



Learn to Create Real World Web Applications using Go

by: Jonathan Calhoun

Web Development with Go

Learn to Create Real World Web Applications using Go

Jonathan Calhoun

Contents

About the author	v
Copyright and license	vii
The Book Cover	ix
1 Introduction	1
1.1 Who is this book for?	1
1.2 How to use this book	2
1.3 Reference materials	2
1.4 What if I am already an expert developer?	3
1.5 What are we building?	4
1.6 Conventions used in this book	4
1.6.1 Command-line commands are prefixed with a \$	6
1.6.2 The subl or ‘atom’ command means “open with your text editor”	6

1.6.3	Many code listings will be shortened for clarity	6
1.6.4	All code samples are limited to 60 columns when possible	7
1.7	Accessing the code	8
1.7.1	Why git?	8
1.8	Disclaimer: Not everything is one size fits all	9
1.9	Commenting your exported types	9
2	A basic web application	11
2.1	Building the server	11
2.2	Demystifying our app	14
2.3	Using Go Modules	22
3	Adding new pages	25
3.1	Routing with if/else statements	26
3.2	Popular routers	29
3.2.1	net/http.ServeMux	30
3.2.2	github.com/julienschmidt/httprouter	30
3.2.3	github.com/gorilla/mux.Router	31
3.2.4	What about other router>	31
3.3	Using the gorilla/mux router	31
3.4	Exercises	34
3.4.1	Ex1 - Add an FAQ page	35

<i>CONTENTS</i>	v
3.4.2 Ex2 - Custom 404 page	35
3.4.3 Ex3 - [HARD] Try out another router	36
4 A brief introduction to templates	37
4.1 What are templates?	38
4.2 Why do we use templates?	39
4.3 Templates in Go	39
4.4 Creating a template	41
4.5 Contextual encoding	45
4.6 Exercises	48
4.6.1 Ex1 - Add a new template variable	49
4.6.2 Ex2 - Experiment with different data types	49
4.6.3 Ex3 - [HARD] Learn how to use nested data	50
4.6.4 Ex4 - [HARD] Create an if/else statement in your template	50
5 Understanding MVC	51
5.1 Model-View-Controller (MVC)	51
5.2 Walking through a web request	53
5.3 Exercises	57
5.3.1 Ex1 - What does MVC stand for?	57
5.3.2 Ex2 - What is each layer of MVC responsible for?	57

5.3.3	Ex3 - What are some benefits to using MVC?	58
6	Creating our first views	59
6.1	The home template	60
6.2	The contact template	63
6.3	Creating a reusable Bootstrap layout	66
6.3.1	Named templates	67
6.3.2	Creating a view type	71
6.3.3	Creating the Bootstrap layout	76
6.4	Adding a navigation bar	82
6.5	Cleaning up our code	87
6.5.1	What is globbing?	88
6.5.2	Using <code>filepath.Glob</code>	89
6.5.3	Simplifying view rendering	94
6.5.4	Moving our footer to the layout	96
6.6	Exercises	97
6.6.1	Ex1 - Create an FAQ page with the Bootstrap layout	98
6.6.2	Ex2 - Update the navbar to link to the FAQ page	98
6.6.3	Ex3 - Create a new layout	99
7	Creating a sign up page	101
7.1	Add a sign up page with a form	101

7.1.1	Creating a Bootstrap sign up form	102
7.1.2	Wrapping our form in a panel	109
7.1.3	Adding the sign up link to our navbar	114
7.2	An intro to REST	117
7.2.1	How REST affects our code	118
7.3	Creating our first controller	119
7.3.1	Create the users controller	122
7.3.2	Moving the sign up page code	123
7.3.3	Connecting our router and the users controller together	127
7.4	Processing the sign up form	130
7.4.1	Stubbing the create user action	131
7.4.2	HTTP request methods and gorilla/mux	133
7.4.3	Parsing a POST form	137
7.4.4	Parsing forms with gorilla/schema	140
7.4.5	Keeping our parsing code DRY	144
7.5	Cleaning up and creating a static controller	148
7.5.1	Creating the static controller	149
7.5.2	Simplifying the creation of new views	154
7.6	Exercises	159
7.6.1	Ex1 - Add the FAQ page to the static controller	159
7.6.2	Ex2 - Create a new controller for galleries	160

7.6.3	Exercise cleanup	160
8	An introduction to databases	161
8.1	Our web app will use PostgreSQL	162
8.2	Setting up PostgreSQL	164
8.2.1	Install PostgreSQL	164
8.2.2	Learn how to connect to Postgres	165
8.2.3	Learn the basics of SQL	166
8.2.4	Gather information needed to connect to your Postgres install	167
8.3	Using Postgres with Go and raw SQL	169
8.3.1	Connecting to Postgres with the database/sql package .	169
8.3.2	Creating SQL tables to test with	175
8.3.3	Writing records with database/sql	176
8.3.4	Querying a single record with database/sql	178
8.3.5	Querying multiple records with database/sql	180
8.3.6	Writing a relational record	183
8.3.7	Querying related records	185
8.3.8	Delete the SQL tables we were testing with	186
8.4	Using GORM to interact with a database	186
8.4.1	Installing GORM and connecting to a database	187
8.4.2	Defining a GORM model	189

8.4.3	Creating and migrating tables with GORM	191
8.4.4	Logging with GORM	193
8.4.5	Creating a record with GORM	194
8.4.6	Querying a single record with GORM	197
8.4.7	Querying multiple records with GORM	202
8.4.8	Creating related models with GORM	202
8.4.9	Querying relational data with GORM	205
8.5	Exercises	207
8.5.1	Ex1 - What changes won't the AutoMigrate function provided by GORM handle for you?	208
8.5.2	Ex2 - What is <code>gorm.Model</code> used for?	208
8.5.3	Ex3 - Experiment using a few more GORM methods. .	208
8.5.4	Ex4 - Experiment with query chaining	209
8.5.5	Ex5 - Learn to execute raw SQL with GORM	209
9	Creating the user model	211
9.1	Defining a <code>User</code> type	211
9.2	Creating the <code>UserService</code> interface and querying for users .	215
9.3	Creating users	221
9.4	Querying by email and DRYing up our code	223
9.5	Updating and deleting users	227
9.6	AutoMigrating and returning errors from <code>DestructiveReset</code> . .	231

9.7	Connecting the user service and controller	233
9.7.1	Adding the name field the sign up form	233
9.7.2	Setting up a user service in our web application	236
9.7.3	Using the users service in our users controller	239
9.8	Exercises	242
9.8.1	Ex1 - Add an Age field to our user resource	242
9.8.2	Ex2 - Write a method to query the first user with a specific age	243
9.8.3	Ex3 - [HARD] Write a method to query many users by age range	243
10	Building an authentication system	245
10.1	Why not use another package or service?	246
10.2	Secure your server with SSL/TLS	247
10.3	Hash passwords properly	247
10.3.1	What is a hash function?	248
10.3.2	Store hashed passwords, not raw passwords	249
10.3.3	Salt and pepper passwords	250
10.4	Implementing password hashing for our users	254
10.4.1	Adding password fields to the user model	254
10.4.2	Hash passwords with bcrypt before saving	256
10.4.3	Retrieving passwords from the sign up form	260

10.4.4	Salting and peppering passwords	261
10.5	Authenticating returning users	264
10.5.1	Creating the login template	264
10.5.2	Creating the login action	266
10.5.3	Implementing the Authenticate method	268
10.5.4	Calling Authenticate from our login action	272
10.6	Exercises	274
10.6.1	Ex1 - What is a hash function?	274
10.6.2	Ex2 - What purpose does a salt and pepper serve?	274
10.6.3	Ex3 - Timing attacks	274
11	Remembering users	277
11.1	What are cookies	278
11.2	Creating our first cookie	279
11.3	Viewing cookies	283
11.3.1	Viewing cookies in Google Chrome	283
11.3.2	Viewing cookies with Go code	289
11.4	Securing our cookies from tampering	291
11.4.1	Digitally signing data	291
11.4.2	Obfuscating cookie data	294
11.5	Generating remember tokens	295

11.5.1 Why do we use 32 bytes?	301
11.6 Hashing remember tokens	303
11.6.1 How to use the hash package	304
11.6.2 Using the crypto/hmac package	306
11.6.3 Writing our own hash package	307
11.7 Hashing remember tokens in the user service	310
11.7.1 Adding remember token fields to our User type	311
11.7.2 Setting a remember token when a user is created	312
11.7.3 Adding an HMAC field to the UserService	314
11.7.4 Hashing remember tokens on create and update	316
11.7.5 Retrieving a user by remember token	317
11.7.6 Resetting our DB and testing	318
11.8 Remembering users	321
11.8.1 Storing remember tokens in cookies	322
11.8.2 Restricting page access	326
11.9 Securing our cookies from XSS	328
11.10 Securing our cookies from theft	330
11.11 Preventing CSRF attacks	332
11.12 Exercises	333
11.12.1 Ex1 - Experiment with redirection	333
11.12.2 Ex2 - Create cookies with different data	333

11.12.3 Ex3 - [HARD] Experiment with the hash package	333
12 Normalizing and validating data	335
12.1 Separating responsibilities	338
12.1.1 Rethinking the models package	339
12.1.2 Static types vs interfaces	344
12.2 The UserDB interface	353
12.3 The UserService interface	362
12.4 Writing and organizing validation code	366
12.4.1 The straightforward approach	367
12.4.2 A few more validation examples	370
12.4.3 Reusable validations	373
12.5 Writing validators and normalizers	379
12.5.1 Remember token normalizer	379
12.5.2 Ensuring remember tokens are set on create	382
12.5.3 Ensuring a valid ID on delete	384
12.6 Validating and normalizing email addresses	387
12.6.1 Converting emails to lowercase and trimming whitespace	388
12.6.2 Requiring email addresses	392
12.6.3 Verifying emails match a pattern	394
12.6.4 Verifying an email address isn't taken	402

12.7 Cleaning up our error naming	406
12.8 Validating and normalizing passwords	409
12.8.1 Verifying passwords have a minimum length	409
12.8.2 Requiring a password	411
12.9 Validating and normalizing remember tokens	414
13 Displaying errors to the end user	419
13.1 Rendering alerts in the UI	420
13.2 Rendering dynamic alerts	423
13.3 Only display alerts when we need them	427
13.4 A more permanent data type for views	431
13.5 Handling errors in the sign up form	435
13.6 White-listing error messages	441
13.7 Handling login errors	449
13.8 Recovering from view rendering errors	452
14 Creating the gallery resource	457
14.1 Defining the gallery model	458
14.2 Introducing the GalleryService	460
14.3 Constructing many services	461
14.4 Closing and migrating all models	466
14.5 Creating new galleries	470

14.5.1	Implementing the gallery service	471
14.5.2	The galleries controller	473
14.5.3	Processing the new gallery form	476
14.5.4	Validating galleries	480
14.6	Requiring users via middleware	482
14.6.1	Creating our first middleware	483
14.6.2	Storing request-scoped data with context	490
14.7	Displaying galleries	498
14.7.1	Creating the show gallery view	499
14.7.2	Parsing the gallery ID from the path	501
14.7.3	Looking up galleries by ID	505
14.7.4	Generating URLs with params	508
14.8	Editing galleries	514
14.8.1	The edit gallery action	514
14.8.2	Parsing the edit gallery form	519
14.8.3	Updating gallery models	522
14.9	Deleting galleries	525
14.10	Viewing all owned galleries	529
14.10.1	Querying galleries by user ID	529
14.10.2	Adding the Index handler	531
14.10.3	Iterating over slices in Go templates	534

14.11 Improving the user experience and cleaning up	539
14.11.1 Adding intuitive redirects	540
14.11.2 Navigation for signed in users	544
15 Adding images to galleries	553
15.1 Image upload form	554
15.2 Processing image uploads	558
15.3 Creating the image service	571
15.4 Looking up images by gallery ID	577
15.5 Serving static files in Go	582
15.6 Rendering images in columns	586
15.7 Letting users delete images	594
15.8 Known Bugs	601
16 Deploying to production	603
16.1 Error handling	604
16.2 Serving static assets	605
16.3 CSRF protection	608
16.4 Limiting middleware for static assets	617
16.5 Fixing bugs	619
16.5.1 URL encoding image paths	619
16.5.2 Redirecting after image uploads	622

16.6 Configuring our application	624
16.6.1 Finding variables that need provided	625
16.6.2 Using functional options	630
16.6.3 JSON configuration files	639
16.7 Setting up a server	646
16.7.1 Digital Ocean droplet	647
16.7.2 Installing PostgreSQL in production	649
16.7.3 Installing Go	654
16.7.4 Setting up Caddy	656
16.7.5 Creating a service for our app	662
16.7.6 Setting up a production config	665
16.7.7 Creating a deploy script	666
17 Filling in the gaps	671
17.1 Deleting cookies and logging out users	671
17.2 Redirecting with alerts	674
17.3 Emailing users	679
17.4 Persisting and prefilling form data	683
17.5 Resetting passwords	688
17.5.1 Creating the database model	689
17.5.2 Updating the services	694

17.5.3	Forgotten password forms	700
17.5.4	Controller actions and views	704
17.5.5	Emailing users and building URLs	709
17.6	Where do I go next?	713
18	Appendix	715
18.1	Using interfaces in Go	715
18.2	Refactoring with gorename	718
18.3	Handling Bootstrap issues	721
18.4	Private model errors	721

About the author

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

Copyright and license

Web Development with Go: Learn to Create Real World Web Applications using Go. Copyright © 2016 by Jon Calhoun.

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

The full license is listed below.

The MIT License

Copyright (c) 2016 Jon Calhoun

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR

PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

The Book Cover

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

Chapter 1

Introduction

Welcome to [Web Development with Go: Learn to Create Real World Web Applications using Go!](#)

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

1.1 Who is this book for?

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

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

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

1.2 How to use this book

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

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

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

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

1.3 Reference materials

While reading this book you are likely to come across some things you are unfamiliar with that you want to research further. To help aid you, I have created

and continue to maintain a beginners guide with resources for diving deeper into Go, HTML, CSS, SQL, and the command line.

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

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

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

1.4 What if I am already an expert developer?

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

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

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

1.5 What are we building?

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

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

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

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

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

1.6 Conventions used in this book

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

[H]

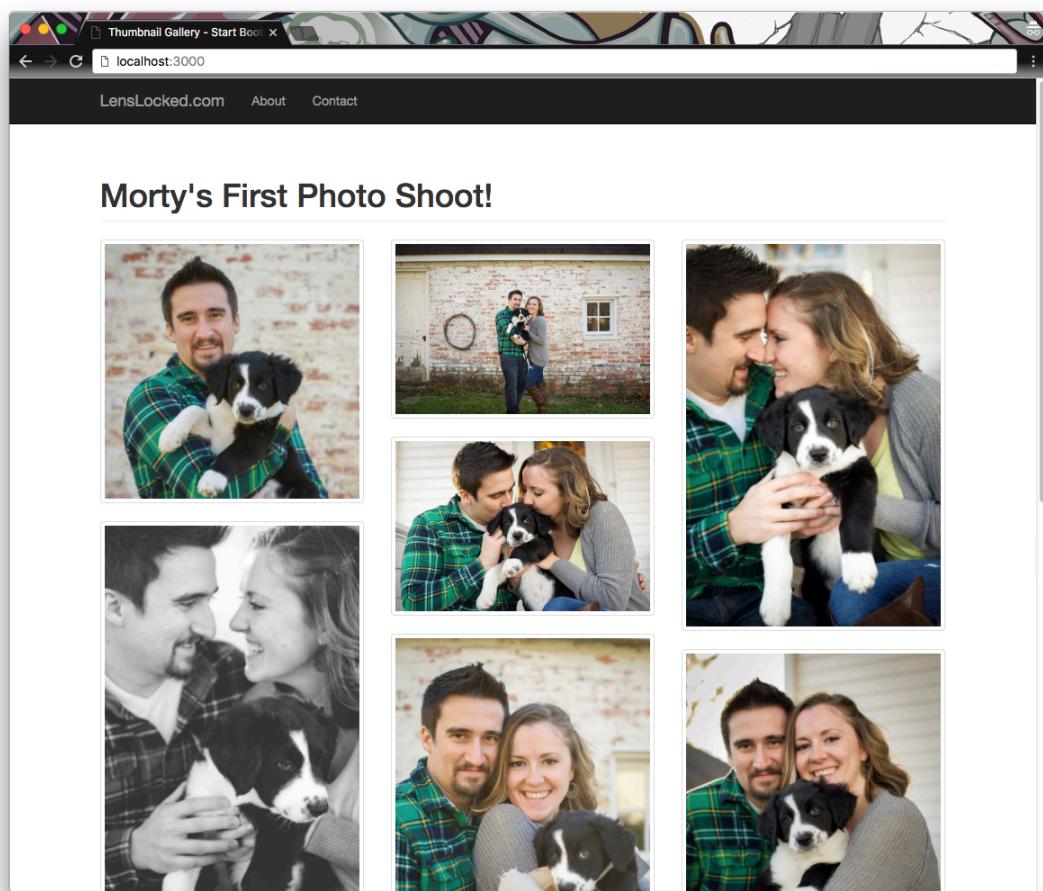


Figure 1.1: A picture of the web application we will build

1.6.1 Command-line commands are prefixed with a \$

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

Listing 1.1: Unix-style prompt example

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

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

1.6.2 The `subl` or ‘atom’ command means “open with your text editor”

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

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

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

1.6.3 Many code listings will be shortened for clarity

Throughout this book there will be many examples where we are working within a larger file but only need to make a slight change to a line or two. When this

happens, showing a code listing of the entire file would be both confusing and a waste of space.

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

Listing 1.2: A simple example of unchanged code

```
func main() {
    // ... everything before this remains unchanged
    var name string
    fmt.Println("What is your name?")
    fmt.Scanf("%s", &name)
}
```

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

1.6.4 All code samples are limited to 60 columns when possible

In the screencasts I often write code that is wider than 60 columns, but in the ebook formats this doesn't tend to work very well. As a result, I have attempted to limit all code samples to 60 characters or less per line.

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

1.7 Accessing the code

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

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

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

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

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

1.7.1 Why git?

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

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

Rather than introduce this confusion, I instead opted to use git branches with

links to each branch at the end of each section.

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

1.8 Disclaimer: Not everything is one size fits all

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

Over time, your personal applications will likely become more robust and complex, and you might find yourself questioning and changing the structure we use here. That is to be expected.

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

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

1.9 Commenting your exported types

If you use **golint**, or if you use an editor which automatically runs it on your code you may see warnings throughout the book. These typically read something like “exported function XXX should have comment or be unexported.” What this warning means is that you are exporting a function (making it avail-

able outside of the package), but you haven't provided proper comments explaining what the package does for other developers.

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

Chapter 2

A basic web application

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

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

2.1 Building the server

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

Listing 2.1: Creating the application directory

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

```
$ cd lenslocked.com
```

Now that we have a proper directory to work from we can create our first go file, `main.go`. Create the file and then open it up in your favorite editor. I will be using Atom¹, so I will represent creating files with a command like the one in Figure ??.

Listing 2.2: Creating `main.go`

```
$ atom main.go
```

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

Listing 2.3: Creating a simple web application

```
package main

import (
    "fmt"
    "net/http"
)

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

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

Finally, we run the app.

¹<https://atom.io/>

[H]

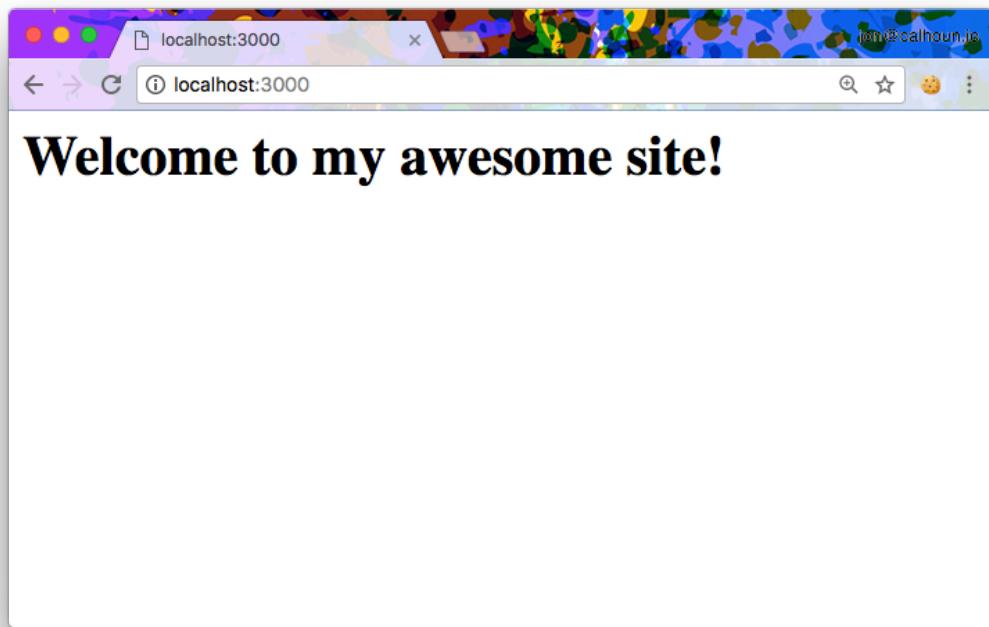


Figure 2.1: Our first web application!

Listing 2.4: Running `main.go`.

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

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

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

Box 2.1. Troubleshooting installation issues

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

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

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

Source code

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

Completed - book.usegolang.com/2.1

2.2 Demystifying our app

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

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

Starting with the first line...

```
package main
```

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

Next up are the imports:

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

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

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

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

Box 2.2. What are standard libraries?

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

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

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

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

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

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

Box 2.3. A brief introduction to web requests

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

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

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

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

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

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

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

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

Box 2.4. What are pointers?

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

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

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

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

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

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

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

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

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

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

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

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

Box 2.6. What are interfaces?

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

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

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

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

Finally we get to the main function.

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

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

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

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

Box 2.7. Port 3000 and localhost.

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

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

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

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

2.3 Using Go Modules

Go 1.11 introduced modules, which are a way of tracking third party library dependencies and making sure anyone else building your Go app uses the same versions of those libraries.

You can still complete this entire course without using Go modules, but if you want to use them you can do so by first navigating to your code directory, then running the following:

```
$ go mod init lenslocked.com
```

This will create a file named **go.mod** which will keep track of any third party libraries you install as you build your app, along with their version. Go will update this file and make any changes for you as we go, but you will probably occasionally want to run the following to tidy things up, remove unused libraries, and just keep your mod file tidy:

```
$ go mod tidy
```

As of this writing, my **go.mod** file for the completed course looks like this:

```
module lenslocked.com

go 1.15

require (
    github.com/davecgh/go-spew v1.1.1 // indirect
    github.com/facebookgo/ensure v0.0.0-20200202191622-63f1cf65ac4c // indirect
    github.com/facebookgo/stack v0.0.0-20160209184415-751773369052 // indirect
    github.com/facebookgo/subset v0.0.0-20200203212716-c811ad88dec4 // indirect
    github.com/gorilla/csrf v1.7.0
    github.com/gorilla/mux v1.8.0
    github.com/gorilla/schema v1.2.0
    github.com/jinzhu/gorm v1.9.16
    github.com/onsi/ginkgo v1.14.2 // indirect
```

```
github.com/onsi/gomega v1.10.4 // indirect
golang.org/x/crypto v0.0.0-20201221181555-eec23a3978ad
gopkg.in/mailgun/mailgun-go.v1 v1.1.1
)
```

Again, you do not need to copy this over to your `go.mod` file, and the `go` CLI should handle all of this for you as you go. This is just here for your reference.

Chapter 3

Adding new pages

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

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

Box 3.1. HTTP Status Codes.

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

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

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

3.1 Routing with if/else statements

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

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

Listing 3.1: Routing via the URL path.

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

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

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

Box 3.2. Dynamic reloading

If you are interested in learning about dynamic reloading, check out the Go library [fresh - github.com/pilu/fresh](#)

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

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

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

Listing 3.2: Creating a 404 page

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

[H]

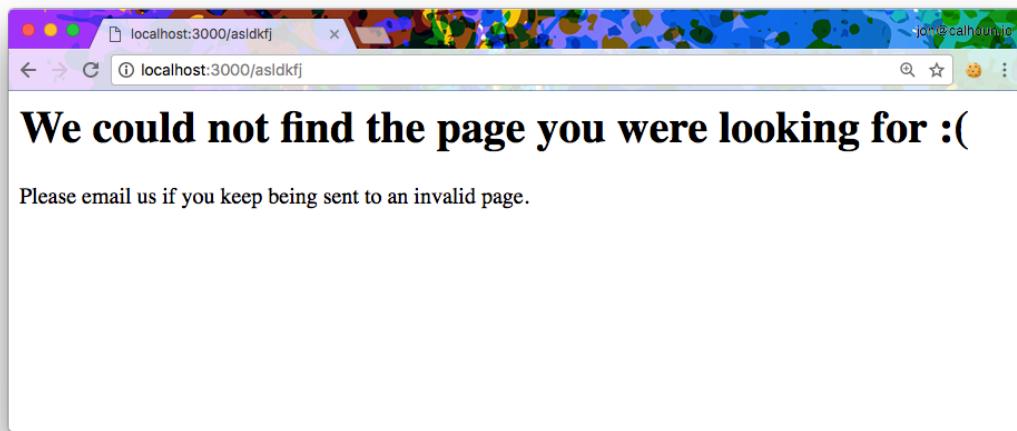


Figure 3.1: Our new 404 page

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

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

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

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

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

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

Box 3.3. Exporting common constants

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

Source code

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

Completed - book.usegolang.com/3.1

Changes - book.usegolang.com/3.1-diff

3.2 Popular routers

Now that we have some basic routing in place we are going to look into ways to improve and simplify our application. Specifically, we are going to explore

open source routers and choose one of those to use with our project.

3.2.1 net/http.ServeMux

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

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

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

3.2.2 [github.com/julienschmidt/httprouter](#)

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

What I love about this router is its strong focus on being simple, fast, and efficient. If you check out some of the benchmarks on its Github page you will notice that it outperforms many other routers. On top of this focus on being fast, it also supports named URL parameters and routing based on the [HTTP request method](#) used. We will learn more about this in a future chapter.

Ultimately I opted not to use [**httprouter**](#) for this book for two reasons. The first is that the library requires you to write http handler functions that take in a third argument if you want to access any path parameters. There isn't anything

wrong with this approach, but I was concerned that it might lead to some confusion with new programmers and opted to instead use the router we are going to cover next - **gorilla/mux.Router**.

3.2.3 [github.com/gorilla/mux.Router](#)

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

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

3.2.4 What about other router>

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

3.3 Using the gorilla/mux router

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

Listing 3.3: Installing `gorilla/mux`.

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

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

Box 3.4. If you see an error...

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

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

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

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

Listing 3.4: Breaking up our handlers into functions.

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

func contact(w http.ResponseWriter, r *http.Request) {
    w.Header().Set("Content-Type", "text/html")
```

```

fmt.Fprintf(w, "To get in touch, please send an email "+
    "to <a href=\"mailto:support@lenslocked.com\">" +
    "support@lenslocked.com</a>.")
}

```

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

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

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

```

import (
    "fmt"
    "net/http"

    "github.com/gorilla/mux"
)

```

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

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

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

```

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

```

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

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

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

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

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

Source code

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

Completed - book.usegolang.com/3.3

Changes - book.usegolang.com/3.3-diff

3.4 Exercises

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

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

Box 3.5. Our first practice problems.

