has a static IP address we can use to connect to the server, and that it is a fresh install of Ubuntu 22.04 LTS (Long Term Support) that we can install programs on.

Even though we can use other hosting, I strongly recommend using Digital Ocean when learning so things are identical. Various hosts can have subtle differences that lead to issues that are hard to debug.

*Other versions of Ubuntu may work, but they have not been tested.*

#### Aside 24.1. Free Digital Ocean Credits

If you create a new account for Digital Ocean and use a referral link, you can get

some free credits to start out with. My own referral link is ([do.co/webdevgo](https://do.co/webdevgo)).

As of this writing, the referral is \$200 in free credits for 60 days, after that point the credits expire. Chances are you won't use anywhere near \$200 worth of credits, so it is basically a two month free trial. Just be sure to delete any droplets or services you don't want to pay for before the credits expire.

Full disclosure - if you end up paying Digital Ocean \$25 or more in hosting fees I will receive a \$25 credit myself. This ends up helping me pay for my course hosting, and makes it easier to offer free courses like Gophercises. You obviously don't need to use my referral link, but if you do I appreciate it.

Once we are signed into our Digital Ocean account, we are going to want to navigate to the new droplet page. From here we want to select the following options:

Setting	Selection
Region	New York
Datacenter	Datacenter 1
Image	Ubuntu - 22.04 LTS
CPU Options	Basic - Regular - \$6/mo option with 1GB of RAM
Authentication Method	SSH Key

You can pick whatever options you want for mostly everything else.

### Aside 24.2. Which Region Should I Use?

Typically the region and datacenter would be picked based on what is closer to your users, and which datacenter has the features you need available. For the course, I recommend picking New York and Datacenter 1 as these are known to have every feature we will be using.

### Aside 24.3. Why 1GB of RAM?

We are going to deploy and have our server build the Go code using Docker. The upside to this approach is that it is pretty easy to manage once setup and it is fairly simple to setup. One of the downsides is that the build process, especially with embedding, can be memory intensive, and we will want 1GB of memory for the build.

### Aside 24.4. Setting up an SSH Key

Follow the instructions from Digital Ocean for setting this up. If you already have a key, feel free to skim the directions for the part on printing out the `id_rsa.pub` file contents, then use that as your key in Digital Ocean.

Setup an Ubuntu. I am going to opt for 22.04 LTS (Long Term Support) since it will be supported for quite a while.

I will be using an SSH key to connect. I suggest doing the same thing - you may need to google how to set this up. This is likely something you had to setup to make ssh work with github, so same public key can be used. Alternative you can use a password.

Going to use the name `demo.lenslocked.com`.

Once booted, ensure you can SSH into the server.

```
ssh root@<your_server_ip>
Type "Yes" to agree to the fingerprint
```

Or if you created a new SSH key with a different name.

```
ssh -i ~/.ssh/your_id_rsa root@<your_server_ip>
Type "Yes" to agree to the fingerprint
```

We are now connected to the server!

*If using a new SSH key with a different name, we will see how to setup our git config to automatically use this key in the future.*

**ctrl+d** to exit.

## 24.2 Setting up DNS

If you want to use your own domain, add an **A** record pointing to your server's IP address.

Add a subdomain (eg www) if you want one, or use no subdomain if you prefer to not have one, or add two records, one with the **www** subdomain, and another with an empty subdomain so just the URL works.

For the TTL, start with something like 60 seconds when setting things up so we have more flexibility to change it without waiting for minutes. Then once things are working we can use a longer TTL.

Once this all kicks in we should be able to SSH via the A domain. For instance, if we used the domain **lenslocked.com** with the **casts** subdomain, we could do the following:

```
ssh root@casts.lenslocked.com
Type "Yes" to agree to the fingerprint
```

We are now connected to the server using a domain!

**ctrl+d** to exit.

Optionally, we can also tell SSH to use a specific username and SSH key. On Mac & Linux this should be:

```
code ~/.ssh/config
```

Then add:

```
Host spike.lenslocked.com
HostName spike.lenslocked.com
User root
IdentityFile ~/.ssh/lenslocked_id_rsa
```

IdentityFile is your SSH key you made. User is the username you are using with ssh (root is the default for us)

In windows this may be different, so you may need to do some research on your own.