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

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

3.4.1 Ex1 - Add an FAQ page

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

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

3.4.2 Ex2 - Custom 404 page

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

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

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

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

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

3.4.3 Ex3 - [HARD] Try out another router

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

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

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

Chapter 4

A brief introduction to templates

In this chapter we will be introducing a new feature of Go called templates. Specifically, we will be examining the [html/template](#) package from the standard library and exploring how it works.

If you are already familiar with templates in another language you can probably skim or skip this chapter. None of the code examples in this section will be used in your web application, and instead this section focuses on teaching the fundaments of templates in Go with other examples.

That said, you should keep in mind that while templating in Go is very similar to templating in other languages, it is also different in some unique ways which may catch you off guard at times. You might benefit from at least skimming through this section, even if you don't actually code along with the examples.

4.1 What are templates?

At their very core, templates are text files that are used to create dynamic content.

[Mad Libs](#) are an example of a template being used to create dynamic content. In the game one person asks everyone for specific types of words, such as “noun”, “verb”, “place”, or “part of a car”, without giving them any context about how the word will be used. When they have gathered enough words to fill in the entire story, it is read aloud.

Here is an example Mad Lib given from the original book Mad Libs book:

Listing 4.1: Mad Lib Example Sentence

```
_____! he said _____  
exclamation      adverb  
  
as he jumped into his convertible  
  
_____ and drove off with his  
noun  
  
_____ wife.  
adjective
```

A template in Go is very similar to this, but instead of using “noun” or “adverb” we instead use variables. These variables then change based on what resource the user is viewing. For example, we could create a template for our galleries, and then the images that we show will change based on which specific gallery a user is viewing.

4.2 Why do we use templates?

Up until now we have been putting HTML directly inside of strings in our Go code, but this isn't something that works very well in the long run. Not only will it become a nightmare to maintain in the long run, but it is also hard to share any HTML across multiple pages.

Instead, we will be using templates to create web pages with dynamic content. Templates will enable us to create a single layout that uses the same navigation bar on every page without needing to rewrite the code multiple times. This is important because keeping several copies of the same code in sync can be hard to do.

On top of allowing us to reuse code, our templates will also allow us to add some very basic logic to our web pages. For example, we might want to show the "Sign Out" button if the user is signed in, but if the user isn't signed in we instead want to show the "Sign In" and "Sign Up" buttons so that they can start using our application.

All of this (and more!) is made possible by templates.

4.3 Templates in Go

Go provides two pretty amazing template libraries:

- `text/template`¹
- `html/template`²

The way you use each package is identical, but behind the scenes each package works a little differently.

¹<https://golang.org/pkg/text/template>

²<https://golang.org/pkg/html/template>

The text/template package is intended to render templates exactly as they are without doing any additional work. This is often useful when writing a code generator or some other tool where you need the output to match exactly.

On the other hand, html/template is intended to be used when rendering web pages and has some safety features and helpers built into it. This is useful for preventing code injection, a common security issue in web apps.

Box 4.1. Code Injection

Code injection is a common exploit where users try to get your server or visitors of your page to execute code that they write and input into your application. This typically becomes an issue when you don't process or escape input in order to make it invalid.

For example, if someone were to sign up for your web app with the username `<script>alert("Hi!");</script>` and you simply inserted their username on their profile page this could lead to code injection. In this case, anytime someone visited that user's profile page they would see a javascript alert with the text "hi" in it. This becomes dangerous because javascript is typically attached to a user's session, so if you were to instead run some code that visits the `/delete-my-account` page, it could delete the account of any user who visits the injected user's profile page.

Go combats this by encoding text in html templates so that it can't execute. For example, the username `<script>alert("Hi!");</script>` would get encoded to be `<script>alert("Hi!");</script>`, which will render as the original username, but won't be executed by anyone visiting the page as actual code.

Code injection is a complicated problem to tackle on your own, so my advice is to stick to using the html/template package for your web applications. This

will help ensure your web applications are secure.

4.4 Creating a template

Let's go ahead and create a main function that executes a really simple template so we can see it in action. First we need to create a directory for this application. We are going to create the directory `exp` to signify that this is an experimental directory for testing things out.

Listing 4.2: Creating the `exp` directory.

```
# You should start from the lenslocked.com dir
$ cd $GOPATH/src/lenslocked.com
$ mkdir exp
$ cd exp
```

Next up we need to create the file `hello.gohtml` inside of the `exp` directory and add the code in [Listing 4.3](#) to it.

Listing 4.3: Initial contents for `hello.gohtml`.

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

Box 4.2. The extension `.gohtml` is optional.

The extension `gohtml` is commonly used by editors to indicate that you want to use syntax highlighting for Go HTML templates. I believe both packages in Atom (go-plus) and Sublime Text (GoSublime) recognize this by default, so it is the file type I opted for in this book.

This is our first template, and with it we are essentially saying that we want to create an HTML file that has a heading with the contents `Hello, {{ .Name }}!`, and that we want to replace the `{{ .Name }}` portion with whatever is in the variable `Name`.

Let's go ahead and create our a Go file to run this template so you can see it in action. Create a file in the `exp` directory named `main.go` and add the code in Listing 4.4 to it.

Listing 4.4: Initial contents for `main.go`.

```
package main

import (
    "html/template"
    "os"
)

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

    data := struct {
        Name string
    }{"John Smith"}

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

In Listing 4.4's `main` function we first start out by using the `html/template` package's `ParseFiles` function to parse our `hello.gohtml` template file. `ParseFiles` will open up the template file and attempt to validate it. If everything goes well, you will receive a `*Template` and a nil error, otherwise you will receive a nil template and an error.

Box 4.3. Potential issues with auto-importing.

If you are using a plugin to automatically import packages for you, you might need to take a moment to make sure it imports the correct template package.

Go's standard library ships with both a text/template and an html/template package, and both are used nearly identically, so auto-importers have a hard time determining which you intended to use.

As a result, it is always a good idea to verify that you are using the template package you intended after writing code that uses templates. In this case we want the html/template package to be imported.

After calling ParseFiles we check to see if an error was returned. If an error is present we panic. Otherwise we continue on with our program knowing we have a valid template referenced by the `t` variable.

Next up we create the `data` variable, which is an [anonymous struct](#) with a `Name` field. When we instantiate `data` we set the `Name` field to "John Smith".

Finally we execute the template we created earlier, passing in two arguments:

1. Where we want to write the template output
2. The data to be used when executing the template

In our case, we want to write the template output to Stdout, which is your terminal window. This is where functions like `fmt.Println` write to by default, and it is provided to us by the `os` package in Go's standard library.

We also want to provide our template with the data it needs to render the template. In [Listing 4.3](#) we use the line `\{\{ .Name \}\}` to render a name dynamically,

so we need to pass in a data structure with a `Name` field. The variable `data` that we just created has this field set to “John Smith”, so when we provide it to the template we should see the name “John Smith” in the final output.

Let’s go ahead and verify this all by running the code.

```
# Do this from WITHIN the exp dir
$ go run main.go
<h1>Hello, John Smith!</h1>
```

Box 4.4. Template paths are relative.

When you are working with templates, it is important to note that the file paths used to access them are relative to wherever you run your code.

In this example we are accessing a file named `hello.gohtml` in the same folder as our `main.go` file, so we don’t need to include a directory. If we were to instead move to the directory that contains `lenslocked.com` (eg `cd ..` in the console) and we ran our code with `go run lenslocked.com/main.go` it would error because there isn’t a file named `hello.gohtml` in that folder. It is in the `lenslocked.com` folder.

For now this simply means that you need to be in the folder `lenslocked.com` when you run your program, otherwise you are likely to see errors.

If you see some HTML being printed out in your terminal you have successfully created your first template!

Source code

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

Completed - book.usegolang.com/4.4

Changes - book.usegolang.com/4.4-diff

4.5 Contextual encoding

I mentioned before that Go's html/template package does encoding based on the context of the code. While we don't really need to concern ourselves with this at this point in the book, I do want to demonstrate what I mean so that you understand why we are using the html/template package instead of something else.

In order to demonstrate this, we are going to update the **Name** attribute used in our **hello.gohtml** template to be something that isn't safe to render as-is, such as a script tag with a JavaScript alert inside of it.

Open up **main.go** and update the **Name** field to be a script tag like the one shown in Listing 4.5.

Listing 4.5: Using an unsafe string.

```
func main() {
    // ... this stays the same
    data := struct {
        Name string
    }{`<script>alert('Howdy!');</script>`}
    // ... this stays the same
}
```

Now run your program.

```
# Do this from WITHIN the exp dir
$ go run main.go
<h1>Hello, &lt;script&gt;alert(&#39;Howdy!');&lt;/script&gt;!</h1>
```

[H]

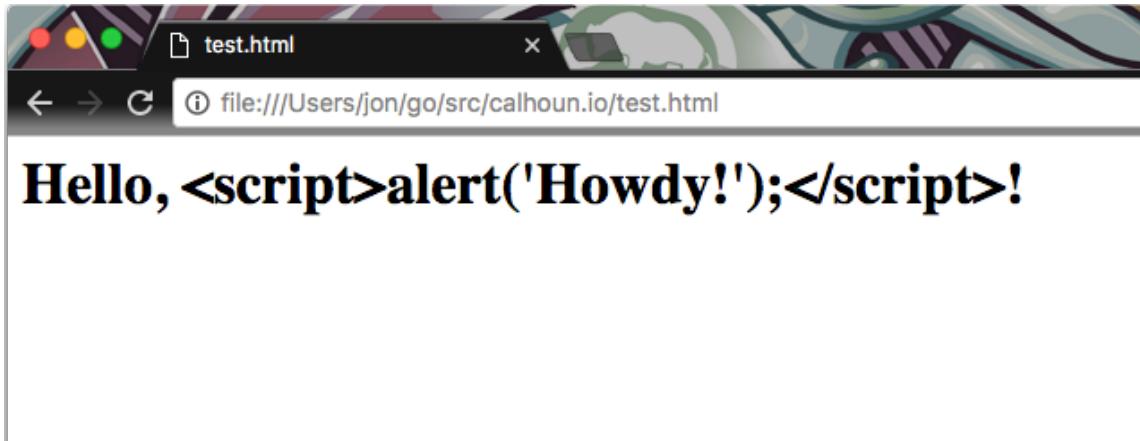


Figure 4.1: A screenshot of test.html

What happened? What is this `<` in our output?

When you run your program, the html/template package will look at every variable you want to print out and will adjust the value of each of these based on where the variable is being used in your template. In this example we are using the variable inside of an HTML section, so the html/template package replaces the `<` character with `<`. This is done to ensure that when your template is viewed in a web browser, you will see the `<` character on the screen and that the browser doesn't try to interpret this as an HTML tag.

While this might not make sense when you see the output, it makes more sense if you view the output from your browser. Copy the output from your program and insert it into a file named `test.html`, then open up the file in your web browser. What do you see?

You should see a page that looks like Figure 4.1. When your web browser sees the text `<`, it instead renders the less than (`<`) character.

Box 4.5. Want to read more about HTML codes?

If you are interested in learning about more HTML symbol and character codes do a quick search for “HTML character codes”.

There are several sites out there that list all of the `&...;` style codes, and you can test each out by editing the `test.html` file we created earlier.

This happens because it would otherwise be impossible for your browser to know whether you wanted the string `<script>` to show the same text on the screen, or if instead you want it to be treated as HTML code. As a result, your browser treats any text inside of `< >` as an HTML element. If you want to show the `<` character on your website you need to use the `<` string, which would be displayed as `<` by the browser as it renders the page.

If the `html/template` package hadn’t encoded the `<` character, your web page would have instead parsed the name as if it were an actual script and rendered one of those annoying alerts messages like in [Figure 4.2](#).

Now that you have a very basic understanding of how templates work in Go, we are going to jump back to our web application and use them to render our pages, but first let’s try out a few exercises with templates.

Source code

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

Completed - book.usegolang.com/4.5

Changes - book.usegolang.com/4.5-diff

[H]

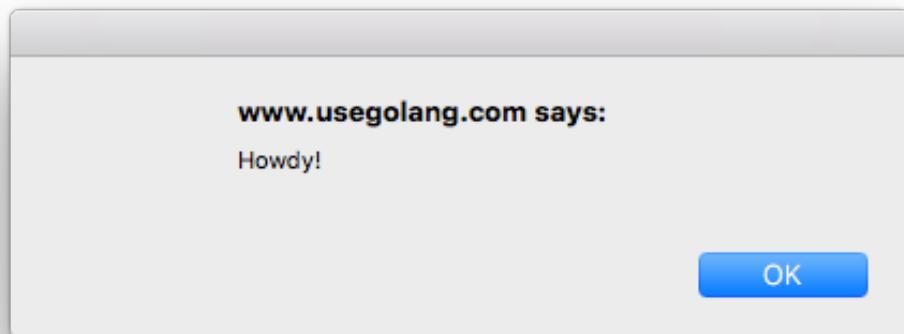


Figure 4.2: An annoying JavaScript alert

4.6 Exercises

At this point you should 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. As we progress through this course we will continue to use templates and will slowly introduce more advanced techniques of using them.

Box 4.6. Additional resources

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

- The docs - <https://golang.org/pkg/html/template/>

- A free series I wrote on HTML templates - <https://www.calhoun.io/html-templates-in-go/>
- An official Go tutorial covering HTML templates - <https://golang.org/doc/articles/wiki/>

4.6.1 Ex1 - Add a new template variable

In this exercise your goal is to add a new variable to your `gohtml` template, and then to update your Go code in `exp/main.go` to provide some data for this variable.

You can approach this in two steps:

1. Adding a new field to the `data` variable in your go code
2. Using that new field inside of your template

4.6.2 Ex2 - Experiment with different data types

Up until now we have only used strings in our HTML templates. Try experimenting with a few other data types, like integers, floats, maps, and slices.

While we are using the `html/template` package, the `text/template` docs provide instructions on how to use both template packages so you might want to refer to both sets of docs while experimenting here.

- `text/template` docs - <https://golang.org/pkg/text/template/>
- `html/template` docs - <https://golang.org/pkg/html/template/>

4.6.3 Ex3 - [HARD] Learn how to use nested data

Create a nested structure and learn to access nested fields inside of your template.

For example, if you add a **map** to your **data** variable, how would you access individual keys and values of that map?

This will be 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.

4.6.4 Ex4 - [HARD] 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.

Again, this is going to be harder to accomplish if your experience with templates and Go are limited, but it is worth giving it a shot.

Chapter 5

Understanding MVC

Before we start using templates in our application we are going to take one more quick detour to discuss the model-view-controller (MVC) architectural pattern.

MVC is important to understand, especially as a new web developer, because it gives us a rough idea of how to organize and structure our code. As you become a more experienced developer you will likely adopt custom design patterns that work well for your application, but MVC is a great starting point for nearly any application. This is why so many frameworks like Rails, Sails.js, and Django use MVC or something very close to it.

If you are already familiar with MVC, you can safely skip this chapter.

5.1 Model-View-Controller (MVC)

As you learn to program, you will eventually start to notice that some code bases are easy to navigate while others are confusing and hard to maintain or update. At first it is hard to tell what makes the difference, but over time you will quickly learn that the biggest factor is how well organized and structured the code is.

The reason for this is simple - if you need to jump in and start adding a new feature, fixing a bug, or doing anything else to a program it is much easier to do if you can guess where in the code you need to start looking. When you have no idea where to even start, you are left digging through hundreds of source files, but in a well structured program you can often guess where the code is even if you haven't read the source code in the past.

Model-View-Controller, commonly referred to as MVC, is an architectural pattern that is designed to help organize applications by separating code based on what it is responsible for, and then relying on developers to make sure the correct code is put in each package. Specifically, the MVC pattern introduces three distinct roles that code can fall under:

Views are responsible for rendering data. That's it. Given a specific page that we want to render, and data for that page, our view is responsible for generating the correct output.

In our case, this is typically HTML code that we want to return to the end user's browser, but in other applications this could vary. The important thing to remember is that the code in a view should have as little logic going on as possible, and should instead focus entirely on displaying data. If a lot of logic starts to creep into your views you are likely doing something wrong, and it could become problematic down the road.

In my applications I also like to have the ability to create common layouts that will be shared across my app, and then use those inside my view layer. This isn't a requirement in the MVC pattern, but it is incredibly helpful so we will be adding this functionality to our application as well.

Controllers are used to handle most of the business logic that happens behind the scenes for a web request. It won't directly create views, or update data in the database, but instead will use views and models to do these things.

You can think of controllers as your [air traffic controllers](#). Air traffic controllers are the people that inform each plane at an airport where to fly, when to land, and on which runway to land. They don't actually do any piloting, but instead are

in charge of telling everyone what to do so that an airport can operate smoothly.

Similarly, your controller shouldn't have too much logic in it, but will instead pass data around to different pieces of your application that actually handle performing whatever work needs done.

Models are the final part of MVC, and these are responsible for interacting with your raw data. This typically means interacting with your database, but it could also mean interacting with data that comes from other services or APIs.

For example, our web application is going to have user accounts, and we will represent them in a database as user objects. To access this data, we will create a **User** model that is representative of that data in our database, and handles doing things like updating a user's contact information in the database.

While these three categories of code will be very important as we proceed, it is also important to note that just because we are using these categories, it *DOES NOT* mean that we need to put all of our code in these three categories.

On the contrary, we will be creating several other packages while we develop our web application that will work alongside these three primary categories to get work done. For example, the router code that we discussed and implemented in [Section 3.3](#) doesn't easily fall into these categories, and I prefer to keep all of my routing logic in its own **routing** package.

5.2 Walking through a web request

MVC is often a little easier to understand if we take a normal web request and examine how each part of the request would be handled. To do this, we will walk through a web request where a user is attempting to update their contact information, and we will follow along with [Figure 5.1](#) and [Figure 5.2](#).

[H]

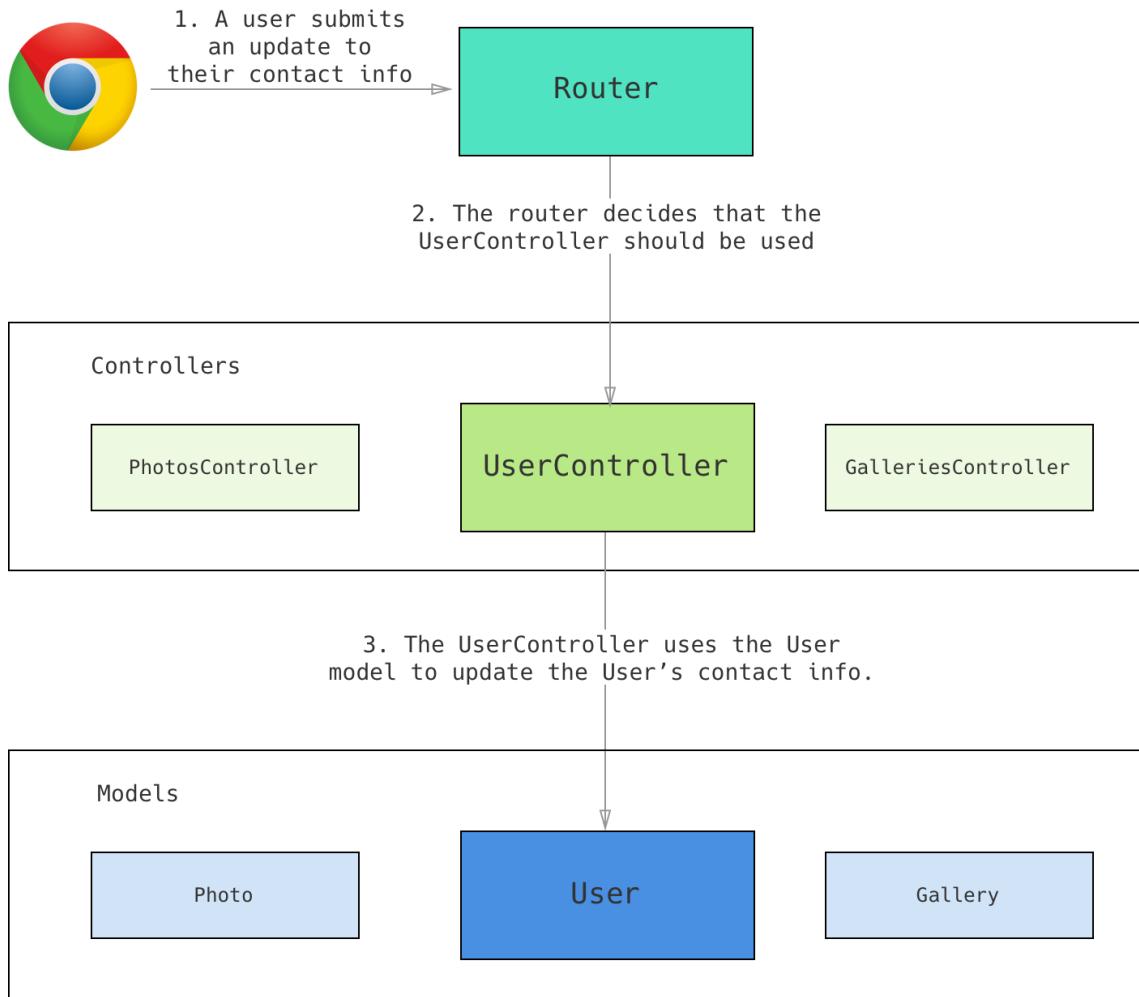


Figure 5.1: MVC Walkthrough Steps 1 to 3

1. A user submits an update to their contact information

This would typically happen when the user goes to their account settings and updates a few fields there. For example, they might update their name and email address. This web request is then sent to our server, where the router is the first code to take action.

2. The router decides that the UserController should be used

When the router gets the web request, it realizes that this is a request to update a User's contact information based on the URL and [HTTP method](#), so it forwards the request along to the **UserController** to handle the request.

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

While handling the request, the **UserController** will need to make a change to data stored in our database, but with MVC our controllers don't interact directly with the database. Instead the controller will use the **User** model to update the contact information, which allows us to isolate all of our database specific code so that our controllers don't need to worry about it.

4. The User model returns the updated data to the UserController

After updating the user in the database, the **User** model will then return the updated user object to the controller so that the controller can use this in the rest of its code.

[H]

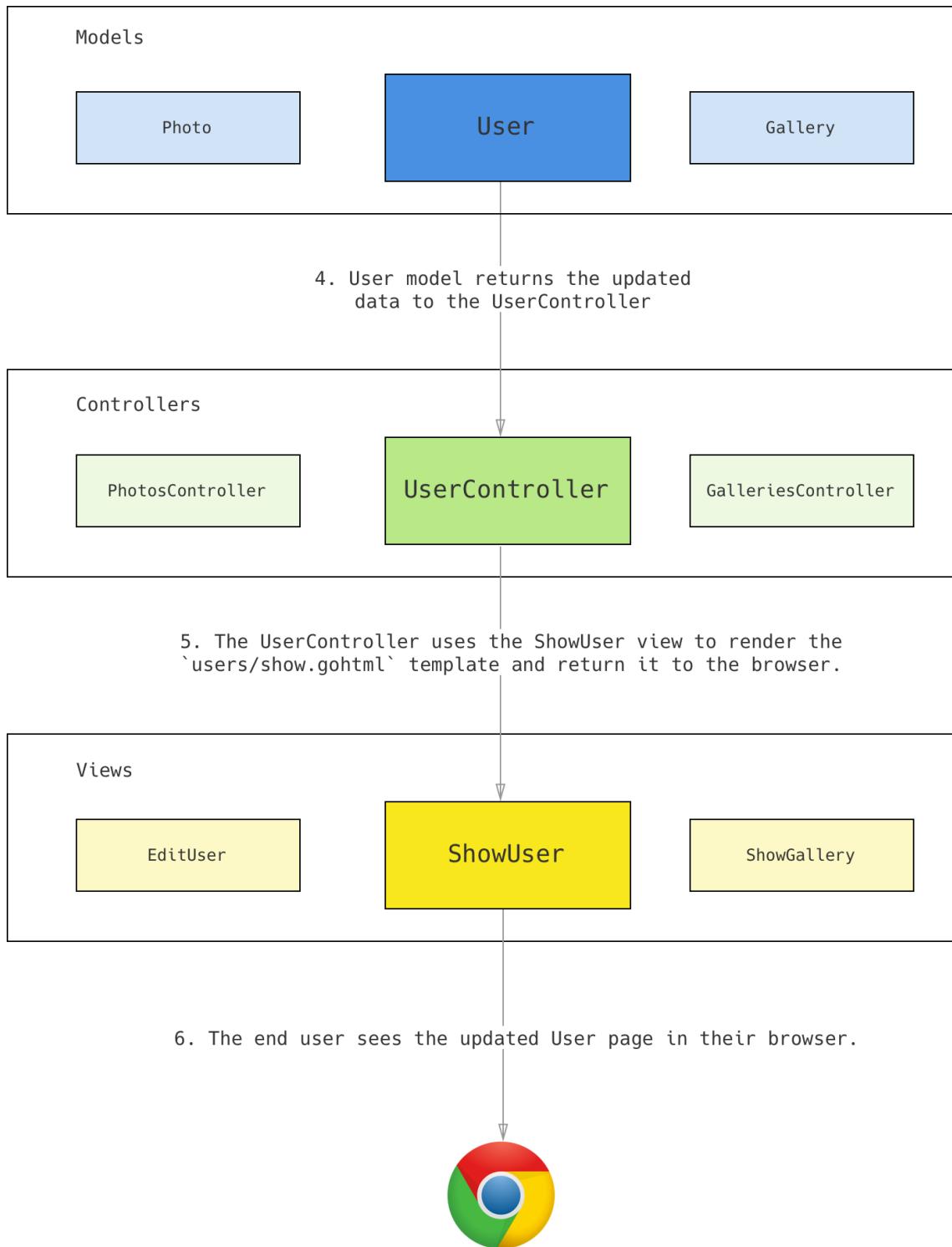


Figure 5.2: MVC Walkthrough Steps 4 to 6

5. The `UserController` uses the `ShowUser` view to render the correct template

Once the user controller gets the user back, it can then use the `ShowUser` view to render the updated user object to the end user. Once again, this allows our controller to operate as more of a director that decides what code should run next, but doesn't get bogged down with a lot of logic of its own.

6. The end user sees the updated User page in their browser

After everything is said and done, our `ShowUser` view will have rendered the HTML page that shows a user and the end user will now see their user account with the contact information updated. Mission success!

5.3 Exercises

For this section we won't be coding during our exercises, but instead will just review a few questions to ensure you understood what we covered.

5.3.1 Ex1 - What does MVC stand for?

What does each letter in the MVC acronym stand for?

5.3.2 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.

5.3.3 Ex3 - What are some benefits to using MVC?

We discuss a few reasons why MVC were important. Just take a moment to refresh your memory on what those are.

It is also good to remember that we DO NOT have to place all of our code into the MVC pattern. Not everything we write will be a model, view, or controller, and trying to force all of your code to fall into one of these three categories can be as bad as not using MVC at all.

Chapter 6

Creating our first views

In the last chapter we covered the MVC pattern, and in the chapter before we introduced templates. In this chapter we are going to put all of that together to create the view portion of MVC.

Like everything else we do in this book, this will be an incremental process. We will start with a fairly basic view and then slowly improve and update our code to make our views easier to declare and reuse.

Finally, we will create a layout for our web application using Bootstrap¹. Bootstrap is an HTML, CSS, and JS framework that will help us quickly create a decent looking website without spending hours writing custom CSS and HTML. It is a great starting place for most web applications because it allows you to get a first version launched quickly and then you can iterate on the design after launching.

If you haven't done so already, leave the `exp` directory we created in [Section 4.4](#) and navigate back to our web app's base directory.

¹<http://getbootstrap.com/docs/3.3/>

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

6.1 The home template

We are going to start by creating a home view. So far we have been creating our HTML by calling `Fprint` with an HTML string, but moving forward we will move that HTML to a template. We will start with the same contents in our `home` function for now to keep things simple.

First, we need to create a directory to store our views. We will call this directory `views` so it is easy to find.

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

Next we need to create our first template - `home.gohtml`.