## 24.3 Installing Git on the Server

SSH to server.

```
apt-get update
apt-get install git
```

Hit enter when asked about which services to restart to select “packagekit.service”.

```
git version
You should see something like:
git version 2.34.1
```

Ensure we are using **main** as the default branch, since this is what git uses by default in most installs now. We likely don’t really need to do this and could just specify our branch later on, but it helps avoid some warning messages.

```
We are going to use the default branch of main, as this is what newer
installs of git tend to default to. If you want to use another branch
you can, but be sure to update it everywhere we reference "main".
I'm not even sure we need this, but without it you will see weird warnings.
git config --global init.defaultBranch main
```

## 24.4 Setting Up a Bare Git Repo

Prepare the repo.

```
mkdir -p repos/lenslocked.git
cd repos/lenslocked.git
git init --bare
```

This gives us an empty repo to work with. We used bare because it means our repo on the server won’t have an active branch and we can push our code to it at anytime.

We will use a hook to deploy our code once it is pushed. This is a way of having our server do something after an event. In our case the hook will be on the event of receiving a new branch push. Let's set the boilerplate for that up now.

```
cd hooks
We want to be in the ~/repos/lenslocked.git/hooks directory, and we are
currently in the ~/repos/lenslocked.git dir, so we move into the hooks dir
nano post-receive
```

Add the following

```
#!/bin/bash
while read oldrev newrev ref
do
if [[$ref =~ .*/main$]];
then
echo "Deploying main branch to production..."
else
echo "Ref $ref successfully received, but only the main branch will be deployed."
fi
done
```

This is a bash script that will check if the branch pushed is `main`. If it is, we can run one set of code, otherwise we can notify the user that we won't deploy the branch they pushed.

We will expand this, but for now we just want to make sure it is working.

*If you are using another branch other than `main` (like `master`) update this to reflect that.*

To exit press `ctrl+x`, then hit `y` to confirm saving, and then `enter` to use the provided filename of `post-receive`.

Our bash script needs to be executable, so we can add that with `chmod`.

```
chmod +x post-receive
```

Exit the prod server with **ctrl+d**. We will work on setting up a local git repo and pushing to test this all next.

## 24.5 Setting Up a Local Git Repo

*If in the server, exit it (**ctrl+d**) or open a new terminal tab so you are working on your local machine.*

If we don't have a git repo on our local machine for our code, we need to create one and add all of our code to it.

*If you already have a git repo, you can skip the creation process, but follow along for the steps to add a remote server.*

```
git init
git checkout -b main
```

Now we have a way to track code changes, let's commit our code.

```
Add all changed files to this commit
git add -A
Commit the changes with a message of "Getting ready to deploy"
git commit -m 'Getting ready to deploy'
```

*If you have a git repo, this is where you will continue following along.*

Now we want to add our server as a remote.

```
If you setup a domain name, you can also use that here instead of the IP
git remote add prod root@<your_server_ip>:repos/lenslocked.git
```

And we can push the code.

```
push the main branch to prod
git push prod main
```

After doing this we should see output similar to that below. Some details will vary with your server IP or domain, but it keep an eye for the “Deploying main branch to production” message.

```
Enumerating objects: 981, done.
Counting objects: 100% (981/981), done.
Delta compression using up to 12 threads
Compressing objects: 100% (656/656), done.
Writing objects: 100% (981/981), 469.94 KiB | 42.72 MiB/s, done.
Total 981 (delta 543), reused 543 (delta 271), pack-reused 0
remote: Resolving deltas: 100% (543/543), done.
remote: Deploying main branch to production...
To casts.lenslocked.com:repos/lenslocked.git
```

Now we have a way to push our code changes to our server. Our next few lessons will focus on setting up our server to build the code and deploy our app.

## 24.6 Checking Out Our Code on the Server

ssh to server

```
cd ~
mkdir -p apps/lenslocked.com
cd apps/lenslocked.com
git --work-tree=$HOME/apps/lenslocked.com \
 --git-dir=$HOME/repos/lenslocked.git \
 checkout -f main
ls
Should see our Go source code
```

Our app code is here, and later we will use it to build our app as part of our deploy process.

## 24.7 Email Sending Server Setup

We are going to continue using the Mailtrap test emails for now, as I have no intentions of actually running this web app. But, if you wanted to setup a mail server for a real app you would...

If using mailtrap...

Head to the [Sending Domains](#) page and add your domain.

This will require setting up some DNS records.

Again, I recommend using short TTL while setting up, then going longer once everything is working.

Verify the domain.

Go through the survey to verify your domain and the app you are building.

*If using another service, follow the steps for that service.*

## 24.8 Production .env File

While SSHed into the server, we are going to setup a .env file for production

```
cd ~/apps/lenslocked.com
cp .env.template .env
nano .env
```

Fill it in with your information. - Setup all your SMTP info - Set the **PSQL\_HOST** to **db** - Set a PSQL username and password. This will be used later when setting up the DB, so we can define a new one now. - Use a new **CSRF\_KEY** (randomly generate a 32 char string for this) - If your **SERVER\_ADDRESS** has **localhost** in it, remove it. Eg **localhost:3000** becomes **:3000**

**ctrl+x** to exit. Hit Y to save

## 24.9 Install Docker in Prod

Docs: <https://docs.docker.com/engine/install/ubuntu/>

Remove old versions that may be present.

```
cd ~
for pkg in docker.io docker-doc docker-compose podman-docker containerd runc; do sudo apt-get r
```

Via apt:

Setup the apt repository to install from.

```
sudo apt-get update
sudo apt-get install ca-certificates curl gnupg
```

```
sudo install -m 0755 -d /etc/apt/keyrings
curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo gpg --dearmor -o /etc/apt/keyrings/docker.gpg
sudo chmod a+r /etc/apt/keyrings/docker.gpg

echo \
 "deb [arch=$(dpkg --print-architecture) signed-by=/etc/apt/keyrings/docker.gpg] https://download.docker.com/linux/ubuntu $(. /etc/os-release && echo "$VERSION_CODENAME") stable" | \
sudo tee /etc/apt/sources.list.d/docker.list > /dev/null
```

Install now

```
sudo apt-get update
sudo apt-get install docker-ce docker-ce-cli containerd.io docker-buildx-plugin docker-compose-*
Hit Y and then enter if asked to confirm
Hit enter when asked about restarting any services
```

Verify it worked.

```
docker -v
Docker version 24.0.5, build ced0996
docker compose version
Docker Compose version v2.20.2
```

Docker is now installed and runs via systemd. This is a service taht will restart docker if it crashes, or if our server restarts. That means docker will always be running on our machine, so we if we use it to deploy our app we can use it to restart our app for us if things crash.

If you received a warning about needing to restart, you can do that with the following:

```
shutdown -r now
```

## 24.10 Production Caddyfile

Leave the server, and make these changes locally. We will commit them and push them to the server later.

This is intended for production, so we are going to set it up for that environment.

```
code Caddyfile
```

If you do NOT have a domain, you can leave this as-is.

```
Use your own domain here.
casts.lenslocked.com

reverse_proxy server:3000
```

**server** here comes from our docker setup, where we named our Go service **server**.

### Source Code

- [Source Code](#) - see the final source code for this lesson.
- [Diff](#) - see the changes made in this lesson.

## 24.11 Production Data Directories

**Aside 24.5. Caddy Bug Fix**

The videos for this lesson have a bug. I forgot to set up the directory for the Caddy data, and without this Caddy has to reissue a certificate every time we deploy which can eventually lead to issues since there is a limit to how many times Caddy will issue a new certificate in any given time period.

The fix for this is to create the caddy data directory, which is shown in the written version of this lesson, and to update the production docker-compose file, which is also shown in the written version of the lesson.

At the moment our images are stored in a directory relative to our application. This was handy when testing the production setup locally, but probably not ideal when in our production environment on the off-chance we want to delete everything in that directory and start with a clean slate before building our code.

We also have our database data stored inside of the **db** Docker container. This isn't the worst thing in the world, but it could be useful to move that data to a directory on our server that we can more easily backup. It is also nice to know that destroying our Postgres docker container won't lose all of our DB data.

In this lesson we will look at how to move all of our persistent data like this to their own directories.

We will start by SSHing into our server.

```
ssh root@casts.lenslocked.com
```

Next we will make the directories we need.

```
mkdir -p ~/data/lenslocked.com/sql
mkdir -p ~/data/lenslocked.com/images
Warning: This line isn't in the videos, but that is a mistake
mkdir -p ~/data/lenslocked.com/caddy
```

Now we can leave the server with **ctrl+d**.