```
$ atom views/home.gohtml
```

After that we need to add the contents of Listing 6.1 to our template.

Listing 6.1: Initial contents for `views/home.gohtml`.

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

Now we need to render this template instead of writing a string directly with `Fprint`. To do this we are going to create a global variable that will store our parsed template, and then use this template inside of our `home` function.

First we need to open `main.go` and add the global `*template.Template` variable near the top of the file after the imports section. We are going to name this variable `homeTemplate` like in [Listing 6.2](#).

Listing 6.2: Creating the `homeTemplate` global.

```
var homeTemplate *template.Template
```

Box 6.1. We will clean this up later...

Global variables like we just created are usually frowned upon. There are many reasons for this, but the main reasons are that they make code harder to test and can have unexpected side effects that cause bugs.

We will eventually clean this code up, but for now we are using them because they make it easier to understand what is going on in our code.

Next we need to make sure this variable gets assigned to a parsed template. As of now it is just an empty pointer. We are going to do this inside of the `main` function.

Add the code in [Listing 6.3](#) at the very start of the `main` function. Everything else in the function should remain, but will happen after this code executes.

Listing 6.3: Parsing the `views/home.gohtml` template.

```
func main() {
    var err error
    homeTemplate, err = template.ParseFiles("views/home.gohtml")
    if err != nil {
        panic(err)
    }
    // ...
}
```

Finally, we need to update our `home` function so that it uses our newly template. To do this we are going to call the template's `Execute` method like we did in Listing 4.4, but instead of writing to Stdout we are going to write the results to our ResponseWriter. This means the results of executing the template will be returned to the user who is making a web request to our application.

The updated code is shown in Listing 6.4.

Listing 6.4: Updating the `home()` function.

```
func home(w http.ResponseWriter, r *http.Request) {
    w.Header().Set("Content-Type", "text/html")
    if err := homeTemplate.Execute(w, nil); err != nil {
        panic(err)
    }
}
```

While executing the `homeTemplate`, we also added some error handling to our code. When calling the `Execute` method, it is possible for our template to encounter an error. For example, it might try to access a variable that we didn't provide and this might lead to an error.

If we encounter an error we are going to simply panic in our code so that we know about the issue. We shouldn't actually see any errors right now, but it is good to have this precaution in place for now as it will help us find any unexpected bugs in our code during development.

You might also notice that in Listing 6.4 we are also passing `nil` to the `Execute` method instead of any actual data. This is because our template doesn't currently use any dynamic data, but we still need to provide that second argument. The simplest way to pass “no data” in this case is to pass `nil`.

At this point you are ready to stop your application (`ctrl + c`) and then restart it. If all goes well, your home page shouldn't look any different, but should now be rendered via the `home.gohtml` template.

```
$ go run main.go  
# Then go visit localhost:3000 in your browser
```

If you want to try changing the contents of your template you can, but you will need to stop and restart your application to see the changes. Our application only parses the template once when it starts up, so it will not notice any changes to the file until the application is restarted. This is tedious during development, but will speed up our application when we ship to production because it doesn't need to reread template files every time a user visits a page.

The completed code from this section can be found at:

- book.usegolang.com/6.1

You can also see what code was added (green) and what code was removed or changed (red) at:

- book.usegolang.com/6.1-diff

6.2 The contact template

In this section we are going to do pretty much the same thing we just did in the last section, but for our contact page.

First up is the template file. We need to create it, then move our HTML from the **contact** function into it.

```
$ atom views/contact.gohtml
```

Listing 6.5 has the source code for the contact template.

Listing 6.5: Initial contents for `views/contact.gohtml`.

```
To get in touch, please send an email to
<a href="mailto:support@lenslocked.com">
    support@lenslocked.com
</a>.
```

Now we need to update `main.go` like before, and again we need to do three things:

1. Create the global `contactTemplate` variable
2. Parse our template file at and assign it to the variable
3. Update our `contact` function to use the template variable

My suggestion at this point is to try to do these three steps on your own. It is nearly identical to what we did in [Section 6.1](#), but using the contact page instead of the home page.

Once you have given it a shot, you can compare your code with the code in [Listing 6.6](#).

Listing 6.6: `main.go` using both templates.

```
package main

import (
    "html/template"
    "net/http"
```

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

var homeTemplate *template.Template
var contactTemplate *template.Template

func home(w http.ResponseWriter, r *http.Request) {
    w.Header().Set("Content-Type", "text/html")
    if err := homeTemplate.Execute(w, nil); err != nil {
        panic(err)
    }
}

func contact(w http.ResponseWriter, r *http.Request) {
    w.Header().Set("Content-Type", "text/html")
    if err := contactTemplate.Execute(w, nil); err != nil {
        panic(err)
    }
}

func main() {
    var err error
    homeTemplate, err = template.ParseFiles(
        "views/home.gohtml")
    if err != nil {
        panic(err)
    }
    contactTemplate, err = template.ParseFiles(
        "views/contact.gohtml")
    if err != nil {
        panic(err)
    }

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

Stop your server and restart it to make sure everything is working as intended. Visit both the contact page (localhost:3000/contact) and the home page to make sure both pages work.

Everything should look the same, but your templates are being used to render the HTML output.

Source code

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

Completed - book.usegolang.com/6.2

Changes - book.usegolang.com/6.2-diff

6.3 Creating a reusable Bootstrap layout

Earlier, when discussing views in the MVC chapter, I mentioned that I personally like to create views that allow me to reuse templates. We are going to start creating a navigation bar and footer for our website, so it would be nice to do this with a layout. That way we can always show the same navigation bar and footer, and our templates can focus solely on filling in the contents of each individual page.

In order to do this, we need to learn how named templates work.

Box 6.2. We will also be using Bootstrap

While we are at it, we are going to include [Bootstrap](#) in our layout. Some people dislike the look of Boostrap because it is used so frequently on the web, but I believe it is a great tool for creating the first version of a web application because it allows you to build quickly and then customize the look and feel of your application once you have it working.

As much as I love a pretty website, they aren't much good to users unless they actually work which is what we are focusing on now.

6.3.1 Named templates

As our templates start to grow we will quickly find that it is much easier to manage templates if we can break them into smaller pieces and reuse them across our application. Luckily, Go's template library comes with a solution for this out of the box in the form of named templates.

In order to use named templates, you must first define a name for your template. This tells the `html/template` package how you want to refer to a template in the future so that you can include it elsewhere. The name must also be unique so that the template package knows exactly which template you are referring to.

Naming an existing template is very easy. You simply need to start your template code with `{{define "name-here"}}`, and then when your template is completed you close it with `{{end}}`.

[Listing 6.7](#) shows an example of what a named footer template might look like.

Listing 6.7: A named footer template example.

```
 {{define "footer"}}
  <footer>
    <p>
      Copyright 2016 lenslocked.com
    </p>
  </footer>
 {{end}}
```

If you wanted to use the template shown in [Listing 6.7](#) in another template you would do so by writing the following code inside of your template:

```
 {{template "footer"}}
```

Seeing this all in action really helps solidify it, so let's do that now.

We will first create a layouts folder inside of our views directory to store our shared layout templates. Then we will create a file named **footer.gohtml** inside of that directory.

```
$ cd $GOPATH/src/lenslocked.com
$ mkdir views/layouts
$ atom views/layouts/footer.gohtml
```

Then add the code in Listing 6.7 to the footer template that we just created.

Before we can use this layout we need to tell our Go code about it. Open up **main.go** and update the your main function. We are going to be altering both of our **ParseFiles** function calls to tell them both about the footer template we just created so that both of these views can use it. The code is shown in Listing 6.8.

Listing 6.8: Adding the footer layout.

```
func main() {
    var err error
    homeTemplate, err = template.ParseFiles(
        "views/home.gohtml",
        "views/layouts/footer.gohtml")
    if err != nil {
        panic(err)
    }
    contactTemplate, err = template.ParseFiles(
        "views/contact.gohtml",
        "views/layouts/footer.gohtml")
    if err != nil {
        panic(err)
    }
    // ... The code below here doesn't change
}
```

By adding the footer template to our list of parsed files we now have access to the named “footer” template inside of our home and contact templates. We just need to update our code so that it knows to also render the footer template.

Update both the home and contact templates using the code in Listing 6.9 and Listing 6.10.

Listing 6.9: Home template with a footer.

```
<h1>Welcome to my awesome site!</h1>  
&{{template "footer"}}&
```

Listing 6.10: Contact template with a footer.

```
To get in touch, please send an email to  
<a href="mailto:support@lenslocked.com">  
    support@lenslocked.com  
</a>.  
&{{template "footer"}}&
```

Stop (**crtl + c**) and restart your your server, then navigate to the home and contact pages to see if the footer is being rendered.

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

If everything is working, you should see a footer on both pages similar to the one in Figure 6.1.

Box 6.3. Passing data to named templates

When we render the footer template in Listing 6.9 we are not passing any data into the footer template. As a result, the footer template won't ever have access to any of the data passed into our template by the **Execute** method.

If you need to pass data into a template you can do so by providing it as an additional argument in your code.

[H]

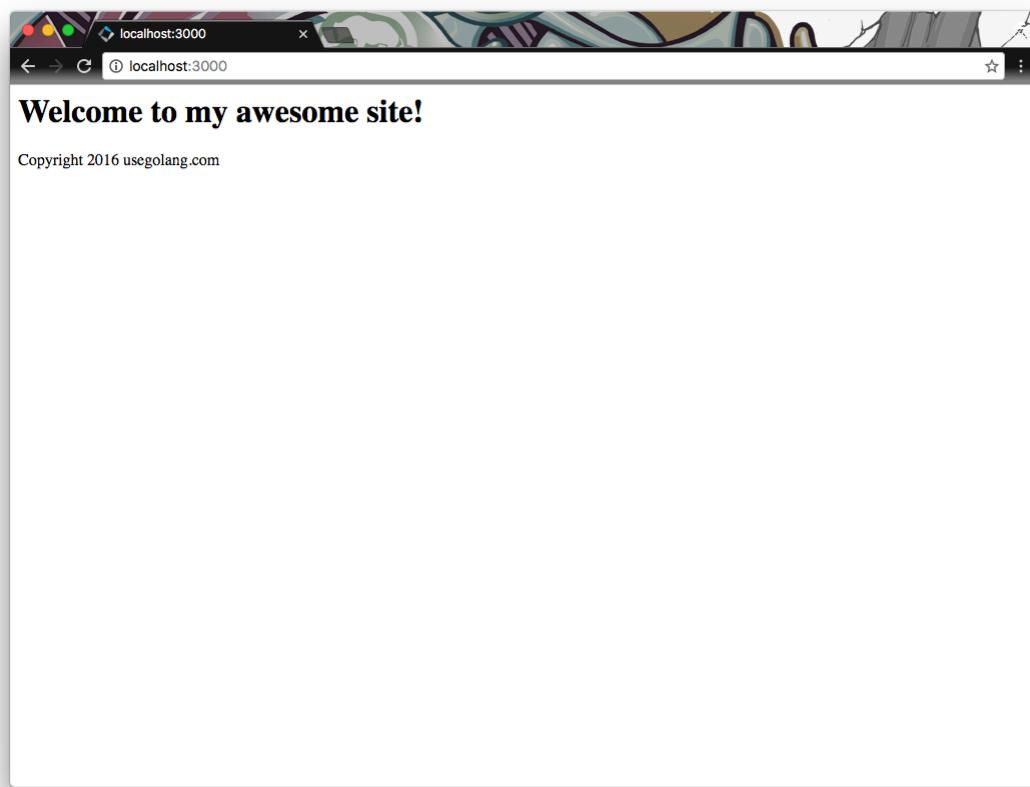


Figure 6.1: The Home Page with the Footer Template

```
 {{template "footer" .}}
```

The period (`.`) in the code above is saying “pass all of the data we have to the footer template”. You can also pass a specific piece of data, like a user’s name.

```
 {{template "footer" .Name}}
```

Source code

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

Completed - book.usegolang.com/6.3.1

Changes - book.usegolang.com/6.3.1-diff

6.3.2 Creating a view type

We are eventually going to take what we learned about named templates and apply it to create a layout for all of our pages, but first we are going to address another issue that you may have noticed.

In order to use our footer template, we had to include this file in every single template that we parsed. For one or two templates this is fine, but if we start doing this more frequently it could quickly become problematic.

Not only is this a lot of extra code, it is also really hard to maintain. Every time we add a new layout file we need to change code for each template that we use

in our application.

Instead of doing that, we are going to create a **View** type that handles all of this logic for us. This will allow us to reuse some of our code and will simplify maintenance. To get started, we are first going to create a new Go source file to contain our view type.

```
$ atom views/view.go
```

Then add the code in Listing 6.11 to the new source file.

Listing 6.11: Creating the **View** type.

```
package views

import "html/template"

func NewView(files ...string) *View {
    files = append(files, "views/layouts/footer.gohtml")
    t, err := template.ParseFiles(files...)
    if err != nil {
        panic(err)
    }

    return &View{
        Template: t,
    }
}

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

At the bottom of Listing 6.11 we are declaring our **View** type. It is a struct that contains one attribute - a pointer to a **template.Template**, which will eventually point to our compiled template.

Our view type is going to evolve over time, but for now all we really know is that it needs to store the parsed template that we want to execute, so we will start there.

Above our type declaration we have the `NewView` function. This function takes in any number of strings as its argument using a *variadic parameter* (explained in [Box 6.4](#)).

We are creating a the `NewView` function in order to make it easier to create views. This function is going to handle appending common template files to the list of files provided, then it will parse those template files, and finally it will construct the `*View` for us and return it.

Box 6.4. Variadic parameters (...)

A *variadic parameter*, is essentially a function parameter that can be 0, 1, or any other number of items as long as they match the correct type, and are the last argument to the function call. This is represented by using the triple dot (...) operator before the argument type when declaring the function `NewView()`. We are saying that our function can take in any number of strings, and then Go merges all of these strings into a string slice (`[]string`) named `files` for us to use inside of the function.

Similarly, we can use the ... operator after a variable name to “unravel” the items in a slice. This is necessary because the `template.ParseFiles()` function expects a variadic parameter of type `string`, meaning that it expects 0, 1, or more strings to be passed in like we were doing before in [Listing 6.8](#), but `files` isn’t a string, it is a slice of strings. The ... operator allows us to take a slice of strings and have it be treated as it if it was a list of strings, like `template.ParseFiles("a", "b", "c")`.

Now that we have a `View` type, let’s put it to use. Open up `main.go` and update our global templates to instead be of the type `*View`. In order to access this type we need to import the `views` package and then reference the type as `*views.View`. We should also update the names of our variables to match their new type. This is all shown in [Listing 6.12](#)

NOTE: Our code won't compile just yet. You will need to proceed to the end of this section before it will be ready to run.

Listing 6.12: Updating the global variables.

```
package main

import (
    "html/template"
    "net/http"

    "lenslocked.com/views"
    "github.com/gorilla/mux"
)

var homeView *views.View
var contactView *views.View

// ... This all stays the same
```

Next we need to update our **main** function to handle constructing our views using the **NewView** function we created. This is shown in Listing 6.11.

Listing 6.13: Updating **main()**.

```
func main() {
    homeView = views.NewView("views/home.gohtml")
    contactView = views.NewView("views/contact.gohtml")

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

Notice how much shorter our **main** function became now that have offloaded a lot of the logic to the **NewView** function?

Finally, we need to update our **home** and **contact** functions to use the view variables. This is roughly the same as before, but instead of directly referencing

the template we are going to access the **Template** field of each view object. This code is in Listing 6.14.

Note: I adjusted the error checking code to span a few lines in order to make sure the code isn't too wide for the ebook.

Listing 6.14: Updating **home()** and **contact()**.

```
func home(w http.ResponseWriter, r *http.Request) {
    w.Header().Set("Content-Type", "text/html")
    err := homeView.Template.Execute(w, nil)
    if err != nil {
        panic(err)
    }
}

func contact(w http.ResponseWriter, r *http.Request) {
    w.Header().Set("Content-Type", "text/html")
    err := contactView.Template.Execute(w, nil)
    if err != nil {
        panic(err)
    }
}
```

With all of these changes you should be able to restart your server and see the new views in action. Your pages will look exactly the same as before, but boy doesn't that code just look so much cleaner? □

Source code

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

Completed - book.usegolang.com/6.3.2

Changes - book.usegolang.com/6.3.2-diff

6.3.3 Creating the Bootstrap layout

We are finally ready to create a layout that uses Bootstrap². To do this we are first going to create our template file.

```
$ atom views/layouts/bootstrap.gohtml
```

Next, insert the code in Listing 6.15 into the new template file.

Listing 6.15: Initial `bootstrap.gohtml`.

```
{{define "bootstrap"}}
<!DOCTYPE html>
<html lang="en">
  <head>
    <title>LensLocked.com</title>
    <link
      href="//maxcdn.bootstrapcdn.com/bootstrap/3.3.7/css/bootstrap.min.css"
      rel="stylesheet">
  </head>

  <body>

    <div class="container-fluid">
      <!-- Our content will go here... somehow -->
    </div>

    <!-- jquery & Bootstrap JS -->
    <script src="//ajax.googleapis.com/ajax/libs/jquery/1.11.3/jquery.min.js">
    </script>
    <script src="//maxcdn.bootstrapcdn.com/bootstrap/3.3.7/js/bootstrap.min.js">
    </script>
  </body>
</html>
{{end}}
```

I'm not going to spend much time on the actual HTML here - if you are interested in learning more about HTML I suggest you check out the beginner resources in the front of this book (see Section 1.3).

²<http://getbootstrap.com/docs/3.3/>

What I am going to talk about is a little issue we just ran into. We have a bootstrap layout created, but how are we going to insert our pages *inside* of the layout? We haven't done that before.

Remember how we included the footer template inside of our home and contact templates earlier in this chapter? What we would like to do is something similar to that, but we want to insert a different template depending on what page the user is visiting.

There are several different ways to do this, but in my opinion the easiest is to always render the same template from within our bootstrap layout. Let's call that template **yield**.

Then when we want to change what page is rendered in our web application, we simply need to provide a new definition for the **yield** template.

Once again this is a situation where seeing the final code will help clear up any confusion. First we will open up our home and contact templates and turn them into named templates with the name **yield**.

```
$ atom views/home.gohtml
```

Then update the code to match Listing 6.16.

Listing 6.16: Turn **home.gohtml** into a named template.

```
{{define "yield"}}
<h1>Welcome to my awesome site!</h1>

{{template "footer"}}
{{end}}
```

After that we need to update the contact template.

```
$ atom views/contact.gohtml
```

We are doing the same thing here - naming our template **yield** as shown in Listing 6.17.

Listing 6.17: Turn **contact.gohtml** into a named template.

```
{{define "yield"}}
  To get in touch, please send an email to
  <a href="mailto:support@lenslocked.com">
    support@lenslocked.com
  </a>.

  {{template "footer"}}
{{end}}
```

Next we need to update our bootstrap layout to render the **yield** template.

```
$ atom views/layouts/bootstrap.gohtml
```

And update the “Our content will go here” comment section with the code shown in Listing 6.18. The entire **div** is shown in the listing, but you really only need to add the template line.

Listing 6.18: Use **yield** in **bootstrap.gohtml**.

```
<div class="container-fluid">
  {{template "yield" .}}
</div>
```

Note: In Listing 6.18 we added the `.` after the template name to tell our program that we want to pass any data provided to the bootstrap template on to the yield template. Without this, the yield template wouldn't have access to any of our data we pass into our templates.

If we tried to run our code at this point you would end up getting a blank page. Why does that happen?

Now that we have named our home and contact templates both **yield**, we need to tell our template to execute those named templates before their contents will be rendered. One way to do this is to use the `ExecuteTemplate`³ method provided by the `Template` type.

```
// NOTE: We won't be using this code in our app, so you do
// not need to add it to your app.
homeView.Template.ExecuteTemplate(w, "yield", nil)
```

Doing this would tell our template that we want to execute a template named “yield”, writing the results to `w`, and using the data `nil`.

We are going to update our code to do something similar to this, but we are going to store the name of the template we want to execute inside of our `View` type that we created earlier.

To do this we need to make some changes to our `View` type. We are first going to store what template we want our view to execute. We could call this whatever we want, but because we know that this will often refer to a layout template (eg **bootstrap**) we are going to name this field **Layout**.

Open up `views.go` and update the `View` type by adding the `Layout` field, like in [Listing 6.19](#).

Listing 6.19: Adding **Layout** to `NewView()`.

```
type View struct {
    Template *template.Template
    Layout   string
}
```

³<https://golang.org/pkg/html/template/#Template.ExecuteTemplate>

Next we need to update our home and contact functions to use this layout when executing the templates. Open up `main.go` and use Listing 6.20 as a guide to do this.

Listing 6.20: Using the `Layout` in handlers.

```
func home(w http.ResponseWriter, r *http.Request) {
    w.Header().Set("Content-Type", "text/html")
    err := homeView.Template.ExecuteTemplate(w,
        homeView.Layout, nil)
    if err != nil {
        panic(err)
    }
}

func contact(w http.ResponseWriter, r *http.Request) {
    w.Header().Set("Content-Type", "text/html")
    err := contactView.Template.ExecuteTemplate(w,
        contactView.Layout, nil)
    if err != nil {
        panic(err)
    }
}
```

While our code will compile right now, we aren't done making changes yet. We still haven't actually set the `Layout` field for any of our views, so if we try to run our code it still won't know which layout we want to render and will likely panic at some point.

To address this issue, we are going to update the `NewView` function in the `view.go` file. We are going to make two changes here:

1. We are going to accept a `layout` parameter so that we can set this on the view we are creating
2. We are going to add the bootstrap layout to the slice of template files we are parsing so that it is available for rendering

The code for these changes is shown in Listing 6.21

Listing 6.21: Adding **Layout** to **NewView()**.

```
func NewView(layout string, files ...string) *View {
    files = append(files,
        "views/layouts/footer.gohtml",
        "views/layouts/bootstrap.gohtml")
    t, err := template.ParseFiles(files...)
    if err != nil {
        panic(err)
    }

    return &View{
        Template: t,
        Layout:   layout,
    }
}
```

Once again, we have made a change that breaks out code. Drats!

Because we updated the **NewView** function, we need to update any code that called this function. We were calling **NewView** from within our **main** function, so head over to **main.go** and update the main function using [Listing 6.22](#) as a guide.

We will be using the “bootstrap” layout that we just created for all of our views, which is why it is added in as the first argument to both **NewView** function calls, but due to the way we wrote our code we could actually render the home and contact views using different layouts if we wanted to test out a new website layout.

Listing 6.22: Passing a **layout** to **NewView()**.

```
func main() {
    homeView = views.NewView("bootstrap",
        "views/home.gohtml")
    contactView = views.NewView("bootstrap",
        "views/contact.gohtml")
    //...
}
```

Finally, after making all those small incremental changes, our code should com-

pile again. Go ahead and stop the server and restart it.

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

If we did everything right, our page should look *slightly* different than before. The change is subtle now because we don't have a navbar or any other bootstrap components, but the font should have changed a bit because our bootstrap layout, which includes bootstrap CSS, is now being used. [Figure 6.2](#) shows a screenshot of our updated home page.

This might not seem like much now, but as we continue to add new elements to our page you will start to see how Bootstrap helps us create decent looking pages much faster.

Source code

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

Completed - book.usegolang.com/6.3.3

Changes - book.usegolang.com/6.3.3-diff

6.4 Adding a navigation bar

Now that we have things setup, adding new templates and pages is going to be a breeze! To illustrate this, we are going to add a [navbar](#) to our layout that will be shown on every page.

First we need to create a template file.

[H]

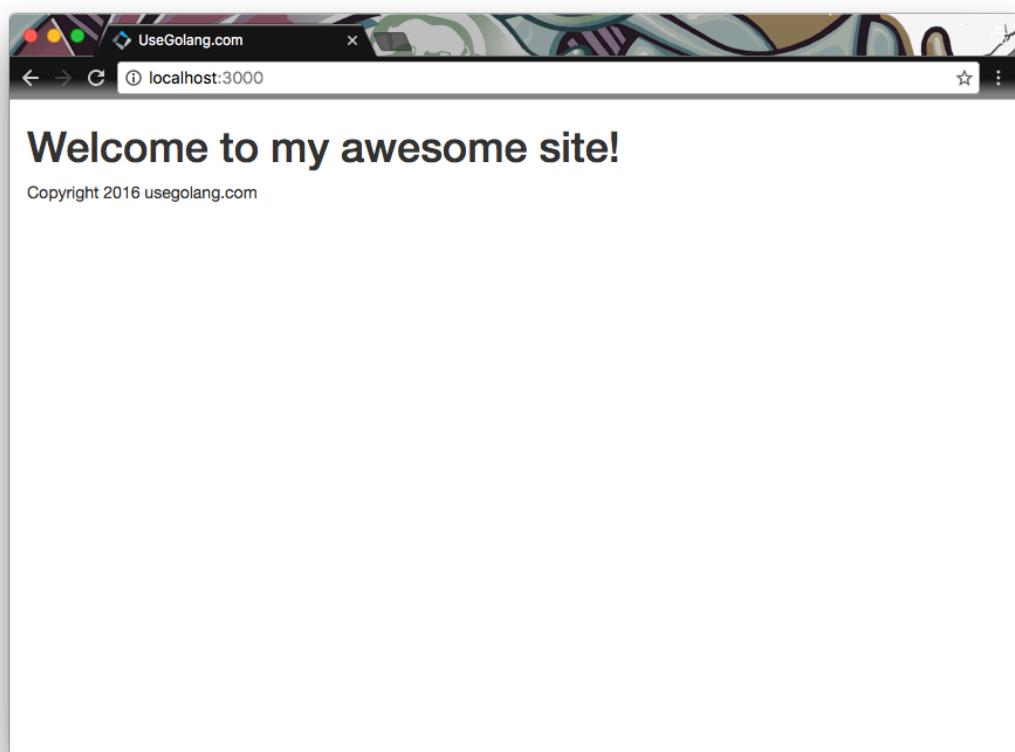


Figure 6.2: Bootstrap live!

```
$ atom views/layouts/navbar.gohtml
```

After that we need to add the template code to the navbar. To do this we are going to grab some code from the Bootstrap navbar docs⁴, and then remove a few of the pieces we won't be using.

Specifically, we won't need the search bar, or the links on the right side of the navbar so both of these have been removed from [Listing 6.23](#).

Listing 6.23: Creating a navbar.

```
{{define "navbar"}}
<nav class="navbar navbar-default">
  <div class="container-fluid">
    <div class="navbar-header">
      <button type="button" class="navbar-toggle collapsed"
        data-toggle="collapse" data-target="#navbar"
        aria-expanded="false" aria-controls="navbar">
        <span class="sr-only">Toggle navigation</span>
        <span class="icon-bar"></span>
        <span class="icon-bar"></span>
        <span class="icon-bar"></span>
      </button>
      <a class="navbar-brand" href="#">LensLocked.com</a>
    </div>
    <div id="navbar" class="navbar-collapse collapse">
      <ul class="nav navbar-nav">
        <li><a href="/">Home</a></li>
        <li><a href="/contact">Contact</a></li>
      </ul>
    </div>
  </div>
{{end}}
```

Just like our footer and bootstrap layout, our views need to know about this template before they can use it. Open up `view.go` and add the template to the files being appended to our files slice. This is shown in [Listing 6.24](#).

⁴<http://getbootstrap.com/docs/3.3/components/#navbar>

```
$ atom views/view.go
```

Listing 6.24: Parsing `navbar.gohtml`.

```
func NewView(layout string, files ...string) *View {
    files = append(files,
        "views/layouts/footer.gohtml",
        "views/layouts/navbar.gohtml",
        "views/layouts/bootstrap.gohtml")
    //...
}
```

Now we are ready to use our navbar template. Open the bootstrap layout and add the code from [Listing 6.25](#) just after the `body` HTML tag.

```
$ atom views/layouts/bootstrap.gohtml
```

Listing 6.25: Using the navbar template.

```
<body>
    {{template "navbar"}}

    <div class="container-fluid">
        {{template "yield"}}
    </div>

    <!-- ... this is all the same -->
</body>
```

Now stop your server and restart it.

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

Bam! We have a navbar on all of our pages and it hardly took us any time at all. Neat right?

[H]

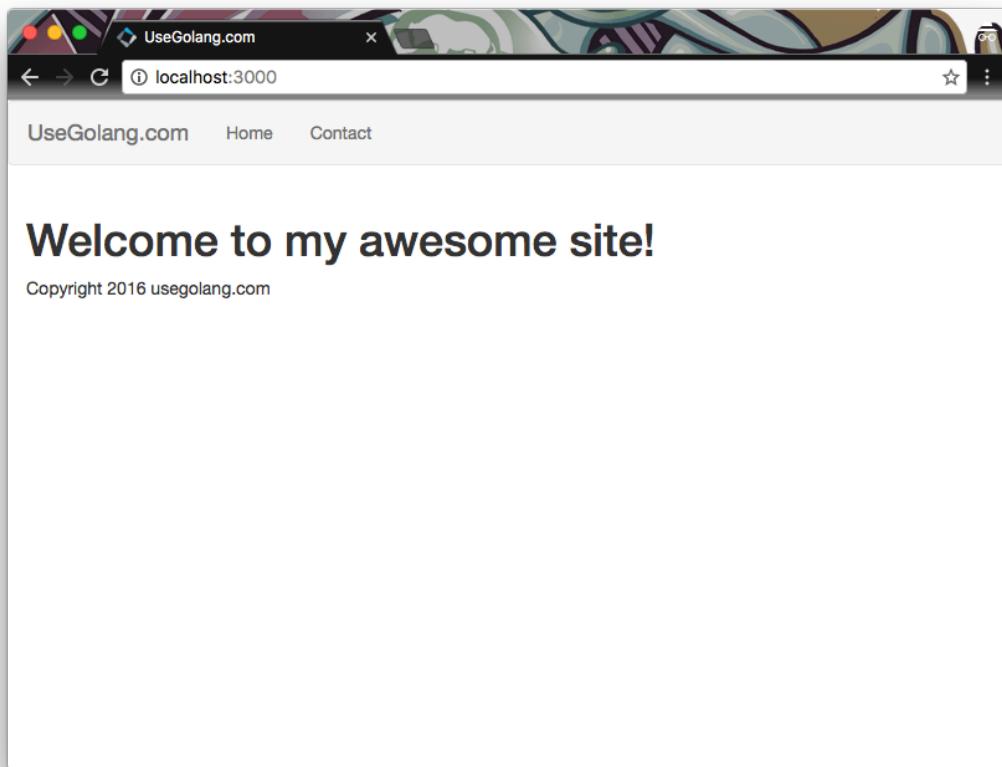


Figure 6.3: A layout with a navbar

Before moving on, take a moment to test out all of your links and make sure they work. If you have been doing exercises it might also be a good idea to add links to any custom pages you created, such as the FAQ page.

You can also check out a screenshot of the site with a navbar in [Figure 6.3](#)

Source code

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

Completed - book.usegolang.com/6.4

Changes - book.usegolang.com/6.4-diff

6.5 Cleaning up our code

Now that we have a working Bootstrap layout along with a helpful navbar it is time to take a break from adding new features and to instead spend some time cleaning up our code a bit.

Open up `views/view.go` and check out the `your` append statement. While this is only three lines now, it has grown every single time we added a new template to our application. Over time this is going to get fairly large, and as we remove templates we will need to remove them from this list as well.

Rather than spending our time maintaining this list of templates, we are instead going to dedicate some time to write a function that will simply include *all* `.gohtml` files in the layouts directory in all of our views. We may not always need them all, but this will drastically simplify the task of managing layout templates in our views and will likely prevent a few, “Why isn’t my code working?!?!” bugs.

To make this change we are going to first discuss Globbing, covering what it is and what we will be using it for.

After that we will explore how to glob files in Go and write some code to glob all of our layout templates.

Then we are going to focus on cleaning up our template execution code. Right

now our handlers need to work directly with templates and layouts, but when we are done our handlers won't need to worry themselves with how our views are rendered, they will simply need to call a single render method.

Finally we are going to take a moment to reorganize our templates, moving the footer template call out of the home and contact templates and inserting it into the bootstrap layout.

6.5.1 What is globbing?

Globbing is a way of telling a computer which files you want and which files you want to ignore. If you are familiar with the command line, **mv** is a command that lets you move a file from one location to another. For example, you could say **mv x.txt old/x.txt** and it would move the file **x.txt** in your current directory to the **old** directory.

If you wanted to move all **.txt** files to the **old** directory it would be really slow to do this one by one, but with a glob pattern you could instead type **mv *.txt old/** and it would find every file with the **.txt** extension to the **old** folder.

Note: We don't need to specify the files name in the second part of this statement because the original name will be used.

Globbing isn't the act of actually moving the files, but is instead the way that we determine which files get moved. It is the ***.txt** part of **mv *.txt old/**. It is a simple way of saying "give me all of the files that end in **.txt**"

You don't need to be a master with globbing to take advantage of it. For now there are only three big takeaways you need to know about:

1. The asterisk (*****) is used to match against any number of characters, even none, no matter what character they are.

2. Most other characters (like each character in `.txt`) will look for an exact match.
3. Globbing requires a complete match. For example, `*.txt` would not match the file `test.txt` because nothing matches the final “e” in the file. If we added a second asterisk (*) at the end of our pattern it would work, but without it we don’t have a match.

Box 6.5. Glob is not the same as a regular expression.

If you don’t have any programming experience and don’t know what a regular expressions is you are unlikely to make this mistake, but for those of you who do - Glob **is not** the same as a regular expression, so don’t expect it to work the same!

If you would like to read more about this, you can check out this [stack overflow article](#) that discusses the differences between glob-style patterns and regular expressions.

6.5.2 Using `filepath.Glob`

Go’s standard library includes the `path/filepath`⁵ package, which is incredibly useful for manipulating and dealing with file paths without having to write operating-system specific code.

In this library there is a function named `Glob`⁶, which implements the globbing functionality we talked about in the last section. It doesn’t do anything to move files or anything like that, but given a glob pattern it will return a slice of file paths (in the form of a string slice) that match the glob pattern.

⁵<https://golang.org/pkg/path/filepath/>

⁶<https://golang.org/pkg/path/filepath/#Glob>

This is incredibly useful when we want to include all of the files in a particular directory that match a specific pattern. Instead of needing to list every single **.gohtml** file in our layouts directory, we can instead say “grab every .gohtml file in that directory” and be on our merry way.

We are going to use **Glob** to simplify our **NewView** function, so we will start by opening the view source file.

```
$ atom views/view.go
```

Once you have it open, the first thing we want to do is to add the **path/filepath** package to our imports. This is shown in [Listing 6.26](#).

Listing 6.26: Importing path/filepath

```
import (
    "html/template"
    "path/filepath"
)
```

Next we are going to create a few global variables that will help us when constructing our glob pattern. This is shown in [Listing 6.27](#).

Listing 6.27: Adding glob helper variables

```
var (
    LayoutDir  string = "views/layouts/"
    TemplateExt string = ".gohtml"
)
```

LayoutDir will help us by specifying the layout directory, and TemplateExt will tell us what extension we expect all our template files to match.

Note: Technically these could both be constants, but we will leave them as variables for now in case we want to test this code with different values in the future.

Next we need to write some code that uses both of these variables, along with the **Glob** function, and returns a slice of templates to include in our view. We are going to call this function **layoutFiles**, and it is shown in Listing 6.28.

Listing 6.28: Writing the **layoutFiles()** function.

```
func layoutFiles() []string {
    files, err := filepath.Glob(LayoutDir + "*" + TemplateExt)
    if err != nil {
        panic(err)
    }
    return files
}
```

Let's take a moment to review what is happening in Listing 6.28.

First we declare our function - **layoutFiles** - and state that it will return a string slice. Inside our function we call the **Glob** function provided by the path/filepath package and pass in a string that we create by combining a few variables and a hard-coded asterisk (*) string.

Assuming we don't change our variables, the glob pattern we are using will evaluate to:

```
"views/layouts/*.gohtml"
```

This will match all files ending in **.gohtml** in the layouts directory, which is what we wanted.

If for some reason our call to **Glob** fails, we will panic with whatever error we received. Otherwise we return all of the file paths we receive from our call to **Glob**.

Box 6.6. Why so many panics?

At this point I want to address the question, “Why are we panicking so much in `view.go`?” Aren’t panics bad?

As a general rule, yes, panicking is indeed bad. Typically you should try to handle any errors that you can, or defer them to the caller of your function so that they can handle them.

If we were following that advice now, we would update our `NewView` function to return two values - a pointer to a View, and an error. So why aren’t we?

In this specific situation I know that we are only using the `NewView` function when we start up our application. That is, when our program first starts running it will create all of the views it needs, and then it will never call `NewView` again. It will instead continue to use views that were already created.

If this were to change, we would want to update our code, but since we know that our code is working this way I find it acceptable to panic when an error is encountered because even if we returned the error to our `main` function, it would need to panic immediately after receiving the error anyway because we can’t start our web application up without our views being initialized correctly. We wouldn’t have a way to render any HTML pages.

Once we have our `layoutFiles` function written we can start using it inside of our `NewView` function. Navigate to that function definition. We are going to update the first few lines where we append our layout files to the `files` slice. Instead of passing in individual files to `append`, we are going to pass in all of the files returned by the `layoutFiles` function call. The code is shown in Listing 6.29.

Listing 6.29: Using `layoutFiles()` in `NewView()`.

```
func NewView(layout string, files ...string) *View {
    files = append(files, layoutFiles()...)
    // ... this is the same
}
```

Box 6.7. Unpacking slices

In Listing 6.29 we are doing the reverse of what we have been doing so far with a variadic parameter.

Instead of packing multiple string parameters into a single string slice, we are instead telling Go that we want to unpack a slice and use each element in the slice as a variadic parameter.

Unpacking doesn't work quite the same in Go as other languages, but it can be very handy when passing a slice into a function that accepts a variadic parameter.

After putting all of this code together, we are ready to stop our program, restart it, and quickly verify that both the home and contact pages both work. Again, these shouldn't look much different than before, but moving forward we won't have to specify each layout template file in `view.go`.

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

Source code

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

Completed - book.usegolang.com/6.5.2

Changes - book.usegolang.com/6.5.2-diff

6.5.3 Simplifying view rendering

If you open up `main.go` and look at the home and contact functions you will see that we are calling `ExecuteTemplate` in both of our handler functions.

We discussed earlier how views should contain all of the logic for rendering data, but here it looks like our handler functions are taking care of this logic instead of offloading it to the view.

To fix that we are going to add a method to the `View` type that is responsible for rendering the view. Then moving forward we can simply call this method in our handlers without worrying about any rendering logic in our handlers. Open up `view.go` again and we will get started.

```
$ atom views/view.go
```

Add the code in Listing 6.30 to your file. This will add a `Render` method to the `View` type, which is why the `(v *View)` part comes before the function name.

I typically place methods like this below the type declaration in my source files, but you are welcome to do whatever you see fit.

Listing 6.30: Adding the `Render` method.

```
func (v *View) Render(w http.ResponseWriter,
    data interface{}) error {
    return v.Template.ExecuteTemplate(w, v.Layout, data)
}
```

If you don't have any auto-imports enabled, you will also need to add an import to the `net/http` package to your source.

```
import (
    "html/template"
    "net/http"
    "path/filepath"
)
```

Now we are ready to go back to `main.go` and use our new rendering method. We are going to be updating both the home and contact functions, and the resulting code is shown in Listing 6.31. I have also added a helper function - `must` - that will simplify our error handling a bit.

Listing 6.31: Using the `Render` method.

```
func home(w http.ResponseWriter, r *http.Request) {
    w.Header().Set("Content-Type", "text/html")
    must(homeView.Render(w, nil))
}

func contact(w http.ResponseWriter, r *http.Request) {
    w.Header().Set("Content-Type", "text/html")
    must(contactView.Render(w, nil))
}

// A helper function that panics on any error
func must(err error) {
    if err != nil {
        panic(err)
    }
}
```

And that's it! Go ahead and restart your server and make sure everything is working.

At this point it might be tempting to remove the `data` parameter used in the `Render` method because we aren't using it now, but we will be using it in the future so I suggest leaving it.

Source code

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

Completed - book.usegolang.com/6.5.3

Changes - book.usegolang.com/6.5.3-diff

6.5.4 Moving our footer to the layout

The last thing we need to do in our cleanup phase is remove the footer template calls from the home and contact templates.

While this code isn't incorrect, the footer template really belongs in the bootstrap layout template that we created earlier, and moving it is pretty simple.

Open up both the home and contact templates and remove the line that reads:

```
 {{template "footer"}}
```

Listing 6.32: `home.gohtml` without the footer.

```
 {{define "yield"}}
    <h1>Welcome to my awesome site!</h1>
 {{end}}
```

Listing 6.33: `contact.gohtml` without the footer.

```
 {{define "yield"}}
    To get in touch, please send an email to
    <a href="mailto:support@lenslocked.com">
        support@lenslocked.com
    </a>.
 {{end}}
```

If we want to keep the footer we need to add it back into the bootstrap layout template. Or if you want to delete it, you can do that too. I know it is pretty ugly right now.

```
$ atom views/layouts/bootstrap.gohtml
```

Listing 6.34: `bootstrap.gohtml` with the footer.

```
<!-- ... -->
<div class="container-fluid">
  {{template "yield" .}}

  {{template "footer"}}
</div>
<!-- ... -->
```

And that is it! Restart your application and verify everything is still working. If it is you are officially done coding for chapter 6. Well, at least until we get to the exercises.

Source code

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

Completed - book.usegolang.com/6.5.4

Changes - book.usegolang.com/6.5.4-diff

6.6 Exercises

Assuming everything went well, you should now have a pretty solid framework for creating new views. In fact, what we have now is pretty close to what I have

used in several applications of my own.

If anything up until now seems a little fuzzy or unclear, remember that we will be continuing to add new views to our web application throughout the book. Each time we do this, it will help reinforce how we use our view type and give you a chance to experiment with it a bit. That means that even if you don't understand all this code perfectly now, you will have plenty of opportunities to reexamine the code we just wrote and get a better feel for how it works.

In the meantime, here are a few exercises to help you verify that you understood everything we did in this chapter.

6.6.1 Ex1 - Create an FAQ page with the Bootstrap layout

You may already have an FAQ page from earlier exercises. If so, you can use that page.

Your goal here is to create an FAQ page that also uses the bootstrap layout like we did with our home and contact pages. It should use the **view** type just like those pages do as well.

6.6.2 Ex2 - Update the navbar to link to the FAQ page

Once you have a working FAQ page, update your navigation bar to include a link to the FAQ page.

While you are at it, explore the navbar docs⁷ and experiment with different ways of changing your navbar using Bootstrap.

⁷<http://getbootstrap.com/docs/3.3/components/#navbar>

6.6.3 Ex3 - Create a new layout

We have our Bootstrap layout, but what if we wanted a new one for a specific page?

Create a new layout, possibly with a different color background, navbar, or something else that is different, and then update the FAQ page to use this new layout.

This should give you a sense for how easy it is to swap out our layout without breaking most of our web pages.

Chapter 7

Creating a sign up page

In this chapter we are going to focus on creating a page where users can sign up for a new account. While this sounds simple at first, we will actually be covering a lot of material that isn't directly related to the sign up page, but is useful for organizing our code in a maintainable way.

We will be learning about REST, and how it will affect our overall code design. We will then create our first controller - the users controller - and add our signup page to this controller.

After that we will discuss how to process forms with Go, and how to handle different HTTP methods with the gorilla/mux router.

Finally we will spend some time cleaning up our existing code, moving each view we have into a controller so that we are following a proper MVC pattern.

7.1 Add a sign up page with a form

We are going to start by creating a view for our sign up page. To do this, we are going to utilize the Bootstrap docs pretty heavily, as we don't want to spend

too much time designing our sign up page when it doesn't even work yet.

Our sign up page is going to be created in two phases.

First we are going to grab an HTML form example from the Bootstrap docs and then customize it with the fields and text that we want in our form.

Once we have the correct data in our form, we are going to check out Bootstrap panels and use one to wrap our form and make it look slightly better. It still won't look amazing, but it will definitely look better than an unstyled HTML form.

Finally, we are going to add a link to our sign up page to our navbar. This is something we have done in past exercises, so it should look familiar with one caveat - we will be placing our sign up link on the right side of the navbar.

Box 7.1. You may be able to skim these subsections.

[Section 7.1.1](#) and [Section 7.1.2](#) are mostly focused on HTML changes, so if you are comfortable with HTML and templates you can skim both sections and then grab the source code at the end before proceeding.

7.1.1 Creating a Bootstrap sign up form

First let's head over to the [Bootstrap CSS Docs](#) and scroll down to the forms section of this page. This is where Bootstrap gives us demos for a few different ways to use forms, and shows code samples for each. If you ever need to create a new form and you are using Bootstrap (we are!) this is a great place to start as it could save you a good bit of work trying to figure it out on your own.

In our case, the first example - [the basic example](#) - is exactly what we want, minus the file input and checkbox sections. Awesome!

First we will create the signup template file

```
$ atom views/signup.gohtml
```

Next we will copy the code from the bootstrap docs and paste it into the file. Once it is pasted in there, you should go ahead and wrap all of that code with a **define** and **end** block. The resulting code should match Listing 7.1.

Listing 7.1: Initial sign up view.