Now in our code directory we are going to update our production docker compose file to reflect these changes.

```
code docker-compose.production.yml
```

We will be updating the volume we attach to our **server** service to use the images data directory we created.

```
version: "3.9"

services:
 server:
 build:
 context: ./
 dockerfile: Dockerfile
 restart: always
 # Change this section
 volumes:
 - ~/data/lenslocked.com/images:/app/images
 depends_on:
 - db

...
```

Update Caddy to use the caddy data directory. (*Note: This is not in the video but should be.*)

```
version: "3.9"

services:
 # ... existing stuff

 caddy:
 image: caddy
 restart: always
 ports:
 - 80:80
 - 443:443
 volumes:
```

```
- ./Caddyfile:/etc/caddy/Caddyfile
Add this line. This is not shown in the video, but that is a mistake
- ~/data/lenslocked.com/caddy:/data

...
```

Next up we will be referencing the **PGDATA** section of the **postgres** image. Here we see that we need two things to mount a volume for our postgres data:

1. To set the **PGDATA** variable specifying where we want to store our database data.
2. To mount a volume at this same location.

We will do this by editing our production docker-compose file.

```
version: "3.9"

services:
 # ... existing stuff

 # Add this
 db:
 environment:
 PGDATA: /var/lib/postgresql/data/pgdata
 POSTGRES_USER: ${PSQL_USER}
 POSTGRES_PASSWORD: ${PSQL_PASSWORD}
 POSTGRES_DB: ${PSQL_DATABASE}
 volumes:
 - ~/data/lenslocked.com/pgsql:/var/lib/postgresql/data/pgdata
```

Now save the file and commit it to your git repo.

## Source Code

- [Source Code](#) - see the final source code for this lesson.
- [Diff](#) - see the changes made in this lesson.

## 24.12 Running Our App in Prod

We need to make sure we have the latest code in production. We do this by committing locally, then pushing it to production.

```
git add -A
git commit -m "Updated the production docker setup to use a data directory for persistent data,
git push prod main
```

SSH into the server.

```
ssh root@casts.lenslocked.com
```

Then navigate to our app repo and update the source code.

```
cd ~/apps/lenslocked.com
git --work-tree=$HOME/apps/lenslocked.com \
--git-dir=$HOME/repos/lenslocked.git \
checkout -f main
```

Now that our code is updated we can use docker to start up our server.

```
docker compose -f docker-compose.yml -f docker-compose.production.yml up
```

This may take a few minutes.

Once the server is up and running you will see Docker logs like you did when running locally.

At this time visit your server domain (or IP address if you didn't use a domain) in your browser.

If you have a domain, Caddy should be handling all of the HTTPS bits for us!

Stop Caddy (**ctrl + c**).

When we are done, Docker will build and run our app automatically when we push new code, but for now we need to finish setting a few things up.

## 24.13 Post-receive Deploy Updates

SSH into your server

```
ssh root@lenslocked.com
```

Open the post-receive file.

```
nano ~/repos/lenslocked.git/hooks/post-receive
```

Replace its contents with the following.

```
#!/bin/bash
while read oldrev newrev ref
do
if [[$ref =~ .*/spike.*$]];
then
echo "Deploying main branch to production..."
cd ~/apps/lenslocked.com
git --work-tree=$HOME/apps/lenslocked.com \
--git-dir=$HOME/repos/lenslocked.git \
checkout -f spike/s24
echo "Restarting services..."
docker compose \
-f docker-compose.yml \
-f docker-compose.production.yml \
up --build --force-recreate -d
echo "Services restarted."
docker compose \
```

```
-f docker-compose.yml \
-f docker-compose.production.yml \
ls
docker compose \
-f docker-compose.yml \
-f docker-compose.production.yml \
ps
else
echo "Ref $ref successfully received, but only the main branch will be deployed."
fi
done
```

#### Aside 24.6. Storing post-receive in git

Our **post-receive** code could be stored in git, making it easier to manage and edit, but keep in mind that we do NOT automatically update this on the server, so the changes need to be copied to production as well. In that case, it might make sense to add a comment to the file about manually updating it, or adding a way to automate the process.

## 24.14 Deploy via Git

If we try to push our code to prod, we will see the following:

```
git push prod casts
Everything up-to-date
```

We need to make a change so that we can push it to production and have our deploy script run.

This could be any file, but let's update our **.env.template** file to mention the **db** setting if running the Go server via docker.

```
code .env.template
```

```
SMTP_HOST=sandbox.smtp.mailtrap.io
SMTP_PORT=587
SMTP_USERNAME=<your username>
SMTP_PASSWORD=<your password>

PSQL_HOST should be set to "db" if running the Go application as a
docker service.
PSQL_HOST=localhost
PSQL_PORT=5432
PSQL_USER=baloo
PSQL_PASSWORD=junglebook
PSQL_DATABASE=lenslocked
PSQL_SSLMODE=disable

CSRF_KEY=<32 byte string>
CSRF_SECURE=false

SERVER_ADDRESS=:3000
```

Save the file and commit it to git.

```
git add -A
git commit -m 'Added some extra docs to the .env.template explaining changes needed if running'
```

Push the changes to production and watch the deploy.

```
git push prod main
```

We are going to verify that our server restarts everything after we power it off. We will do this by telling our server to restart, giving it a few minutes, then seeing if we can access our app.

```
ssh root@lenslocked.com
```

Now run the following to tell the server to restart immediately.

```
shutdown -r now
```

Now wait a few minutes, and attempt to visit your app via its URL.

### Aside 24.7. Faster Builds with More Powerful Servers

The build currently takes a bit to complete. This is due to our server needing to both build docker containers and our Go application. One way to speed this up is to pay for a more powerful server. For instance, a 2 CPU droplet with 2GB of memory will build and deploy our application noticeably faster than our current 1 CPU with 1GB of memory setup.

### Source Code

- [Source Code](#) - see the final source code for this lesson.
- [Diff](#) - see the changes made in this lesson.

## 24.15 Logging Services

We can currently see logs by SSHing into our server and running:

```
cd ~/apps/lenslocked.com/
docker compose \
-f docker-compose.yml \
-f docker-compose.production.yml \
logs
```

We can also use services like [highlight.io](#) to send our logs to another service. Head to their website and do the following:

1. Sign up for a free account.
2. Go to the Logging tab on the left.
3. Select Infrastructure/Other.
4. Select the Docker option.

We will be using code from the Docker Compose setup to update our production file.

```
code docker-compose.production.yml
```

If you opt to use the code below, be sure to get your project ID from Highlight.io and fill it in.

```
version: "3.9"

x-logging:
 &highlight-logging
 driver: fluentd
 options:
 # These lines may vary - use what is in your setup.
 fluentd-address: "otel.highlight.io:24224"
 fluentd-async: "true"
 fluentd-sub-second-precision: "true"
 # This line will need your own project ID
```

```
tag: "highlight.project_id=<your project id>"

services:
 # ... unchanged
```

Now update your services to use this logging.

```
services:
 server:
 build:
 context: ./
 dockerfile: Dockerfile
 restart: always
 volumes:
 - ~/data/lenslocked.com/images:/app/images
 depends_on:
 - db
 logging: *highlight-logging

 caddy:
 # ...
 logging: *highlight-logging

 db:
 # ...
 logging: *highlight-logging
```

Push your changes and wait for the server to restart.

```
git push prod main
```

After a few moments there should be logs in the highlight.io account. In my experience, it can take up to 15 minutes for logs to make their way to Highlight.io, so some patience may be required.

Once logs are in Highlight.io, they can be filtered and searched, but are only persisted for a few days in the free tier.

## Source Code

- [Source Code](#) - see the final source code for this lesson.
- [Diff](#) - see the changes made in this lesson.

## **24.16 Deploy Exercises**

### **24.16.1 Ex1 - Explore services offered by Digital Ocean**

This is completely optional, but consider exploring other services provided by Digital Ocean and consider trying some of them. For instance, they offer managed database instances. How might this improve our application? What would the trade-offs be? What would need changed in our code to connect to a managed database service?

# Chapter 25

## Bug Fixes and Updates

### 25.1 Fixing a Caddy Data Bug

In the videos a mistake was made when setting up Caddy for production. The short version is that we are not persisting any of the certificates that Caddy gets for our server to run with **https**, and this will eventually lead to issues as Caddy can only acquire so many certificates in a set period of time. If you are deploying once a week the code would likely be fine as-is, but if you deploy many times in a day then it will become problematic.

In this lesson we are going to look at how to fix the bug, as it is a pretty quick and easy fix that was an oversight on my part while creating the course.

First, we SSH to the production server so that we can create a data directory for our Caddy data. This is going to match the pattern we made for the other data we store.