```
 {{define "yield"}}
<form>
  <div class="form-group">
    <label for="exampleInputEmail1">Email address</label>
    <input type="email" class="form-control"
      id="exampleInputEmail1" placeholder="Email">
  </div>
  <div class="form-group">
    <label for="exampleInputPassword1">Password</label>
    <input type="password" class="form-control"
      id="exampleInputPassword1" placeholder="Password">
  </div>
  <div class="form-group">
    <label for="exampleInputFile">File input</label>
    <input type="file" id="exampleInputFile">
    <p class="help-block">
      Example block-level help text here.
    </p>
  </div>
  <div class="checkbox">
    <label>
      <input type="checkbox"> Check me out
    </label>
  </div>
  <button type="submit" class="btn btn-default">
    Submit
  </button>
</form>
{{end}}
```

Now we need to trim out the parts that we don't need. We won't be using a file input or a checkbox, so we can start by removing those two snippets. Both are near the end of the form, and are shown below.

```
<div class="form-group">
  <label for="exampleInputFile">File input</label>
  <input type="file" id="exampleInputFile">
  <p class="help-block">
    Example block-level help text here.
  </p>
</div>
```

```
<div class="checkbox">
  <label>
    <input type="checkbox" checked="checked" value="option1"> Check me out
  </label>
</div>
```

Next we want to change our email input to match what our server is going to expect. To do this we need to do roughly three things:

1. Change the **id** of our input to “email” instead of “exampleInputEmail1”
2. Update the email label’s **for** field to match our new id
3. Add the **name** attribute to our email input tag and assign it the value “email”

[Listing 7.2](#) shows all three of these changes.

Listing 7.2: Updated email form group.

```
<div class="form-group">
  <label for="email">Email address</label>
  <input type="email" name="email" class="form-control"
    id="email" placeholder="Email">
</div>
```

Box 7.2. The `name` attribute.

When you submit a form to a server, it receives the data in a format very similar to a map. The name of each input field is used as the key in the map, and whatever the user typed into the input field is used as the value.

By setting the `name` attribute to “email”, we are stating that we want the key for this input field to be “email” when it is submitted to our server.

Now we are going to repeat the same process we just went through for the password input, but instead of using the name “email” we will be using the name “password”. This is shown in Listing 7.3.

Listing 7.3: Updated password form group.

```
<div class="form-group">
  <label for="password">Password</label>
  <input type="password" name="password"
    class="form-control" id="password"
    placeholder="Password">
</div>
```

The last thing we are going to change in our form is the text on the submit button. Having a button that reads, “Submit” isn’t awfully clear to an end user. Instead, we will change the text to read “Sign Up”, since that is what the user is doing. The code for this is shown in Listing 7.4.

Listing 7.4: Updated button text.

```
<button type="submit" class="btn btn-default">
  Sign Up
</button>
```

Now let's go ahead and add some code to `main.go` that will allow us to view our new sign up form. To do this, we need to create a new view like we did for the home and contact pages, and then create a handler function that we will provide to our router.

```
$ atom main.go
```

Add the global variable `signupView` near our other view variables. While we don't have to, we can go ahead and move these all into a variable group while we are at it.

```
var (
    homeView    *views.View
    contactView *views.View
    signupView   *views.View
)
```

Now that we have a view we need to initialize it. We will do this inside of the `main` function like before, and once again we will use the `NewView` function that is provided by the `views` package.

```
signupView = views.NewView("bootstrap",
    "views/signup.gohtml")
```

We also need to create a handler function that we can pass to our router. We will call this `signup`, and it needs to accept both a `ResponseWriter` and a `Request` argument like our existing handler functions. Once we are inside the function all it needs to do is call the `Render` method on our `signupView`. You can also set the `Content-Type` header if you want, but it isn't required.

```
func signup(w http.ResponseWriter, r *http.Request) {
    w.Header().Set("Content-Type", "text/html")
    must(signupView.Render(w, nil))
}
```

Finally, we plug the **signup** function we just created into our router at the path **/signup**.

```
r.HandleFunc("/signup", signup)
```

Putting this all together, your **main.go** source file should look similar to Listing 7.5.

Listing 7.5: Updated main.go.

```
// ... this is all the same

var (
    homeView    *views.View
    contactView *views.View
    signupView  *views.View
)

// ... home, contact, and must functions stay the same

func signup(w http.ResponseWriter, r *http.Request) {
    w.Header().Set("Content-Type", "text/html")
    must(signupView.Render(w, nil))
}

func main() {
    homeView = views.NewView("bootstrap",
        "views/home.gohtml")
    contactView = views.NewView("bootstrap",
        "views/contact.gohtml")
    signupView = views.NewView("bootstrap",
        "views/signup.gohtml")

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

[H]

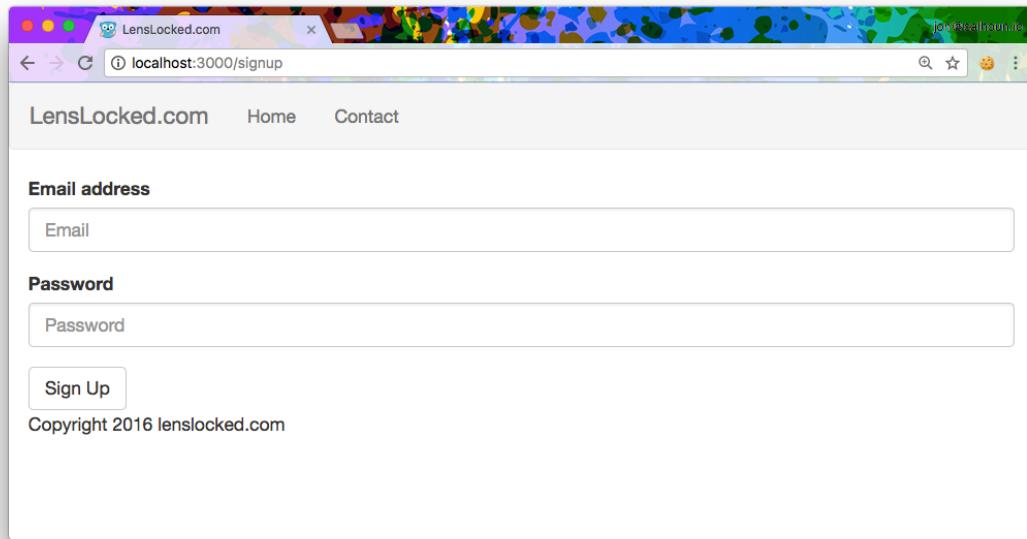


Figure 7.1: First version of the signup form

Restart your server and check out our new signup page at localhost:3000/signup. It should look similar to [Figure 7.1](#).

Source code

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

Completed - book.usegolang.com/7.1.1

Changes - book.usegolang.com/7.1.1-diff

7.1.2 Wrapping our form in a panel

With our form created and viewable, we are now ready to try to utilize Bootstrap to make our form look a little better. We will do this by using two features of bootstrap.

1. Panels¹
2. The grid system²

Panels are a part of bootstrap that help you wrap content in a container making it obvious that everything in that container is related. We will use a panel to make it clear which fields are used in our sign up form.

The grid system allows us to define rows and columns, effectively giving us a way to layout our entire page. Where grid really shines is when transitioning from a web to a mobile view, as it will allow us to write code once and have it look reasonably decent on both a computer and a mobile phone.

We will be using grid to define how large we want our signup form to be depending on the user's screen size, and then we will place our form inside of the grid elements we define.

Note: We won't be spending a lot of time on HTML, CSS, and styling, but if you want to spend more time on your own customizing the look of your web application you can. As long as the form inputs retain the same name fields, it shouldn't affect how our application works.

Getting started, we are going to alter our signup template.

¹<http://getbootstrap.com/docs/3.3/components/#panels>

²<http://getbootstrap.com/docs/3.3/css/#grid>

```
$ atom views/signup.gohtml
```

While we are currently defining our template as “yield”, what we have created so far is actually just the signup form. We are going to rename our form template to **signupForm** and then create a new “yield” template.

Inside the new “yield” template we are going to create a panel using an example from the bootstrap panel docs. We will use the second example under the *Panel with heading*³ section - the panel with a title.

The changes we want are shown in [Listing 7.6](#). Take a moment to check out the Bootstrap docs and get a feel for where the panel HTML came from.

Listing 7.6: Panel yield template.

```
{{define "yield"}}
<div class="panel panel-default">
  <div class="panel-heading">
    <h3 class="panel-title">Panel title</h3>
  </div>
  <div class="panel-body">
    Panel content
  </div>
</div>
{{end}}

{{define "signupForm"}}
<form>
  <!-- ... this is the form we created already -->
</form>
{{end}}
```

We have a basic outline to work with, so let’s go ahead and tweak it to meet our needs.

First, replace the contents of the **h3** tag with the title “Sign Up Now!”

³<http://getbootstrap.com/docs/3.3/components/#panels-heading>

```
<h3 class="panel-title">Sign Up Now!</h3>
```

Next search for the line “Panel content” inside of a **div** tag. We are going to replace this with our signupForm template.

```
<div class="panel-body">
  {{template "signupForm"}}
</div>
```

Note: We are not passing any data to the signupForm template right now because there isn't a period (.) argument after the “signupForm” portion of the code.

Lastly, we are going to update a few classes to demonstrate how colors work in bootstrap.

Bootstrap comes with a few base contextual colors that can be used on nearly any bootstrap element. For example, you could customize the look of your buttons⁴ using **btn-primary** and **btn-danger** classes in your HTML.

Rather than naming each color, they are instead named by their context. This allows developers to customize each color depending on their app without needing to change all of the HTML. It also helps give your app a more consistent look and feel if all the “primary” buttons are the same color.

Bootstrap class names, along with their default colors, are listed in the table below.

CSS Class / Name	Default Color
default	white/gray
primary	blue
success	green
info	light blue

⁴<http://getbootstrap.com/docs/3.3/css/#buttons-options>

warning	yellow
danger	red

To change the color of our signup form panel, we are going to first look for the class **panel-default** and then change it to the class/color we want to use. Let's try **panel-primary** instead, as filling out the sign up form is the primary action to take on this page. Listing 7.7 shows the updated code.

Listing 7.7: yield with a finished panel.

```
<div class="panel panel-primary">
  <div class="panel-heading">
    <h3 class="panel-title">Sign Up Now!</h3>
  </div>
  <div class="panel-body">
    {{template "signupForm"}}
  </div>
</div>
```

While we are at it, we are also going to update the button to be **btn-primary** instead of the default button.

```
<button type="submit" class="btn btn-primary">
```

Now that our form is wrapped in a primary panel, we are going to utilize the grid system to center our panel and reduce its width on medium sized or larger screens.

Box 7.3. Bootstrap media sizes

You can read more about bootstrap sizes in the bootstrap docs below, but as a general rule you can assume that anyone on screen size of **md** or larger is viewing your site on a computer or a decent sized tablet.

- <http://getbootstrap.com/docs/3.3/css/#grid-media-queries>

The grid system requires us to first create a row, and then inside of that row we are going to place columns. There are 12 columns in the grid system, so we are going to say that our signup form should take up 4 of those columns on a medium or larger sized screen.

Our columns start on the left side of the row, so we are also going to add an offset of 4 columns so that our form is centered. We don't account for any offset on the right side.

[Listing 7.8](#) shows the updated “yield” template, including the grid system changes and all the other changes we have made in this section.

Listing 7.8: Final sign up template.

```
 {{define "yield"}}
<div class="row">
  <div class="col-md-4 col-md-offset-4">
    <div class="panel panel-primary">
      <div class="panel-heading">
        <h3 class="panel-title">Sign Up Now!</h3>
      </div>
      <div class="panel-body">
        {{template "signupForm"}}
      </div>
    </div>
  </div>
{{end}}
```

```
 {{define "signupForm"}}
<form>
  <div class="form-group">
    <label for="email">Email address</label>
    <input type="email" name="email" class="form-control"
           id="email" placeholder="Email">
  </div>
  <div class="form-group">
    <label for="password">Password</label>
```

```
<input type="password" name="password"
       class="form-control" id="password"
       placeholder="Password">
</div>
<button type="submit" class="btn btn-primary">
    Sign Up
</button>
</form>
{{end}}
```

Source code

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

Completed - book.usegolang.com/7.1.2

Changes - book.usegolang.com/7.1.2-diff

7.1.3 Adding the sign up link to our navbar

We have one last change to make to finalize the addition of our sign up view. We need to add a link to this page to our navbar! Ideally we would like to add this to the right side of our navbar, since this is where most people expect to find a sign up link. To do this we are going to check out the Bootstrap docs one more time.

Open up the bootstrap default navbar docs:

<http://getbootstrap.com/docs/3.3/components/#navbar-default>

Note: If you are offline you can just get the code we use below

What we are looking at here is the **ul** HTML tag with the class **navbar-right**. This shows us how to create the right portion of the navbar example.

We won't be using everything in the example, and we will want to update the link to refer to our sign up page. After making each of these changes we should end up with code similar to that in Listing 7.9.

Listing 7.9: Our navbar-right

```
<ul class="nav navbar-nav navbar-right">
  <li><a href="/signup">Sign Up</a></li>
</ul>
```

We want to add this code to our navbar, so open up the navbar template.

```
$ atom views/layouts/navbar.gohtml
```

Next look for the existing **ul** tag that has the class **navbar-nav**. Our new code is going to go right after this **ul** tag. When you are done your code should match Listing 7.10.

Listing 7.10: Adding the sign up link to the navbar.

```
{{define "navbar"}}
<nav class="navbar navbar-default">
  <div class="container-fluid">
    <div class="navbar-header">
      <button type="button" class="navbar-toggle collapsed"
        data-toggle="collapse" data-target="#navbar"
        aria-expanded="false" aria-controls="navbar">
        <span class="sr-only">Toggle navigation</span>
        <span class="icon-bar"></span>
        <span class="icon-bar"></span>
        <span class="icon-bar"></span>
      </button>
      <a class="navbar-brand" href="/">LensLocked.com</a>
    </div>
    <div id="navbar" class="navbar-collapse collapse">
      <ul class="nav navbar-nav">
        <li><a href="/">Home</a></li>
        <li><a href="/contact">Contact</a></li>
      </ul>
      <ul class="nav navbar-nav navbar-right">
```

[H]

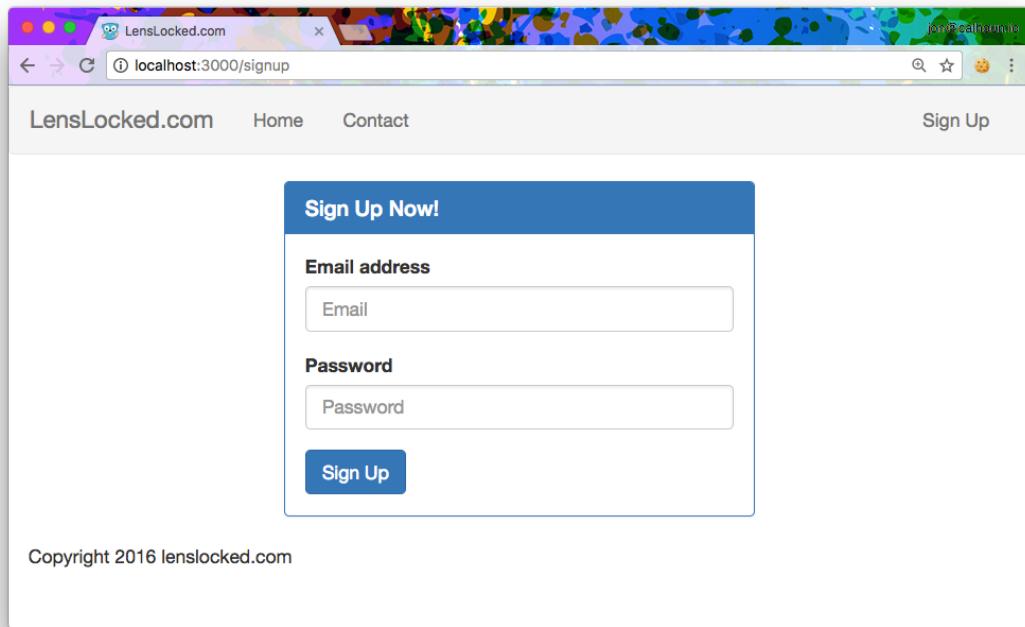


Figure 7.2: Finished sign up form

```
<li><a href="/signup">Sign Up</a></li>
</ul>
</div>
</div>
</nav>
{{end}}
```

Go ahead and restart your server to test all of our changes. We should now have a sign up page, links to it in the navbar, and the page should have a blue panel around it. A screenshot is shown in Figure 7.2.

You can also try resizing your browser window to see how bootstrap automatically changes the look and size of your panel based on the browser window size. Neat, right?

Source code

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

Completed - book.usegolang.com/7.1.3

Changes - book.usegolang.com/7.1.3-diff

7.2 An intro to REST

REST is a subject that is often debated and could likely take up an entire book on its own. Obviously we don't want to spend weeks discussing what is and isn't RESTful, but we do need to take a few moments to get an overview of what REST is and how it will affect our application design.

REST is an architectural style for developing software and networked systems, and was used in the design of most of the World Wide Web⁵. It is also commonly used as a guide for designing web applications and APIs, which is how we will be using it.

REST is pretty abstract, so if someone ever tells you that something isn't RESTful, take it with a grain of salt. There is no golden standard of what is or isn't RESTful, and (to my knowledge) there aren't any real world examples of what the ideal RESTful web application would look like. Instead we lowly developers are left to interpret it the best we can.

So what does that mean for us? Well, it mostly means that we will be trying to model most of the pieces of our web application around *resources*. Resources are things like a user, a gallery, or an image. Things that can be created, read, updated, and deleted within the context of our web application. Some people even take this a step further and also consider things like a session or rela-

⁵https://en.wikipedia.org/wiki/World_Wide_Web

Table 7.2: HTTP request methods and their meaning.

HTTP Method	What it signifies
GET	Reading a resource
POST	Creating a resource
PUT	Updating a resource
PATCH	Updating a resource
DELETE	Deleting a resource

tionship (eg following someone on twitter) as a resource that needs to be created or deleted within the context of our application.

While I don't necessarily disagree with the idea of modeling a session as a resource, I want to stress that you shouldn't get caught up in trying to find the perfect resource representation of everything in your application. Instead, use it as a guiding principle as you create your models, controllers, and views, but don't bend over backwards to make it happen, especially if it means your code will be harder to understand or maintain.

7.2.1 How REST affects our code

There are basically four (okay, we will look at five) HTTP request methods that can be used when requesting a web page. These are used to help dictate your intent when you make a web request, and they are shown in [Table 7.2](#).

Box 7.4. The difference between **PUT** and **PATCH**.

Both **PUT** and **PATCH** are used to represent updating a resource, but they both do it in a fundamentally different way. **PUT** generally is expected to accept an entirely new representation of an object, even if some of the fields didn't change, while

PATCH was proposed as a way to update resources and also signify that you won't be passing all of the fields for the resource, but instead will only be providing the updated fields.

For all practical purposes you mostly just need to remember that these are both used to update resources.

The HTTP methods listed in [Table 7.2](#) are used in conjunction with paths in our web application to ultimately shape our code, and the paths are also a strong indicator as to what controllers we should be creating.

For example, we will eventually create a gallery resource that will have paths like the ones shown in [Table 7.4](#)

We will also create models, controllers, and organize our views according to this resource. That means that our application will eventually have a controller for the galleries resource, a model for the galleries resource, and views that deal with the gallery resource will be organized in a galleries folder and constructed when we construct the galleries controller.

7.3 Creating our first controller

Over the course of this book we are going to create several controllers, and within each controller we are going to create handler functions like we have already.

Box 7.5. Actions are synonymous to handler functions

In this book I often refer to functions that handle web requests as handler functions, because this is quite literally what they are. In the MVC world, the functions or

Table 7.4: Example gallery paths.

HTTP Method and Path	What It Is Used For
GET /galleries/new	This path is used to retrieve a form or some other page that will allow you to create a new gallery. It would be akin to asking for a driver's license application - you don't create the gallery here, but you get the page that you are going to submit to create one.
POST /galleries	Whenever you use the HTTP method 'POST' it signifies that you want to create something or perform some action. When you POST to the bare resource path, it signifies that you want to create that resource, and you are typically expected to provide the information required to create that resource along with the web request. In this case, we would be saying that we want to create a gallery and we would need to include the information used to create a gallery. The data will often come from an HTML form, but it could also be JSON or any other format. Going back to the driver's license metaphor, this would be like dropping off your application at the front desk and having them hand you a license in return.
GET /galleries/:id	This path is used to read a single gallery. The ':id' section is replaced with something we used to determine which gallery is being read, often a numerical id like '313'. In our driver's license example, this would be like looking up a driver's license by the unique number on each ID card.
PUT /galleries/:id	This path works similar to the previous one, but because we are using the 'PUT' HTTP method we are stating that we want to update the gallery, and we are expected to provide the updated data along with our web request. With a driver's license this would be similar to submitting an application to change your address.
DELETE /galleries/:id	This path works similarly to the previous one - we use the ID to specify which gallery we are referencing, but rather than updating it we are stating that we want to delete the resource.

methods that handle requests for a specific page are often referred to as “actions”. This terminology is especially popular in Rails and ASP.NET.

I have never found a well defined rule for what is or isn’t an action, but generally speaking an action is a function that handles a web request for a specific path. For example, when we create the sign up page, it will be mapped to the `New()` method on the `Users` type. In this case, the `New()` method is an action used to render the sign up page.

Nearly every handler function we write from this point on in the book will be an action, so you might see me interchange the terms “handler function” and “action” along the way, but rest assured that I am referring to the same thing.

The first controller we are going to create is going to be the users controller, which will contain all of the handler functions for pages that interact with the user resource. For example, we might have an edit page for users where they can update their user information, and when a user visits this page the request would be processed by handlers inside of our users controller. Similarly, when they submit a form with the updated information this update would be handled by the users controller.

At their very core, the handler functions inside of a controller aren’t really any different than the functions we have been writing so far. The functions will still take in a `ResponseWriter` and `Request` as their arguments, and they will use both of these arguments the same way we have been.

The only real difference is that instead of being a function declared in our `main.go` file, they will instead be methods attached to a `Users` type that we will be declaring inside of a `controllers` package.

While this might not appear to have any positive effects on our code at first, it will eventually have a larger impact on our code. The most notable difference is that we can share views and database connections with our controllers and

then have access to them in our actions, but there are other long term benefits such as keeping code organized and easier to test.

7.3.1 Create the users controller

Our controllers are going to be in the **controllers** package, so we will start by creating the directory for this package and then we will create the users controller source file.

```
# You are likely in this dir already
$ cd $GOPATH/src/lenslocked.com

$ mkdir controllers
$ atom controllers/users.go
```

We are going to be creating a new type for our users controller. How you name this is up to you, but I prefer to use a name like **Users** because it will often be referred to in other code as **controllers.Users**, making it clear that this is the users controller.

We don't really know what data our users controller is going to need just yet, so we will start out with an empty struct as the type. The code is shown in Listing 7.11.

Listing 7.11: Creating the **Users** controller type

```
package controllers

type Users struct {}
```

That's it. We now have a Users controller available to use in our code, and we are ready to start adding views and any other relevant fields to it.

Source code

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

Completed - book.usegolang.com/7.3.1

Changes - book.usegolang.com/7.3.1-diff

7.3.2 Moving the sign up page code

We already have most of the sign up page created, but in this section we are going to migrate that page from our `main.go` source file into our users controller. It will become part of the users controller because submitting the form ultimately creates a new user account, and anything creating or altering users should typically be part of the users controller.

Technically, we could keep the sign up template where it is. Our users controller could just load the template file from its current location and it wouldn't break our application. Despite the fact that it wouldn't break anything, we are going to be introducing a lot of new pages over time and I have found it is best to introduce an organization pattern for your templates early on. The way I prefer to organize views is to have them stored in a directory with the same name as the controller they relate to, and to name them something similar to the action that uses the view created with the template.

What this means in practice is that I like to store all user related templates in the directory `views/users`, and I like to name each template based on the action that is being rendered. For example, we will be rendering the signup page with the "new" user action, so we will be moving the signup template into the users directory with a new filename of `new.gohtml`.

```
# Create the users template dir
$ mkdir views/users

# Move and rename the signup template
$ mv views/signup.gohtml views/users/new.gohtml
```

Going forward we are going to name all of our templates based on the action that they relate to in our code so that when a developer is working on the **Edit** action they will know that the corresponding template is stored in the **edit.gohtml** template file.

After moving our template into its final location, we need to tell our users controller about the template. Rather than telling it about the template itself, we are going to store a view on our users controller that represents this template. That way the controller doesn't need to handle executing a template, but can instead just call our view's **Render** method like we are doing in the **signup** function in now.

We will start by adding a field to the **Users** type that will store the new user view. You will also need to import the views package while doing this. The new code is shown in Listing 7.12.

```
$ atom controllers/users.go
```

Listing 7.12: Add the **NewView** field to **Users**

```
import "lenslocked.com/views"

type Users struct {
    NewView *views.View
}
```

This view doesn't do us much good unless it gets initialized to something, so we need to write some code to create our view using the **NewView** function and the new user template.

Rather than doing this in `main.go` like we have been, we are instead going to provide a `NewUsers` function in our controllers that will handle setting up all the views our users controller will need. This will make it easier to reuse our controllers in the future.