*Note: If you followed a recent version of the written lessons then you may have already addressed this bug and you can skip this lesson.*

```
Warning: This line isn't in the videos, but that is a mistake
mkdir -p ~/data/lenslocked.com/caddy
```

Exit the production server with **ctrl+d**.

Now in the code directory we will update the production docker-compose file to use the data directoty we created.

```
code docker-compose.production.yml
```

Update it with the following:

```
version: "3.9"

services:
 # ... existing stuff

 caddy:
 image: caddy
 restart: always
 ports:
 - 80:80
 - 443:443
 volumes:
 - ./Caddyfile:/etc/caddy/Caddyfile
 # Add this line.
 - ~/data/lenslocked.com/caddy:/data
 # ...
```

And that's it! Now all of our docker data is persisted on our production server and won't be destroyed any time we redeploy our application.

## Source Code

- [Source Code](#) - see the final source code for this lesson.
- [Diff](#) - see the changes made in this lesson.

## 25.2 Go 1.22's Updated ServeMux

In Go 1.22 the `ServeMux` in the standard library was updated to support a number of additional patterns. Most notably, it now provides the option to define routes with specific HTTP methods and it also supports URL path parameters. In this lesson we are going to explore how we might use the standard library with these changes rather than `chi`, and I'll explain why I still prefer `chi` for this application.

First, let's discuss what changed. Prior to Go 1.22, the `ServeMux` type in the `net/http` package could only route web requests based on the path. It did not support any differentiation between HTTP methods. Go 1.22 made it possible to define the same path with varying HTTP methods and have each handled with a different HTTP handler.

```
// This was NOT possible in Go 1.22
mux := http.NewServeMux()
mux.HandleFunc("GET /signup", showSignupForm)
mux.HandleFunc("POST /signup", processSignupForm)
```

Without the support for HTTP methods, the `ServeMux` was challenging to use, as developers would be forced to check the HTTP method on their own and route the application accordingly. It wasn't impossible, but it wasn't always clean code. For instance, below is one way to do this prior to Go 1.22.

```
mux := http.NewServeMux()
mux.HandleFunc("/signup", func(w http.ResponseWriter, r *http.Request) {
 switch r.Method {
 case http.MethodGet:
 showSignupForm(w, r)
 case http.MethodPost:
 processSignupForm(w, r)
 default:
 errorPage(w, r)
 }
})
```

While this code works, it isn't as easy to read since we are no longer just looking at routes being declared and need to start reading switch statements. It also takes more time to code, which isn't necessarily ideal if that extra code isn't providing clarity or some other benefit.

Another change in Go 1.22 was the addition of path parameters. For instance, we can now define paths like the ones shown below with the `ServeMux` type.

```
mux := http.NewServeMux()
mux.HandleFunc("GET /galleries/{id}", showGallery)
```

In our HTTP handler we can get the gallery ID by calling the `PathValue` method on the `*Request` type.

```
func showGallery(w http.ResponseWriter, r *http.Request) {
 galleryID := r.PathValue("id")
 // ...
}
```

These two changes alone get the `ServeMux` type much closer to chi and make it a much more viable choice, but there are still a few edge cases where chi is significantly easier to use than the `ServeMux` type. For instance, the following routes are much more challenging to replicate with the `ServeMux` type:

```
r.Route("/galleries", func(r chi.Router) {
 r.Get("/{id}", galleriesC.Show)
 r.Get("/{id}/images/{filename}", galleriesC.Image)
 r.Group(func(r chi.Router) {
 r.Use(umw.RequireUser)
 r.Get("/", galleriesC.Index)
 r.Get("/new", galleriesC.New)
 r.Post("/", galleriesC.Create)
 r.Get("/{id}/edit", galleriesC.Edit)
 r.Post("/{id}", galleriesC.Update)
 r.Post("/{id}/delete", galleriesC.Delete)
 r.Post("/{id}/images", galleriesC.UploadImage)
 r.Post("/{id}/images/url", galleriesC.ImageViaURL)
 r.Post("/{id}/images/{filename}/delete", galleriesC.DeleteImage)
 })
})
```

As a result, I still believe chi is the best fit for our application. It is well tested, has always worked well for students learning web development, and it still teaches the same core concepts necessary to understand how routing works in a web app. That said, I would highly recommend anyone taking the course to spend some time trying to replace chi with ServeMux. Not only will it be a good practice exercise, but it will ensure that you are comfortable with the ServeMux type if you encounter it in another project.