[Listing 7.13](#) shows the updated code for this. In the new code we first import the views package, then we begin to write the `NewUsers` function that will be used to construct and return a `Users` object.

Inside the `NewUsers` function we call the `NewView` function provided by the views package and pass in the layout and template that we want to use, and then the newly created view is assigned to the `NewView` field of the users controller we are constructing. Finally, the users controller is returned.

Listing 7.13: Add the `NewView` to `Users`

```
package controllers

import "lenslocked.com/views"

func NewUsers() *Users {
    return &Users{
        NewView: views.NewView("bootstrap", "views/users/new.gohtml"),
    }
}

type Users struct {
    NewView *views.View
}
```

Finally, we are going to create our handler (aka action) that will handle web requests when a user visits the sign up page. Our method will be named `New` because it is used to create a new user. This is also why we named the template `new.gohtml`.

This is going to be very similar to the `signup` function we created earlier, but it is going to be a method associated with the `Users` type. This is important, because it means our `New` method will also have access to the `NewView` field that we just created.

The code for our **New** action is shown in Listing 7.14.

Listing 7.14: The new user action

```
import (
    "net/http"

    "lenslocked.com/views"
)

// ... this is all the same

// New is used to render the form where a user can
// create a new user account.
//
// GET /signup
func (u *Users) New(w http.ResponseWriter, r *http.Request) {
    if err := u.NewView.Render(w, nil); err != nil {
        panic(err)
    }
}
```

Inside of our **New** method we are using **u** to access the **Users** controller. This enables us to reference the **NewView** field, and then ultimately call that view's **Render** method just like we did in our original **signup** function.

After calling the render method we check for any errors and panic if any occur. We will eventually update our code to handle that panic a little more gracefully, but for now we are going to leave it because we don't have a better system for handling errors in place.

Box 7.6. Why do we create a **NewView field?**

At first creating a **NewView** field on the users controller might seem pointless. Why not just use a global variable like we were before? Wouldn't that make our code simpler?

While global variables often appear simpler at first, they can easily lead to bugs in the long term because *any code* in the controllers package could (intentionally or not) end up altering the variable. This is what is known as a *side effect*⁶, and it can be very hard to track down and test against.

By attaching the new user view to our controller, we avoid needing to use global variables and have a local copy of the view attached to our controller that can be edited without affecting other copies of the users controller.

Source code

WARNING: Your code will not compile at this point because we moved the signup template. We will fix that in the next section.

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

Completed - book.usegolang.com/7.3.2

Changes - book.usegolang.com/7.3.2-diff

7.3.3 Connecting our router and the users controller together

Now that we have a users controller available, we want to use it instead of the **signup** function that we previously created. First let's open up **main.go** and remove some code that we will no longer be needing.

```
$ atom main.go
```

Now remove the following lines of code.

```
// Delete this from within the var group
signupView *views.View
```

```
// Delete this from within main()
signupView = views.NewView("bootstrap",
    "views/signup.gohtml")
```

After removing those two, we also need to delete the entire `signup` function. As I said before, we will be using the users controller to render the sign up page moving forward.

```
// Delete this function entirely
func signup(w http.ResponseWriter, r *http.Request) {
    w.Header().Set("Content-Type", "text/html")
    must(signupView.Render(w, nil))
}
```

Next we need to import the controllers package that we created so that we can construct a users controller. Once we have the package imported, we will go ahead and create a users controller and assign it to a variable inside of our `main` function. This code is shown in Listing 7.15, but again your code won't compile just yet.

Listing 7.15: Instantiating a users controller

```
import (
    "net/http"

    "lenslocked.com/controllers"
    "lenslocked.com/views"

    "github.com/gorilla/mux"
)

// ... this is all the same

func main() {
```

```
homeView = views.NewView("bootstrap", "views/home.gohtml")
contactView = views.NewView("bootstrap", "views/contact.gohtml")
usersC := controllers.NewUsers()

// ... this is all the same
}
```

Finally, we are ready to use the **New** method on the users controller as our handler function passed into our router. Find the line that reads:

```
r.HandleFunc("/signup", signup)
```

It should be located inside of the **main** function. We are going to update the last part of this, the **signup** argument, and we are instead going to pass in **usersC.New** which is the new user handler we created. The new code is shown in Listing 7.16.

Listing 7.16: Updating the router

```
r.HandleFunc("/signup", usersC.New)
```

It is important that you **do not** add parenthesis after “New”. Your editor may try to autocomplete and add these for you thinking that you are calling this method, but that is not what we want. Rather than calling the **New** method, we are instead passing this method itself as an argument to the **HandleFunc** function call.

At this point your program should start compiling again and will work much like it did before we introduced the users controller.

By changing the last argument in the call to **HandleFunc** we have instructed our router that we now want it to use the **New** method we defined to handle any web requests for the page **/signup**. Because this method matches the definition of a HandlerFunc our program will accept it happily. It doesn’t matter to

our router that this is a method attached to the users controller. All that matters is that the **New** method will accept two arguments of the type ResponseWriter and Request.

While it may not matter to our router that this is a method, it is very important to us because it means that our **New** method has access to any data stored on the **Users** type, such as the **NewView** we setup earlier.

We are now ready to stop and restart your program if you haven't already. Make sure everything is compiling correctly and that all of our pages are working. We still won't be able to submit our sign up form, but the form should render correctly again.

Source code

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

Completed - book.usegolang.com/7.3.3

Changes - book.usegolang.com/7.3.3-diff

7.4 Processing the sign up form

At this point we have successfully created our first controller. Moving forward, when we want to add a new page the first thing we are going to ask ourselves is "What controller should this be a part of?".

To answer that question we generally just have to think about what resource we are working with. For example, we are about to start writing the handler to process the sign up form. When you submit the sign up form you are attempting to create a new user account, so this should likely be a part of the users controller.

Naming our handler is usually determined in a similar fashion. When we submit the sign up form we want to create a new user account, so we should probably name this action “Create”.

Box 7.7. The most common controller actions

While a controller can have a large number of actions associated with it, the most common actions are generally the **CRUD** actions.

CRUD stands for create, read, update, and delete, and refers to performing these operations on a resource, like a user. While this isn’t always true, most resources will be things that your users can create, read, update, and delete. As a result, you will need actions on your controller to allow a user to perform these operations.

While there isn’t a hard rule for what the actions on a controller should be for each of these operations, the ones that I see most common are show, edit, update, new, create, and delete.

The edit and new actions are used to render the respective forms (GET requests), while update and create are used to process the form submission (POST or PUT requests).

7.4.1 Stubbing the create user action

We know we want to name our action `create`, and that it will be on the users controller, so let’s go ahead and create a stub for that now. There won’t be a ton of code in this section, but we are just going to stub things out so we can go back later and fill it with some real code.

First, open up the users controller.

```
$ atom controllers/users.go
```

Once your source code is opened up, add a temporary **Create** method to the **Users** type. We will also print out a message stating that this is temporary, so we will need to import the **fmt** package. The code for this is shown in Listing 7.17.

Listing 7.17: The users controller's create action

```
// Update our imports
import (
    "fmt"
    "net/http"

    "lenslocked.com/views"
)

// ... existing code stays the same

// Create is used to process the signup form when a user
// tries to create a new user account.
//
// POST /signup
func (u *Users) Create(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintln(w, "This is a temporary response.")
}
```

Next we are going to plug this into our router, but first we need to learn about how we can route web requests based on HTTP methods using gorilla/mux so that we can differentiate between a user requesting the sign up form and a user submitting the sign up form. After that we will plug this action into our router and then start parsing the form data that the user submits.

Source code

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

Completed - book.usegolang.com/7.4.1

Changes - book.usegolang.com/7.4.1-diff

7.4.2 HTTP request methods and gorilla/mux

We talked about HTTP request methods like **GET** and **POST** earlier in this chapter, but up until this point we haven't had a reason to actually use them in our code. All of our paths are unique, and none of our pages have created, updated, or destroyed data, so it really hasn't been a major concern.

Now we want to allow users to create user accounts, so we need to update our code to reflect this. Specifically, we want to map any **POST** requests to the **/signup** path to the **Create** method because this means that a form has been submitted to create a new user. We also want a **GET** to the **/signup** path to show the sign up form like it currently does.

To do this we need to update our router and inform it of which HTTP methods to accept for each handler function. Rather than just telling you what that method is, I want you to try to find it on your own.

Head over to the Gorilla mux docs and look at the [api](#) that lists all of the methods and functions made available by the package.

Does anything there look like it would help us? Does anything mention HTTP methods or verbs?

It looks like there are two different **Methods** definitions that might do what we want, but which is the right one?

To answer this question, we first need to look at our existing code. In **main.go** we have the following code.

```
r := mux.NewRouter()
r.HandleFunc("/", home)
```

```
r.HandleFunc("/contact", contact)
r.HandleFunc("/signup", usersC.New)
```

If you look at the docs for `HandleFunc` you will notice that this method returns a `*mux.Route`. That means we could access the return value by doing something like...

```
route := r.HandleFunc("/signup", usersC.New)
// or we could chain our method calls
r.HandleFunc("/signup", usersC.New).SomeRouteMethod()
```

Knowing that we have a `Route` to work with, we should check out the `Methods` method on that type first.

Reading the docs we learn that this method accepts a variadic parameter, where each string provided represents a single HTTP method that our route will handle. To test this, let's update all of our existing routes to only accept the `GET` method and verify that things are still working.

Open up `main.go` if you haven't already.

```
$ atom main.go
```

Then update the router code in the `main` function to match the code in Listing 7.18.

Listing 7.18: Routing with methods

```
r := mux.NewRouter()
r.HandleFunc("/", home).Methods("GET")
r.HandleFunc("/contact", contact).Methods("GET")
r.HandleFunc("/signup", usersC.New).Methods("GET")
http.ListenAndServe(":3000", r)
```

Stop (**ctrl + c**) and restart your server to verify that things are working.

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

Everything appears to be working as we intended. Nice!

Next we want to add a new route that uses the `/signup` path again, but this time it will only accept the `POST` method and should route the request to the `Create` method we created on the users controller. By doing this we will be able to submit our sign up form to the `/signup` path and then our create action will be called to process the form.

Box 7.8. Another exception to RESTful design

Once again, we will be making another exception to RESTful design for the sign up form. Rather than having this form submit to `/users` with a POST, we are going to have it POST to `/signup`. We don't have to do this, but it is fairly common practice one that I feel gives end users a better overall experience when creating a new account.

You should still have `main.go` open, so head to the `main` function inside that source file. Listing 7.19 demonstrates how to add a new route that only gets run when the HTTP method is POST.

Listing 7.19: Adding the POST sign up route

```
r := mux.NewRouter()
r.HandleFunc("/", home).Methods("GET")
r.HandleFunc("/contact", contact).Methods("GET")
r.HandleFunc("/signup", usersC.New).Methods("GET")
r.HandleFunc("/signup", usersC.Create).Methods("POST")
http.ListenAndServe(":3000", r)
```

The last change we are going to make is to our sign up form's HTML. Right now our form doesn't know where to submit the data, so we need to update it to tell it to POST the data to `/signup`.

Open up the new user template.

```
$ atom views/users/new.gohtml
```

Look for the `signupForm` template that we created earlier. It should be the template that has our HTML form defined inside of it.

The first line in this template should read `<form>` which is a basic form tag. We are going to update this line to tell our browser where to submit the form when we hit the submit button, and we are also going to tell it to use the HTTP POST method. This code is shown in [Listing 7.20](#).

Listing 7.20: Update the sign up form tag

```
<form action="/signup" method="POST">
```

If you haven't already, restart your server and head over to the sign up form. Make sure you refresh sign up page after restarting your server so you get the updated HTML, then submit the form. If things are working correctly, you should be presented with a pretty ugly page that renders the text "This is a temporary response." That means our everything is setup correctly and our `Create` method on the users controller executed!

Source code

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

Completed - book.usegolang.com/7.4.2

Changes - book.usegolang.com/7.4.2-diff

7.4.3 Parsing a POST form

We are finally ready to start learning about how to parse the sign up form so that we can use the submitted data to create a new user.

When a form is submitted to your web application it will be included in the web request that the user is making. Or in Go terms, it will be part of the **http.Request** parameter passed into your handler function.

Before you can access the form, you first need to tell your code to parse it. This isn't done by default because forms aren't always included in web requests, so it would be pretty pointless to always try to parse a form when one isn't there most of the time. You can parse a form in Go by calling the **ParseForm**⁷ method on the **http.Request** that is passed into your handler function.

If you were to read the docs for the ParseForm function they might seem pretty daunting at first. There is a lot of talk about HTTP methods, raw queries, and MaxBytesReader, all of which probably doesn't mean much to you at this point. All of this information is there because parsing data in a web request can be complicated, and the docs need to outline every edge case.

Luckily, our case happens to be fairly standard. When we call ParseForm from within our **Create** method it will parse the HTML form that was submitted and then store the data in the **PostForm** field of the **http.Request**.

Open up our users controller and we will test this out.

```
$ atom controllers/users.go
```

⁷<https://golang.org/pkg/net/http/#Request.ParseForm>

We are going to parse our form, check for errors, then print out the two fields we expect to be in our HTML form. The code for this is shown in Listing 7.21. We will discuss the **PostForm** portion of the code after writing it.

Listing 7.21: Accessing **r.PostForm** data

```
func (u *Users) Create(w http.ResponseWriter, r *http.Request) {
    if err := r.ParseForm(); err != nil {
        panic(err)
    }
    fmt.Fprintln(w, r.PostForm["email"])
    fmt.Fprintln(w, r.PostForm["password"])
}
```

The call the ParseForm should be pretty easy to follow. The function call returns an error, so we capture the return value and proceed to verify that there wasn't an error parsing the form.

After that we print out some data that we retrieve from **r.PostForm**. Looking in the net/http docs at our request object⁸, we can see that this field has the type **url.Values**.

If we follow the link to the url.Values docs⁹ we will find the definition of the type (*shown in Listing 7.22*).

Listing 7.22: **url.Values** definition

```
type Values map[string][]string
```

Our PostForm field is really just a map behind the scenes, which means that we can access fields stored in the PostForm field in the same way we would access fields in a map - by using the **["key"]** syntax. That is what we are doing in Listing 7.21 when we write the code:

⁸<https://golang.org/pkg/net/http/#Request>

⁹<https://golang.org/pkg/net/url/#Values>

```
r.PostForm["email"]
```

We are telling our code that we want whatever value is stored in our PostForm map with the key “email”.

Now is a great time to stop your server and restart it to test things out. Head over to the sign up page and fill out each form then hit the submit button. If everything is working as intended you should see the information you entered into the HTML form. It will look similar to the data below.

```
[jon@calhoun.io]
[test1234]
```

The square brackets (`[` and `]`) might throw you off at first, but they are present because we are printing out a slice of strings. If you refer to [Listing 7.22](#) you will see that each value stored in our map is actually a slice of strings instead of a single string. We won’t be storing multiple values for each key, but it is an important detail to take note of before moving on.

Source code

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

Completed - book.usegolang.com/7.4.3

Changes - book.usegolang.com/7.4.3-diff

7.4.4 Parsing forms with **gorilla/schema**

We have seen how to parse a form and access the data manually, but parsing data this way can be tedious. We need to write code to pull out each field of our form, convert it to the correct data type, and then assign it to whatever data structure we want to work with in our code.

Rather than writing all of this code and potentially introducing a bug into our code, we are going to take advantage of another library in the Gorilla web toolkit¹⁰ - the **schema** package¹¹.

The schema package was designed to make it easier to convert form values into a Go struct. Not only will it handle moving the data from our PostForm and into our struct, but it will also handle most common type conversions for us. For example, if our struct field had a type of **int** then the gorilla/schema package would convert the form value into an integer before assigning it to the struct field.

This is super handy because it means we can continue to write our Go code like we normally would and we also have a quick and easy way to convert the data from any incoming forms into our Go types.

Gorilla's schema package is a third party package, so we will first need to **go get** it.

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

*Note: The **-u** flag is used to get an updated version of the package.*

Next we need to define a type to store our data in. We will be adding this to the **users.go** source file inside of the controllers package.

¹⁰<http://www.gorillatoolkit.org/>

¹¹<http://www.gorillatoolkit.org/pkg/schema>

```
$ atom controllers/users.go
```

We know our form is going to be used for the sign up page, so we will name the form `SignupForm` for now. In the future we might want to create a shared `UserForm` type, but for now we don't know for sure that this will be shared so we can start off with a very specific name.

We also know what input fields we have in our sign up form. One is for the user's email address, and another is for their password. We will want to store both of these after parsing our form so we need to add fields for each to our `SignupForm` type.

```
type SignupForm struct {
    Email    string
    Password string
}
```

The last thing we need to add will be new for many readers. We are going to add some struct tags¹² to our type.

Box 7.9. Struct tags

Struct tags are a form of metadata that can be added to the fields of any struct that you create. When coding them in they will be in the format ``key: "value" ``, and are found directly after the field's type (see [Listing 7.23](#) for an example).

Later, when your program is running, other packages can use the `reflect` package to look up each struct tag and then use that data to determine how it should proceed. For example, the `encoding/json` package uses this data to determine what key you want used for each field when converting a struct into JSON.

¹²<https://golang.org/pkg/reflect/#StructTag>

While it isn't a requirement, most packages will use their package name as the key for all of their struct tags. This makes it easier to determine what package each struct tag is used for when reading your source code.

In Listing 7.23 each struct tag starts with the key **schema** which denotes that it is used by a package named schema. Similarly, all struct tags for the encoding/json package use the key **json**.

We are going to need struct tags because we don't currently have a way to tell the schema package how to map our data in our form to fields in our SignupForm struct. We have both an "email" and "password" input in our sign up form's HTML, but the schema package doesn't know about them.

To fix this, we simply need to add struct tags that let the schema package know about the input fields. The code for this is shown in Listing 7.23.

Listing 7.23: Sign up form struct

```
type SignupForm struct {
    Email    string `schema:"email"`
    Password string `schema:"password"`
}
```

We are now ready to start using the schema package to decode our HTML form and set the data to fields in our SignupForm type. To do this we first need to initialize both a decoder and a SignupForm. Once we have both of those we are going to call the **Decode** method on our decode and pass in the SignupForm as our destination and **r.PostForm** as our data source. While we are at it, we will also check for errors.

The code for this is shown in Listing 7.24.

Don't forget to call **r.ParseForm() in your code!** If you forget to do this then **r.PostForm** won't have any values for our decoder to work with.

Listing 7.24: Using **gorilla/schema** in the create method

```
// Add gorilla/schema to your imports
import (
    "fmt"
    "net/http"

    "github.com/gorilla/schema"
    "lenslocked.com/views"
)

// ... We are only editing the Create method.

func (u *Users) Create(w http.ResponseWriter, r *http.Request) {
    if err := r.ParseForm(); err != nil {
        panic(err)
    }

    dec := schema.NewDecoder()
    form := SignupForm{}
    if err := dec.Decode(&form, r.PostForm); err != nil {
        panic(err)
    }
    fmt.Fprintln(w, form)
}
```

Stop and restart your web app, then head over to the sign up page and submit it. It shouldn't be a lot different from last time, but now we should have our SignupForm type being printed out instead of two string slices.

Source code

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

Completed - book.usegolang.com/7.4.4

Changes - book.usegolang.com/7.4.4-diff

7.4.5 Keeping our parsing code DRY

Programmers don't like to write the same code over and over again, so at some point they came up with the saying "Don't repeat yourself". Or in acronym form - DRY.

What it basically means is that if you find yourself writing code that you are likely to write several times, you should consider breaking it into a function or some reusable piece of code. This helps us avoid bugs where we fix the same code in four places, but miss the fifth and final place where we wrote that code.

While working on our users controller we wrote some code inside of our **create** method that parses a form and decodes it, but if you look closely at the code you will notice that almost all of the code used to parse the HTML form and decode it is pretty generic.

In this section we are going to look at how to move that code out of our Create method and instead move it into its own function so that we can reuse it moving forward. We don't really have a great place to store this yet, so we are going to create a new source file named **helpers.go** inside of our controllers package and store the code there. This isn't the best name in the world, but it will work for now.

```
$ atom controllers/helpers.go
```

Next we will write the package declaration code import the packages we will need. Normally I wouldn't import packages until I use them, but since many of you won't have an auto-importer setup I want to cover this before we get into the code.

```
package controllers

import (
    "net/http"
```

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

Now that we have our imports we are going to create a function named `parseForm` that will take in two arguments. The first will be a `*http.Request` because we want our helper function to handle calling the `ParseForm` method and checking for any errors. This will also be necessary if we want to access the HTML form data.

The second argument our function is going to accept is the destination where we want to store the information we parse from the HTML form. For this we will use the empty interface type (`interface{}`), which is our way of saying that it can be any type. This will allow us to pass in a pointer to a `SignupForm` or any other form types we declare later in our code.

Accepting the empty interface type also means that it is possible to pass in a variable that can't be decoded by the `schema` package. There isn't a way to prevent this at the compiler level, so instead we are going to have our function return an error when this happens. We will also return errors when any other issues arise, like an issue when initially parsing the HTML form.

Putting that all together, our initial function definition should look like the code in [Listing 7.25](#).

Listing 7.25: Initial `parseForm` definition

```
func parseForm(r *http.Request, dst interface{}) error {
}
```

Now we need to implement the `parseForm` function. Luckily we already wrote most of this code in the `Create` method inside of our users controller. We just need to copy it over and make a few changes to it. Open up your users controller...

```
$ atom controllers/users.go
```

And look for the the code we wrote earlier that parses a form and decodes it using the **schema** package. It should be most of the code inside of the **Create** method.

```
if err := r.ParseForm(); err != nil {
    panic(err)
}
dec := schema.NewDecoder()
form := SignupForm{}
if err := dec.Decode(&form, r.PostForm); err != nil {
    panic(err)
}
```

Copy this code and paste it into your **parseForm** function you just created.

Our **parseForm** function won't ever panic and will instead return an error when occurs, so find the two lines of code that call **panic** and change them to instead return the error. We also need to return **nil** whenever no errors occur, so add that to the end of the function.

```
func parseForm(r *http.Request, dst interface{}) error {
    if err := r.ParseForm(); err != nil {
        return err
    }
    dec := schema.NewDecoder()
    form := SignupForm{}
    if err := dec.Decode(&form, r.PostForm); err != nil {
        return err
    }
    return nil
}
```

Our code won't work as intended just yet because we aren't decoding into the **dst** argument that is provided. We still have the old **SignupForm** variable being used.

Delete the line that creates the `form` variable, and then update the `Decode` method call to use the `dst` argument. We don't need an ampersand (`&`) anymore because we are going to expect the `dst` argument to already be a pointer.

```
func parseForm(r *http.Request, dst interface{}) error {
    if err := r.ParseForm(); err != nil {
        return err
    }
    dec := schema.NewDecoder()
    if err := dec.Decode(dst, r.PostForm); err != nil {
        return err
    }
    return nil
}
```

Now we can update the `Create` method in the users controller to use the `parseForm` function we just created.

```
$ atom controllers/users.go
```

We still want to create the `SignupForm` variable, but after that we call `parseForm`, check for an error, and then print out our form data. Much easier!

The final code should match Listing 7.26.

Listing 7.26: Using `parseForm()` inside `Create()`

```
func (u *Users) Create(w http.ResponseWriter, r *http.Request) {
    var form SignupForm
    if err := parseForm(r, &form); err != nil {
        panic(err)
    }
    fmt.Fprintln(w, "Email is", form.Email)
    fmt.Fprintln(w, "Password is", form.Password)
}
```

We still have to handle an error in this code, but we were able to reduce it to only one error check inside of our `Create()` action, which simplifies things a

bit. On top of that, we now have a way to parse any form without rewriting that code.

The last bit of code also update our print statements to demonstrate that we definitely can access each field on the `SignupForm` instance and it has the correct value assigned to it.

All that is left to do in this chapter is to clean up our code and try out a few exercises.

Source code

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

Completed - book.usegolang.com/7.4.5

Changes - book.usegolang.com/7.4.5-diff

7.5 Cleaning up and creating a static controller

Thus far we have been working with the users controller, but in this section we are going to look at how you can use a controller for pages that don't really map to a resource like a user, but instead are closer to static pages in your web application.

A great example of this is your contact page. You might be able to put this under something like an inquiries controller because it is how users find out how to make inquiries, but this controller would end up only have one action - the page that shows all of your contact information.

Instead of creating a bunch of unnecessary controllers that could get confusing, we are going to create one controller to hold all of our pages that are not quite

static pages, but are pretty close to being a static page. We are going to call this the static controller, and inside of it we will have actions for the home page and the contact page.

7.5.1 Creating the static controller

Just like before when we created the users controller, our static controller will live inside the controllers package. Create and open a new source code file inside of the controllers directory for your static controller.

```
$ atom controllers/static.go
```

The first thing we need to do is declare the **Static** type. Inside of this we need to store two different views - the home view, and the contact view. Then, just like we did with the users controller, we are going to create a **NewStatic** function that initializes our a static controller, including initializing our views.

Listing 7.27: Starting to create a static controller

```
package controllers

import "lenslocked.com/views"

func NewStatic() *Static {
    return &Static{
        HomeView: views.NewView(
            "bootstrap", "views/static/home.gohtml"),
        ContactView: views.NewView(
            "bootstrap", "views/static/contact.gohtml"),
    }
}

type Static struct {
    HomeView     *views.View
    ContactView *views.View
}
```

Next, we need to move our views to the correct location. We will be following the same pattern as before, so both `home.gohtml` and `contact.gohtml` (both in the `views` directory) will need to be moved into a static directory where all of our template files for the static controller will live. To do this we need to create the `views/static` directory, then move our `.gohtml` files into the new directory.

Listing 7.28: Moving the static controller templates

```
$ mkdir views/static
$ mv views/home.gohtml views/static
$ mv views/contact.gohtml views/static
```

Next we need to write our handler functions inside of our controller, but if you look at both the `home` and `contact` functions in `main.go` you will notice that neither of these are really doing much.

In fact, the only thing they actually do is set a content type and then render a view. We don't really need a custom function to make that happen. It would be nice if we could just use our `View` type from the `views` package as a handler function that simply renders the view.

To do this, we need to figure out what interface our `views.View` type needs to implement so that we can simply pass our `views.View` type to our router. That means heading back over to the [gorilla/mux](#) docs again.

Skimming over this list, we are looking for a method on the `mux.Router` type (because that is what our router is) that will let us pass in an interface implementation.

Well, the only two methods that look plausible are `Handle` and `HandleFunc`, but we know that `HandleFunc` isn't what we want because we have been using it, so let's check out the `Handle` method.

The `Handle()` method appears to take in a path, and then an `http.Handler` object. That means we need to head on over to the net/http docs and look for

the `http.Handler` type.

The `http.Handler` is an interface that just needs one method:

```
ServeHTTP(ResponseWriter, *Request)
```

That sounds like what we want with our `views.View` type - a function that receives an http response writer and a request - so it appears that we just need to write a `ServeHTTP()` method for our `views.View` type.

Open up our views source file.

```
$ atom views/view.go
```

In the views source code we are going to write a `ServeHTTP` method that resembles the `home` and `contact` functions we have in `main.go`. While we are at it, we should go ahead and move the code that sets the content-type into the `Render` method.

Listing 7.29: Implementing the `http.Handler` interface

```
func (v *View) Render(w http.ResponseWriter, data interface{}) error {
    w.Header().Set("Content-Type", "text/html")
    return v.Template.ExecuteTemplate(w, v.Layout, data)
}

func (v *View) ServeHTTP(w http.ResponseWriter, r *http.Request) {
    if err := v.Render(w, nil); err != nil {
        panic(err)
    }
}
```

Now that `views.View` implements the `http.Handler` interface, we can update `main.go` to just use the views in our static controller as handlers.

```
$ atom main.go
```

That means we can delete the `home` and `contact` functions and the global views, and instead just create a static controller and use its views as our handlers. Listing 7.30 has the updated code.

Listing 7.30

```
package main

import (
    "net/http"

    "lenslocked.com/controllers"
    "github.com/gorilla/mux"
)

func main() {
    staticC := controllers.NewStatic()
    usersC := controllers.NewUsers()

    r := mux.NewRouter()
    r.Handle("/", staticC.HomeView).Methods("GET")
    r.Handle("/contact", staticC.ContactView).Methods("GET")
    r.HandleFunc("/signup", usersC.New).Methods("GET")
    r.HandleFunc("/signup", usersC.Create).Methods("POST")
    http.ListenAndServe(":3000", r)
}
```

Lastly, since we don't plan on creating a `Home` or `Contact` method on the static controller, we can rename our fields from `HomeView` and `ContactView` to just be `Home` and `Contact`. After doing this you will also need to update `main.go`.

Listing 7.31: Renaming view fields in `controllers/static.go`

```
package controllers

import "lenslocked.com/views"
```

```
func NewStatic() *Static {
    return &Static{
        Home: views.NewView(
            "bootstrap", "views/static/home.gohtml"),
        Contact: views.NewView(
            "bootstrap", "views/static/contact.gohtml"),
    }
}

type Static struct {
    Home      *views.View
    Contact   *views.View
}
```

Listing 7.32: Renaming handler fields in `main.go`

```
r.Handle("/", staticC.Home).Methods("GET")
r.Handle("/contact", staticC.Contact).Methods("GET")
```

Restart your application and verify that it is working. You should still be able to view the home and contact pages, but now they should be served via the **ServeHTTP** handler method on the `views.View` type inside of our static controller.

Source code

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

Completed - book.usegolang.com/7.5.1

Changes - book.usegolang.com/7.5.1-diff

7.5.2 Simplifying the creation of new views

Open up both of your controllers (`users.go` and `static.go`) and look at the code that calls the `NewView` function provided by the `views` package. Do you notice anything in common between these two pieces of code?

If you look at the view paths we are passing into these function calls you will notice that they have a lot in common. They both start with the same prefix (`views/`), and they both end with the same file extension (`.gohtml`). Given that we don't plan to store view files outside of the `views` directory, and we don't intend to use another template extension (using multiple would be pretty confusing), it would be nice if we could simplify that code a bit to not require this information. Eg we might instead call:

```
views.NewView("bootstrap", "users/new")
```

To make this happen, we are going to need to update our `views` source code.

```
$ atom views/view.go
```

The first thing we are going to do is add a variable that stores the directory where all of our variables are stored. We will call this `TemplateDir`. The code for this is shown in [Listing 7.33](#).

Listing 7.33: Template variables in `view.go`

```
var (
    LayoutDir  string = "views/layouts/"
    TemplateDir string = "views/"
    TemplateExt string = ".gohtml"
)
```

What we need next is a function that can take the files that are provided to the `NewView` function and can update them to automatically prepend the `views/`

prefix, and append the `.gohtml` suffix. We are going to name this function `addTemplatePath` and it will take in one argument - a slice of strings that represent our template files. `addTemplatePath` will then iterate over every string in the slice and set a new value for each entry in the slice that has the `views/` string prepended to it. The code for this is shown in Listing 7.34.

Box 7.10. Why don't we need a slice pointer?

Most of the time if we wanted to alter data passed into a function we would need to use a pointer variable, so why don't we need to do this with our slice passed into the `addTemplatePath` function?

Behind the scenes slices actually reference an array via a pointer. When you attempt to alter the size of the slice the changes won't be persisted, but if you instead alter some of the data inside of the slice you will end up altering the data in the referenced array.

You can read more about this here: <https://www.calhoun.io/why-are-slices-sometimes-altered-when-passed-by-value-in-go/>

Listing 7.34: Prepending the `views` directory

```
// addTemplatePath takes in a slice of strings
// representing file paths for templates, and it prepends
// the TemplateDir directory to each string in the slice
//
// Eg the input {"home"} would result in the output
// {"views/home"} if TemplateDir == "views/"
func addTemplatePath(files []string) {
    for i, f := range files {
        files[i] = TemplateDir + f
    }
}
```

Next, we are going to write a similar function, but instead of prepending the

views directory we are going to append the template extension suffix. We are going to name this function `addTemplateExt`. This code is shown in Listing 7.35.

Listing 7.35: Appending the `.gohtml` extension

```
// addTemplateExt takes in a slice of strings
// representing file paths for templates and it appends
// the TemplateExt extension to each string in the slice
//
// Eg the input {"home"} would result in the output
// {"home.gohtml"} if TemplateExt == ".gohtml"
func addTemplateExt(files []string) {
    for i, f := range files {
        files[i] = f + TemplateExt
    }
}
```

We aren't using either of these functions yet, so let's update `NewView` to take advantage of them. This code is shown in Listing 7.36.

Listing 7.36: Using the view file path helpers

```
func NewView(layout string, files ...string) *View {
    addTemplatePath(files)
    addTemplateExt(files)
    files = append(files, layoutFiles()...)
    t, err := template.ParseFiles(files...)
    if err != nil {
        panic(err)
    }

    return &View{
        Template: t,
        Layout:   layout,
    }
}
```

Finally, we need to update our controllers so that they don't pass in unnecessary data when creating a view.

```
$ atom controllers/users.go
```

Update the **NewUsers** function in the users controller so that the call to the **NewView** function doesn't pass in the views prefix or the **.gohtml** extension. This is shown in Listing 7.37.

Listing 7.37: Cleaning up the **NewUsers()** function

```
func NewUsers() *Users {
    return &Users{
        NewView: views.NewView("bootstrap", "users/new"),
    }
}
```

We need to do the same thing to the views in the static controller.

```
$ atom controllers/static.go
```

This time we will be updating the **NewStatic** function. Both the home and contact vies need tweaked. The code for this is in Listing 7.38.

Listing 7.38: Cleaning up the **NewStatic()** function

```
func NewStatic() *Static {
    return &Static{
        Home: views.NewView("bootstrap", "static/home"),
        Contact: views.NewView("bootstrap", "static/contact"),
    }
}
```

Our code should be in a good state to restart the server and test everything out, so go ahead and do that.

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

If there is an error you will likely see an error as soon as the server starts, but visit the home, contact, and sign up pages to verify they all work. If the page is blank, check your server logs for any errors.

Box 7.11. Potential errors

If you get an error with a message like “*no such file or directory*” this likely means that you either didn’t update all of the views being created in the controllers, or that you have a bug in your **view.go** source file.

Look at the file your code is trying to open and this should give you a clue about what is going wrong. For example, if the file it is trying to access starts with **views/views/...** (duplicate “views” directories) it likely means that your call to **NewView** is passing in the **views/** prefix, and then the **NewView** function is calling the **addTemplatePath** prefix, resulting in duplicate prefixes.

Likewise, if the file ends with **.gohtml.gohtml** it probably means that you are passing in the extension with your files and then **NewView** function is appending another extension.

Source code

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

Completed - book.usegolang.com/7.5.2

Changes - book.usegolang.com/7.5.2-diff

7.6 Exercises

We covered a lot in this chapter, but towards the end we started to repeat a lot of things that we learned earlier in the chapter. For example, creating a static controller wasn't really much different than creating a users controller.

While we aren't technically using a framework, what we are doing is very similar to building our own. We don't have any fancy generators or other tools to help automate things, but we have helpers for creating views, a common design for controllers, and before long we will also have something similar for our database layer.

Doing this also gives us a lot of the benefits of a framework. For example, once a newcomer learns how we designed our controllers they could easily jump between all of our controllers without much effort.

With that in mind, we are going to focus on some exercises that will help you reinforce your knowledge of how our application is structured. You will need to edit a few files for each of these tasks, but you shouldn't need to make a large number of changes. Instead you should be doing some simple tasks like creating a method on a controller, updating `main.go` to inform the router about this method, and creating a simple view, gaining experience with our entire code base.

7.6.1 Ex1 - Add the FAQ page to the static controller

Just like many web applications have a contact page, our application might eventually need an FAQ page. Create a template (`.gohtml` file) for the FAQ page, and then add the new view to the static controller.

Once you have added it to the static controller, update your code in `main.go` so that the router sends visitors to this page when they visit the `/faq` path.

7.6.2 Ex2 - Create a new controller for galleries

We are eventually going to create a galleries resource, including a controller for it, but in the meantime we are going to use the galleries controller as an exercise.

Create a new galleries controller that has one action, the new action, represented by the **New** method.

You will first need to create a file where you will be declaring your galleries controller. Inside of that file you will want to create a type for the galleries controller, and then create a function that can be used to create a new galleries controller.

After that you will need to create the **New()** method on the type you created to represent your galleries controller. You could technically do this with just a view like we did the FAQ page, but instead I would like you to write a method so that you get some practice doing that as well.

Inside of the **New** method you will need to render a view, so you will need to create one. I would recommend using the new user template as a guide for this.

Finally, you will need to register the **New** method with the router. Use the path **/galleries/new** with the HTTP **GET** method.

7.6.3 Exercise cleanup

We won't be using any of the code from our exercises moving forward, so feel free to delete it all once you have completed the exercises. These were just meant to help reinforce what we are learning.

Chapter 8

An introduction to databases

A web application that didn't save any data wouldn't be very interesting, so we need to talk about ways to actually persist data between web requests.

Technically, we could use anything to store our data, but every approach has its own unique set of pros and cons. That means that there isn't a true "best" option, but instead we need to figure out the best overall solution for our specific needs.

For example, if we wanted a really simple data storage option we could store all of our data in a JSON text file. If we didn't have a lot of data this would be very quick and would allow us to easily read and edit data manually with a text editor.

The downside to this approach is that it might become slow if we start storing too much data, and we could easily have write conflicts if two applications tried to write to the file at the same time.

A JSON file is likely a bad idea for persisting our data, but it helps illustrate the point that every option is good for some situations and bad for others. Databases are exactly the same. There isn't a single silver bullet and no matter what anyone tells you, one single database won't work for every situation. Even companies

like Google and Facebook must utilize several different data storage strategies depending on their needs.

8.1 Our web app will use PostgreSQL

Our web application is going to use an SQL¹ database. Specifically, we are going to be using PostgreSQL², but remember that this doesn't mean that all of your future applications need to use SQL. In fact, once you complete this book I would encourage you to look into out a few other databases so you can get a feel for what else is out there, but SQL is typically my go-to choice.

I am opting to use SQL throughout this book for a few reasons. First, it is an incredibly popular database, and as a result there are a large number of resources available. This is important because the primary focus of this book is to teach you web development with Go, and as much as I would love to it isn't feasibly to also teach SQL within this book. Instead, I am going to recommend that you check out [Box 8.1](#) if you are unfamiliar with SQL. It will provide you with several different resources intended to help you get familiar with SQL.

Box 8.1. Resources for learning SQL.

The following are resources that I recommend checking out if you are not familiar with SQL. You don't need to read them all, and you don't need to become an expert with SQL. Instead you should be shooting to learn just enough that you understand how to create, update, and query data inside your database.

Using PostgreSQL with Golang - This is a series I wrote that covers installing PostgreSQL on various operating systems, the basics of using SQL, and then ex-

¹<https://en.wikipedia.org/wiki/SQL>

²<https://www.postgresql.org/>

plains the basics of how to access an SQL database with your Go code. You can find the course at <http://www.calhoun.io/using-postgresql-with-golang/>

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/lrn/learn-sql>

w3schools.com has 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 also offers an SQL course that can be found here: <https://www.khanacademy.org/computing/computer-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>

The second reason we are going to be using SQL is its blend of scalability and usability. While other databases might work better with billions of users, that comes at the cost of complexity. PostgreSQL on the other hand has a nice blend of both scalability and usability without being too complicated to use or setup. While it might not scale to billions of users easily, it will scale to millions with relative ease, and it is pretty easy to adapt to nearly any use case. This means that we can safely pick it knowing that we are unlikely to run into any major issues until we get to a very large scale. At that point we will have a much better understanding of what our limitations are and we will be able to make educated decisions on what database would meet our needs.

Finally, we are opting to use SQL because it is supported very well within the Go community. The standard library offers an official [database/sql](#) package, and there are a wide variety of third party packages built on top of SQL that simplify things like reading, writing, and migrating data.

While we won't be using the standard library's `database/sql` package, and will instead be opting to use GORM, the code in this book is setup in a way that you could easily replace all of the GORM code with the `database/sql` package with relative ease, so don't be discouraged if you were hoping to use more raw SQL. You still can, but I advice waiting until you have a working application to do so.

8.2 Setting up PostgreSQL

Before we can get started using PostgreSQL with Go you are going to need to do a little bit of work outside of this book.

While I hope to eventually add all of the content required for getting started with SQL and Postgres as either an appendix or additional chapter to this book, I am opting to refer you to an external resource (my blog) for now. You can see the articles here: <http://www.calhoun.io/using-postgresql-with-golang/>

Before heading to the next chapter, you will need to have a basic understanding of everything covered in the following four subsections.

8.2.1 Install PostgreSQL

How you do this will vary based on your operating system, but you will want to find a guide for your OS and follow along with to get Postgres installed.

While installing Postgres, try to keep an eye out for the information in [Table 8.1](#), as you will need this later in the book.

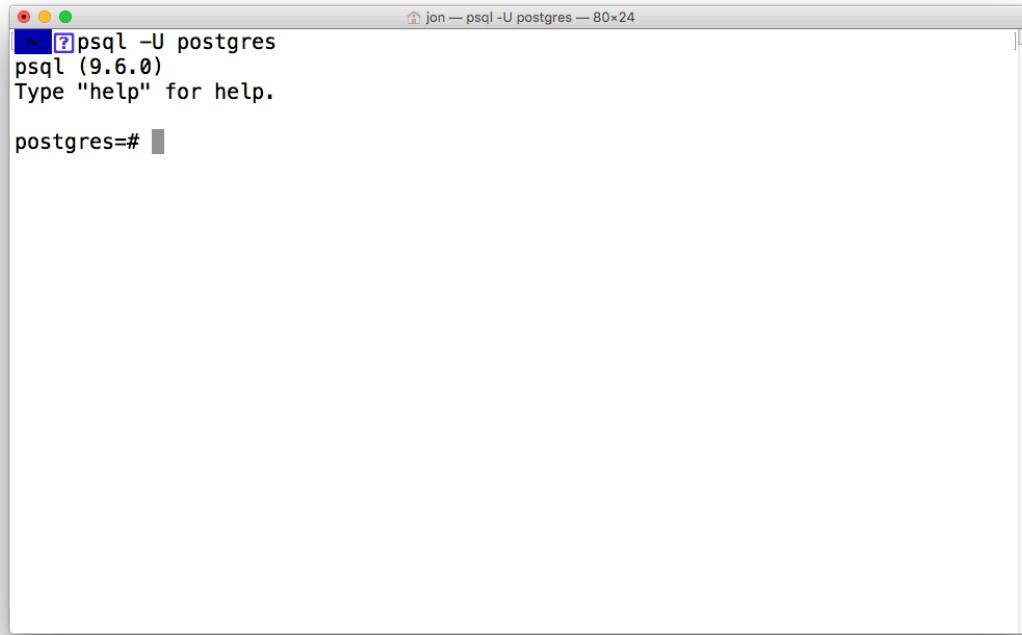
The guides on my blog should all keep this information pretty consistent regardless of your OS, but if you use a different guide it might not have you setup the `postgres` role.

8.2.2 Learn how to connect to Postgres

Once you have installed Postgres you need to learn how to connect to your database to create tables, insert records, etc.

We will be using this to create and drop tables to test out the **database/sql** package with, so it is important that you are able to get your database to a state to follow along with the book.

Connecting to your Postgres instance is most commonly done with **psql**, a command line utility that will open up a text-based connection to your database and then allow you to interact with the database using raw SQL queries. Once you have connected with **psql** you should find yourself looking at something like [Figure 8.1](#).



```
jon — psql -U postgres — 80x24
[?] psql -U postgres
psql (9.6.0)
Type "help" for help.

postgres=#
```

Figure 8.1: A terminal connected to PostgreSQL

If you find yourself typing things like `SELECT * FROM users;`, you are likely connected to your database using something like `psql` and will be fine to proceed.

8.2.3 Learn the basics of SQL

As I said in the last section, we will be using `psql` to create and drop tables, so you will want to understand how to create and drop tables in SQL if you want to have a rough understanding of what is going on inside of our database as we start to interact with it.

On top of knowing how to drop and create tables, you will also want to be familiar with some of the more common SQL queries.

You will want to have experience inserting new records into a table, querying for those records, and you probably also want to know how to update and delete existing records.

If you understand all of the code samples below you are ready to move on. You don't need to understand them 100%, but you should at the very least be vaguely aware of what they do and comfortable modifying the SQL a bit to suite your needs.

Listing 8.1: Creating a table with raw SQL.

```
CREATE TABLE users (
    id SERIAL PRIMARY KEY,
    age INT,
    first_name TEXT,
    last_name TEXT,
    email TEXT UNIQUE NOT NULL
);
```

Listing 8.2: Inserting a record with raw SQL.

```
INSERT INTO users (age, email, first_name, last_name)
VALUES (30, 'jon@calhoun.io', 'Jonathan', 'Calhoun');
```

Listing 8.3: Querying records with raw SQL.

```
SELECT *
FROM users
WHERE email = 'jon@calhoun.io'
AND age > 16;
```

8.2.4 Gather information needed to connect to your Postgres install

Lastly, you will want to gather all of the information you use to connect to your database and jot it down. Specifically, we need to know the host, port, user, password, and the database name.

We will be using this information in our Go code to connect to the database, and if you get it wrong it is likely that you will get errors when trying to connect to the database.

Unfortunately, this information will vary based on how you installed Postgres, so I can't definitively say what you should be using. What I can do is provide you with some of the common values and you can use them to help you figure out which value was used in your installation guide.

The first two you should look for is the host and the port. If you installed Postgres locally, this is most likely going to be **localhost** for the host, and **5432** for the port.

Next up is the user (aka the role). In most of the tutorials your user is going to be **postgres**, but another common value is your operating system username. To find this on Mac OS X or Linux you can run **id -un** in the terminal. Mine happens to be **jon**, but yours is likely going to be something you chose when setting up your computer.

If you find yourself typing something like **psql -U <something>** then that

Table 8.1: PostgreSQL connection parameters and values.

Parameter	Value (yours may vary from what is listed here)
host	localhost
port	5432
user	postgres
password	<none>
dbname	<code>lenslocked_dev</code>

something is your user for Postgres. The **-U** flag is how you connect to Postgres with a specific user.

After that we need your password. This one should be pretty obvious because most tutorials have you set a value on your own, and if you didn't set one you can likely use the empty string for your password.

Lastly, we need to figure out your database name, which we will be listed as the **dbname** in our code. For this one you likely either created a database during your installation, or it could possibly be the same as your Postgres user.

If you did create a database in your tutorial, I suggest you follow those steps again to create a database named **lenslocked_dev** or **<yourapp>_dev** so that it is obvious what the database is used for.

Once you have all of that, jot it down in a table like the one in [Table 8.1](#). I have provided you with the values I will be using in the table, but yours might be different. Just make sure you use the data in your own table in the upcoming Go code.

Once you have all of that you are ready to start using PostgreSQL with Go and are ready to proceed to the next section!

8.3 Using Postgres with Go and raw SQL

While Go offers the `database/sql` package as part of the standard library, we will also need to install a Postgres specific driver to go along with it. For this we are going to use github.com/lib/pq which is written in Go and passes the compatibility test for SQL drivers.

To do this you will need to use the `go get` command again.

Listing 8.4: Install `lib/pq`.

```
$ go get -u github.com/lib/pq
```

8.3.1 Connecting to Postgres with the `database/sql` package

Next we need to write some code that actually uses these two packages. This isn't code we will use in our final product, so the `exp` directory we created in Section 4.4 is going to be a good place for this code to live.

If you don't already have a file at `exp/main.go` you should create one. You may also need to create the `exp` directory.

```
$ mkdir exp # this may already exist
$ atom exp/main.go
```

Delete everything in the file your experimental source file. We are going to be starting from scratch and won't need any of the existing code.

The first thing we want to do is declare our package and specify our imports. We will be importing the `database/sql` package so that we can access our SQL database, the `fmt` package for printing to the terminal and formatting strings, and finally the `github.com/lib/pq` package which implements the Postgres driver used by the `database/sql` package.

Listing 8.5: The start of our code to connect to Postgres.

```
package main

import (
    "database/sql"
    "fmt"

    _ "github.com/lib/pq"
)
```

Box 8.2. Why do we import a package that we don't use?

At first glance the code in Listing 8.5 might look a little different, and you might be asking yourself “what is that underscore doing there?”.

The short version of what is happening is that Go doesn’t allow us to import packages that we don’t directly use in our code.

This is problematic because we need the `github.com/lib/pq` package to be imported so that it’s `init()` function gets called which will register the “`postgres`” driver for the `database/sql` package.

In other words, we can’t connect to our Postgres database unless we have the `github.com/lib/pq` package in our code, but we don’t have any real reason to use this package in our code. Including it does everything we need.

By putting the `_` character before the package we are telling the Go compiler that we won’t be directly using this package, but we still need it to be imported.

If you are interested in reading more about this, I wrote a lengthier post about it on my blog: <http://www.calhoun.io/why-we-import-packages-we-dont-actually-use-in-golang/>

After our imports we are going to declare some constants to store all of our

information about connecting to our database.

Listing 8.6: Constants for connecting to Postgres

```
const (
    host      = "localhost"
    port      = 5432
    user      = "postgres"
    password  = "your-password"
    dbname    = "lenslocked_dev"
)
```

These values should come from the ones you wrote down while gathering information about your Postgres installation (see [Table 8.1](#)).

It is typically frowned upon to put things like passwords directly in your code because this means that any developer who you share your code with can access your password, but in this case we are going to make an exception because we will generally be connecting to a database that is hosted locally and isn't exposed to the public.

Later in the book when we discuss deploying to a production server we will look at ways to take values like these and move them outside of our code and instead provide them to our code via flags that we pass in when we execute the code.

Next we want put all of this together to create a connection string that we use to connect to our database. We will only be using a few of the possible values, but you can read more about every possible parameter in the [docs for github.com/lib/pq](#).

Listing 8.7: Creating the connection string

```
pgsqlInfo := fmt.Sprintf("host=%s port=%d user=%s "+
    "password=%s dbname=%s sslmode=disable",
    host, port, user, password, dbname)
```

The code in [listing 8.7](#) will create a string like `"host=localhost port=5432 user=postgres password=your-password dbname=lenslocked_dev sslmode=disable"` which we will use to connect to our database using the `sql.Open` function. This function will return an `*sql.DB` if successful, and an error if there was some kind of issue, so we are going to panic if the error isn't `nil`.

One important thing to note is that if you are not using a password you should delete both the password constant and the password portion of the connection string we are creating. Keeping it can lead to issues.

Listing 8.8: Opening a database connection

```
db, err := sql.Open("postgres", psqlInfo)
if err != nil {
    panic(err)
}
```

After that we need to use the `Ping` method on the `sql.DB` type to ensure that our code actually tries to talk to the database. Again, we will be panicking if there is an error.

Listing 8.9: Pinging the database

```
err = db.Ping()
if err != nil {
    panic(err)
}
```

If everything went well we are going to print out the string `"Successfully connected!"` and then close our database connection by calling `db.Close()`.

The final code is shown in [Listing 8.10](#).

Listing 8.10: Connecting to Postgres with `database/sql`

```
package main

import (
    "database/sql"
    "fmt"

    _ "github.com/lib/pq"
)

const (
    host      = "localhost"
    port      = 5432
    user      = "postgres"
    password = "your-password"
    dbname   = "lenslocked_dev"
)

func main() {
    psqlInfo := fmt.Sprintf("host=%s port=%d user=%s "+
        "password=%s dbname=%s sslmode=disable",
        host, port, user, password, dbname)
    db, err := sql.Open("postgres", psqlInfo)
    if err != nil {
        panic(err)
    }
    err = db.Ping()
    if err != nil {
        panic(err)
    }

    fmt.Println("Successfully connected!")
    db.Close()
}
```

Once your code is ready, give it a try. Change to the `exp` directory and then build and run your code.

Listing 8.11: Running our code

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

You should see the output **Successfully connected!** in your terminal

which means everything is working!

Again, I want to reiterate that it is important that if you are not using a password you should delete both the password constant and the password portion of the connection string we are creating. Keeping it can lead to issues, but this code will still appear to run correctly.

If you instead see an error like **panic: pq: role "bad_user" does not exist** this means that your user is incorrect. You can either create a new user in Postgres or you will need to figure out a valid user based on your installation.

If you see an error like **panic: pq: database "lenslocked_dev" does not exist** this means that you didn't create the database. You can typically fix this by running **psql** and then running the SQL code in [Listing 8.12](#)

Listing 8.12: Create a database with SQL

```
CREATE DATABASE lenslocked_dev;
```

If your code is having errors with your host or port might take longer to run because it is attempting to contact a server that isn't responding. Generally speaking, your code should run pretty instantly unless you are connecting to a Postgres server hosted on another computer.

Source code

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

Completed - book.usegolang.com/8.3.1

Changes - book.usegolang.com/8.3.1-diff

8.3.2 Creating SQL tables to test with

Before we can insert records and query them we need to create a couple tables to insert records into.

Open up **psql** and create the following the following tables in Listing 8.13. We will be using these with the **database/sql** package and then we will drop them from our table before moving on to **github.com/jinzhu/gorm**, an ORM we will be using.

Listing 8.13: Creating SQL tables

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

CREATE TABLE orders (
    id SERIAL PRIMARY KEY,
    user_id INT NOT NULL,
    amount INT,
    description TEXT
);
```

We have two tables here, and we have a single relationship where users have many orders, or in other words, an order belongs to a user.

This relationship is expressed via the **user_id** field on the **orders** table. It is our way of keeping track of which user owns the order so that we can use this information in the future.

I have intentionally included this because I want you to understand how relationships work at the most basic level before we start using them with **github.com/jinzhu/gorm**.

8.3.3 Writing records with database/sql

We are going to continue coding inside of `exp/main.go` in this section as once again this code won't be used in our web application at all.

The first thing we are going to do is create a user record using the `DB.Exec()` method on our `db` object. We are going to also use a slightly modified SQL `INSERT INTO` statement where instead of putting values directly into our SQL, we will instead put `$1` and `$2` and pass those values as arguments to the `Exec()` method.

Listing 8.14: Inserting our first user

```
_ , err = db.Exec(`  
    INSERT INTO users(name, email)  
    VALUES($1, $2)`,  
    "Jon Calhoun", "jon@calhoun.io")  
if err != nil {  
    panic(err)  
}
```

It is **extremely important** that you take note of the `$1` and `$2` approach used here. We are NOT trying to build this string ourselves, but instead we are using placeholders like `$1` and we are letting the `database/sql` package handle creating the SQL statement.

By doing this we are allowing the `database/sql` package to prevent things like [SQL injection](#), a security vulnerability that allows attackers to execute arbitrary SQL on your database. This is incredibly bad because it could give attackers information about all of your users, or even allow them to delete all of the data in your database! [No bueno!](#)

Instead we use `$1` and `$2` and then the `database/sql` package will replace `$1` with the first argument passed in after the query string, and `$2` will be replaced with the second argument, and so on. While this is happening, the `database/sql` package will also run all of the necessary checks and escape

all of the necessary characters to ensure that an SQL injection attack does not occur.

Acquiring the ID of a new record

When we call `db.Exec(...)` we get an `sql.Result` and an error object returned. We would typically use the `LastInsertId()` method on the `sql.Result` object to get the ID of the record we just created, but unfortunately this doesn't work with Postgres.

Instead we need to modify our SQL query to tell our database to return the ID of the newly created record, and then we can use the `DB.QueryRow()` method instead of the `DB.Exec()` method so that we get a row back with the ID in the row's results.

This sounds confusing, but when we start querying for records it will all look pretty similar and the code isn't much more complicated.

Listing 8.15: Inserting and retrieving the ID of the new record

```
var id int
row := db.QueryRow(`  
    INSERT INTO users(name, email)  
    VALUES($1, $2) RETURNING id`,  
    "Jon Calhoun", "jon@calhoun.io")  
err = row.Scan(&id)  
if err != nil {  
    panic(err)  
}
```

The code in Listing 8.15 demonstrates both the updated SQL and using the `Row.Scan()` to retrieve the ID from the returned `sql.Row` object.

The additional SQL is `RETURNING id` which just tells the database to return the ID of the newly created record, and since we are using `QueryRow()` instead of `Exec()` we always get just an `sql.Row` object returned.

We then call the `Row.Scan()` on the row to tell our code where we would like to store the data retrieved from the database. We do this by passing in the address of an object that has a data type matching the data that was returned. In this case an integer was returned because the `id` field in our database is an integer, so we use a pointer to a Go integer to store the value.

If there is an error instead of getting it back as a return value from the `QueryRow()` method it will be returned when we call the `Scan()` method on the `sql.Row` object. That is why we have to check for the error when we write the code `err = row.Scan(&id)`.

Source code

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

Completed - book.usegolang.com/8.3.3

Changes - book.usegolang.com/8.3.3-diff

8.3.4 Querying a single record with database/sql

We are now going to look into querying records stored in our database using the `database/sql` package. The code here will have some similarities to the code in [Section 8.3.3](#) because we will be using the `QueryRow()` method again, but instead of only getting an ID we will be getting multiple fields for a record.

Our example is going to demonstrate how to query for a user with an ID of 1 and then put the id, name, and email address of that user into three variables named `id`, `name`, and `email`. The code for this is shown in [Listing 8.16](#).

Listing 8.16: Querying a user by ID

```
var id int
var name, email string
row := db.QueryRow(`SELECT id, name, email
                    FROM users
                    WHERE id=$1`, 1)
err = row.Scan(&id, &name, &email)
if err != nil {
    panic(err)
}
fmt.Println("ID:", id, "Name:", name, "Email:", email)
```

This is pretty similar to inserting a record, but because we are selecting three fields (`id`, `name`, and `email`) we will have three pieces of data to assign when we use the `row.Scan()` method.

We are also using the `$1` placeholder like before to make sure we don't have to worry about SQL injection attacks. In this case it is only being replaced by `1`, 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.

Box 8.3. No rows in result set error

If you get the error `sql: no rows in result set` this 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.

Source code

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

Completed - book.usegolang.com/8.3.4

Changes - book.usegolang.com/8.3.4-diff

8.3.5 Querying multiple records with database/sql

We are now going to look into querying multiple records stored in our database at the same time rather than just querying a single record. The code in this section is going to be incredibly similar to the code in [Section 8.3.4](#), especially the SQL, and you could even use this code to query a single record if you wanted.

The primary difference is going to be the fact that instead of working with a `sql.Row` object that just contains a single row, we will be interacting with a `sql.Rows` object. This is very similar to the `sql.Row` and it also has a `Scan()` method, but it also have a `Next()` method which will move on to the next row returned by the database.

If you haven't already, you may want to create a few user objects so that you can see how this works with multiple records, but it isn't mandatory to make it work.

Our example is going to demonstrate how to query for all users in our database with the email address `jon@calhoun.io` or an ID greater than 4. The results will come back as an `sql.Rows` object which we can iterate over. We will just be printing out the values, but you could insert each result into an array or really anywhere you want. The code is shown in [Listing 8.17](#)

Listing 8.17: Querying users by email address or ID

```
package main

import (
    "database/sql"
    "fmt"
    _ "github.com/lib/pq"
)
```

```

const (
    host      = "localhost"
    port      = 5432
    user      = "postgres"
    password  = "your-password"
    dbname    = "lenslocked_dev"
)

func main() {
    psqlInfo := fmt.Sprintf("host=%s port=%d user=%s "+
        "password=%s dbname=%s sslmode=disable",
        host, port, user, password, dbname)
    db, err := sql.Open("postgres", psqlInfo)
    if err != nil {
        panic(err)
    }

    var id int
    var name, email string
    rows, err := db.Query(`

        SELECT id, name, email
        FROM users
        WHERE email = $1
        OR id > $2`,
        "jon@calhoun.io", 3)
    if err != nil {
        panic(err)
    }

    for rows.Next() {
        rows.Scan(&id, &name, &email)
        fmt.Println("ID:", id, "Name:", name, "Email:", email)
    }
    db.Close()
}

```

The first half of Listing 8.17 is the same as our code was before, so we won't spend any time talking about it.

In the second half we start with declaring the same variables as before, and then we use the **db.Query()** method instead of the **db.QueryRow()** method. **db.Query()** has two return values - a ***sql.Rows** and an error.

If an error is returned this means we ran into some sort of error when trying to execute the SQL provided and we shouldn't bother trying to iterate over the

`sql.Rows` because we won't have anything useful there. For example, if we mistyped the table name we might get an error like `pq: relation "uzers" does not exist` and we wouldn't have any rows to look at because our query didn't run successfully.

Before we look at how to iterate over the `sql.Rows` object, I want to also note that once again we are using `$1` and `$2` as placeholders for data we want to insert into our query and then passing the data as a parameter to the `db.Query()` method. I know this has been repeated several times, but it really is important enough to justify multiple reminders.

Back to the `db.Query()` results - if the error object returned is `nil` this means that we are safe to start using the first return object to scan the data that was returned from our SQL query.

The `sql.Rows` object returned by `db.Query()` starts off pointing to a location just before the first row, so we must call `rows.Next()` before we can call `rows.Scan()`, but this actually works out in our favor because it makes it easier to write our for loop. Instead of needing to scan once and then start our loop we can instead call `rows.Next()` on the first pass of our loop and then access the first row of data.

After each call to `rows.Next()` we can use the `sql.Rows` object pretty much the same way we would use an `sql.Row` object. We use the `Scan()` method to retrieve the data from query and tell the method which parameters we want to store that data in. In this case we are scanning for an ID, name, and email address and storing them in the same variables as before.

Once we have the data we print it out and then let the for loop iterate to the next row by calling the `rows.Next()` method. When there isn't a new row to deal with the `rows.Next()` method will return `false` which will terminate our loop, but when there is a new row the call to `rows.Next()` tells our `rows` object that we are ready to move on to the next row. There isn't a way to go back after you call `rows.Next()`, so make sure that you don't call it before you are ready to move on!

8.3.6 Writing a relational record

In this section we are going to create an order for one of our existing users. The code here really isn't any different than creating records like we did before, so we aren't going to spend time talking about the code.

Instead you just need to run the code in Listing 8.18 so that you have a few orders in your database for the next section where we will be doing some relational queries.

Listing 8.18: Creating orders to query

```
package main

import (
    "database/sql"
    "fmt"

    _ "github.com/lib/pq"
)

const (
    host     = "localhost"
    port     = 5432
    user     = "postgres"
    password = "your-password"
    dbname   = "lenslocked_dev"
)

func main() {
    psqlInfo := fmt.Sprintf("host=%s port=%d user=%s "+
        "password=%s dbname=%s sslmode=disable",
        host, port, user, password, dbname)
    db, err := sql.Open("postgres", psqlInfo)
    if err != nil {
        panic(err)
    }

    var id int
    for i := 1; i < 6; i++ {
        // Create some fake data
        userId := 1
        if i > 3 {
            userId = 2
        }
        amount := 1000 * i
        _, err = db.Exec("INSERT INTO orders (user_id, amount) VALUES ($1, $2)", userId, amount)
        if err != nil {
            panic(err)
        }
    }
}
```

```

description := fmt.Sprintf("USB-C Adapter x%d", i)

err = db.QueryRow(`  

    INSERT INTO orders (user_id, amount, description)  

    VALUES ($1, $2, $3)  

    RETURNING id`,  

    userId, amount, description).Scan(&id)
if err != nil {
    panic(err)
}
fmt.Println("Created an order with the ID:", id)
}
db.Close()
}

```

After running this code you should be able to run **SELECT * FROM orders;** in **psql** and get something like Listing 8.19 where we have several orders, with a few associated to the user with the id 1, and a few associated to the user with the id 2.

Listing 8.19: Result from selecting all orders

id	user_id	amount	description
1	1	1000	USB-C Adapter x1
2	1	2000	USB-C Adapter x2
3	1	3000	USB-C Adapter x3
4	2	4000	USB-C Adapter x4
5	2	5000	USB-C Adapter x5

(5 rows)

Source code

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

Completed - book.usegolang.com/8.3.5

Changes - book.usegolang.com/8.3.5-diff

8.3.7 Querying related records

While building our application we are going to want to be able to query for related records at times. For example, we might want to look up users who have a name set and along with all of their orders.

To do this in SQL you would likely use a **JOIN**, so in this section we are going to look at an example of doing this with our Go code. (Hint: Not much changes!)

For demo purposes we are going to use a pretty contrived example where we are looking to get all orders and their corresponding user information in the same query. The SQL for this query is shown in Listing 8.20.

Listing 8.20: SQL to query both orders and their user

```
SELECT users.id, users.email, users.name,
       orders.id AS order_id,
       orders.amount AS order_amount,
       orders.description AS order_description
  FROM users
 INNER JOIN orders
    ON users.id = orders.user_id;
```

If you were to run the SQL in Listing 8.20 you would get output similar to Listing 8.21 where there is a row for each order with a user, and in each row we have both data for the user as well as the order.

Listing 8.21: Output for Listing 8.20

<code>id</code>	<code>email</code>	<code>name</code>	<code>order_id</code>	<code>order_amount</code>	<code>order_description</code>
1	<code>jon@calhoun.io</code>	Jon Calhoun	1	1000	USB-C Adapter x1
1	<code>jon@calhoun.io</code>	Jon Calhoun	2	2000	USB-C Adapter x2
1	<code>jon@calhoun.io</code>	Jon Calhoun	3	3000	USB-C Adapter x3
2	<code>bob@calhoun.io</code>	Bob Calhoun	4	4000	USB-C Adapter x4
2	<code>bob@calhoun.io</code>	Bob Calhoun	5	5000	USB-C Adapter x5

(5 rows)

Doing this with Go's `database/sql` package is actually pretty easy if you understand the SQL, and your code will be nearly identical to Section 8.3.5 but you will need to update the SQL you pass into `db.Query()` and then you will need to update your `rows.Scan()` method call to pass in variables for the additional data.

This is what makes the `database/sql` package powerful - if you already understand SQL it doesn't take much additional learning to get your code up and running with your database.

8.3.8 Delete the SQL tables we were testing with

We are now done using the `database/sql` package directly and will be trying out the `github.com/jinzhu/gorm` package. As a result, we no longer need the tables we created in this chapter and we can safely delete them by opening `psql` and running the SQL code in Listing 8.22.

Listing 8.22: Dropping the orders and users tables

```
DROP TABLE orders;
DROP TABLE users;
```

8.4 Using GORM to interact with a database

GORM is an [ORM](#), which stands for an object-relational mapping. ORMs are used to simplify the process of mapping data in one system to data in another.

In the case of SQL, GORM allows us to express what our data will look like in Go, and then it handles translating that into a format that our Postgres database understand. This is really handy when developing a new web application because it allows us to skip writing a translation layer manually.

GORM also provides a few helper functions that help simplify our code when working with an SQL database. For example, we don't need to write full SQL and instead can use some helper methods to write shorter queries.

In this section and the upcoming subsections we are going to look at how to install GORM, and then we will walk through replicating most of the things we did with raw SQL using GORM so you can see the difference between the two.

We will still be working in `exp/main.go`, and we will be reusing the constants with your database connection information, so be sure to keep those if you decide to clean up your code a bit.

8.4.1 Installing GORM and connecting to a database

The first thing we need to do is install GORM. We have done this several times in the book so far, so you should be pretty comfortable doing this.

Listing 8.23: Install `github.com/jinzhu/gorm`

```
go get -u github.com/jinzhu/gorm
```

Once you have GORM installed, connected to a database is very similar to what we were doing with the `database/sql` package, but there are two notable changes.

First, we are going to import the GORM dialects for Postgres instead of the `github.com/lib/pq` package.

```
import (
    _ "github.com/jinzhu/gorm/dialects/postgres"
)
```

Behind the scenes this will still load `github.com/lib/pq`, but it is a little

easier to remember and handles any other initializations that GORM might require to operate correctly with PostgreSQL. It also is a little clearer as to why we need it since we can will see GORM being used in this file

Along with the imports change, we are going to also update our code to use `gorm.Open()` to open up our database instead of calling `sql.Open()`. We will still use the same arguments, and the two behave very similarly, but `gorm.Open()` will return us a `*gorm.DB` instead of an `*sql.DB`.

Listing 8.24: Connecting to a database with GORM

```
package main

import (
    "fmt"

    "github.com/jinzhu/gorm"
    _ "github.com/lib/pq"
)

const (
    host      = "localhost"
    port      = 5432
    user      = "postgres"
    password = "your-password"
    dbname   = "lenslocked_dev"
)

func main() {
    psqlInfo := fmt.Sprintf("host=%s port=%d user=%s "+
        "password=%s dbname=%s sslmode=disable",
        host, port, user, password, dbname)
    db, err := gorm.Open("postgres", psqlInfo)
    if err != nil {
        panic(err)
    }
    defer db.Close()
}
```

8.4.2 Defining a GORM model

Up until now we have been defining what our database tables will have in them by writing SQL to create the table. With GORM we are going to try taking another approach - we are going to define a struct type in our code and then let GORM translate that into an SQL table and create the table for us.

First let's open up our experimental main.go source file.

```
$ atom exp/main.go
```

We are going to leverage the `gorm.Model` type which already has the basic fields that we will almost always want, like the unique ID for each resource and a timestamp for when the resources was created and updated.

We can also create custom fields. For example, we might want to add a field for a user's email address and name on top of the basic fields provided by `gorm.Model`. Listing 8.25 demonstrates how to create a struct that includes both the base `gorm.Model` as well as a few custom fields. Add the source code in the listing to your experimental main.go source file.

Listing 8.25: First GORM model

```
type User struct {
    gorm.Model
    Name  string
    Email string
}
```

In our previous users table we also had a unique constraint on the email field of a user. This is done with GORM by using tags, which are a form of metadata that can be looked up with the `reflect` package. If you have used the `encoding/json` package much in Go, you have likely already been introduced to tags, but even if you haven't used them before the only thing you

need to know is that these allow us to tell our code a little more about how we want different fields in our struct to be treated by GORM.

The code in [Listing 8.26](#) illustrates how to ensure that the `Email` field can't be null, and adds the unique constraint by adding a unique index to the field. This will tell our database to keep an index of every email stored in the users table and ensure that each entry in this index is unique. Update your code to reflect the listing.

Listing 8.26: Adding a unique index to users

```
type User struct {
    gorm.Model
    Name  string
    Email string `gorm:"not null;unique_index"`
}
```

By using an index we also get the added benefit that searching for users by email address will be more efficient because searching a sorted index is much more efficient than looking at every user in our database to see if any of them have the email address we are looking for.

GORM offers a wide variety of ways to declare a model, and all of them are covered in the docs³. You can check these out if you are curious, or if you need one that isn't ever used in this book, but we should cover most of the more common ones while creating our application.

Source code

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

Completed - book.usegolang.com/8.4.2

³<http://jinzhu.me/gorm/models.html#model-definition>

Changes - book.usegolang.com/8.4.2-diff

8.4.3 Creating and migrating tables with GORM

While we may have declared a type in our Go code, we have actually created a table in our database. To do this we are going to use the migrations features offered by GORM. These will allow us to update our models without actually writing SQL.

The first tool we are going to look at is the **AutoMigrate** function. This takes in instances of several types and creates corresponding tables in our SQL database based on the tags we set, the names of the fields and the type, and other factors. In Listing 8.27 our code will create a table named **users** because it is the snake case plural of **User**. Inside of that table it would have the fields **id**, **created_at**, **updated_at**, **deleted_at** from the embedded **gorm.Model**, and it would also have the **name**, and **email** fields. The last two are created by taking the attributes of the **User** type and turning them into snake case field names.

Listing 8.27: Automigrating with GORM

```
package main

import (
    "fmt"

    "github.com/jinzhu/gorm"
    _ "github.com/lib/pq"
)

const (
    host     = "localhost"
    port     = 5432
    user     = "postgres"
    password = "your-password"
    dbname   = "lenslocked_dev"
)

type User struct {
    gorm.Model
```

```

Name  string
Email string `gorm:"not null;unique_index"`
}

func main() {
    psqlInfo := fmt.Sprintf("host=%s port=%d user=%s "+
        "password=%s dbname=%s sslmode=disable",
        host, port, user, password, dbname)
    db, err := gorm.Open("postgres", psqlInfo)
    if err != nil {
        panic(err)
    }
    defer db.Close()

    db.AutoMigrate(&User{})
}

```

You can also use the `db.AutoMigrate` function to update models. For example, if we added the `Age int` attribute to the `User` type and then used GORM's auto-migrate we would get the `age` field added to our users table.

This makes migrating our database tables really easy, but it does have its limits. **AutoMigrate will only create things that don't already exist**, so if you already had a table named `users` it would not delete that table and attempt to make a new one. Likewise, it will not delete a column or replace it with a new type as these both have the potential to delete data unintentionally. Instead you will need to handle those types of migrations on your own.

Source code

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

Completed - book.usegolang.com/8.4.3

Changes - book.usegolang.com/8.4.3-diff

8.4.4 Logging with GORM

When using GORM it is often useful to enable logging so that you can track what SQL statements are being run behind the scenes. This is incredibly useful when things aren't acting the way you expect and you need to dig in and debug.

GORM's logging can be enabled by calling the **LogMode** method provided by the **gorm.DB** type. Open up your experimental main.go source file and we will enable it now.

```
$ atom exp/main.go
```

The **LogMode** method takes in a single argument - a boolean (true or false). It should be true when you want to enable logging, and false when you want to disable it. We will want to call this inside of our **main** function after the database connection is opened and we have checked for errors.

```
func main() {
    psqlInfo := fmt.Sprintf("host=%s port=%d user=%s "+
        "password=%s dbname=%s sslmode=disable",
        host, port, user, password, dbname)
    db, err := gorm.Open("postgres", psqlInfo)
    if err != nil {
        panic(err)
    }
    defer db.Close()
    db.LogMode(true)

    db.AutoMigrate(&User{})
}
```

With logging enabled we will start to see output like the following when SQL statements are run.

```
(/Users/jon/go/src/lenslocked.com/exp/main.go:35)
[2017-07-18 17:04:12] [6.85ms] CREATE TABLE "users" ...

(/Users/jon/go/src/lenslocked.com/exp/main.go:35)
[2017-07-18 17:04:12] [1.18ms] CREATE INDEX ...

(/Users/jon/go/src/lenslocked.com/exp/main.go:35)
[2017-07-18 17:04:12] [1.00ms] CREATE UNIQUE INDEX ...
```

You will only see output like this when SQL statements are actually run, so if you have already created your users table you won't see this output. If you would like to, you can drop the users table and then rerun your code to see output like above.

Source code

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

Completed - book.usegolang.com/8.4.4

Changes - book.usegolang.com/8.4.4-diff

8.4.5 Creating a record with GORM

Taking everything from the past few sections, we are going to write a program that will ask a user for their name and email address and then create a user record in our database with that information.

The code in Listing 8.28 demonstrates how to create a new record with GORM. We first create a `User` instance and fill in the `Name` and `Email` attributes. We do not need to set the ID or any of the attributes provided by `gorm.Model` because these are going to be filled in automatically by GORM.

Once we have the user object we call `db.Create()` on the `gorm.DB` instance. The `db.Create()` method returns a pointer to a `gorm.DB` object, so we can then grab the `Error` attribute to verify that the create happened successfully, and then check to see if the error is nil before proceeding.

Listing 8.28: Create a new record with GORM

```
name, email := // get the name and email from the user
u := &User{
    Name: name,
    Email: email,
}
if err = db.Create(u).Error; err != nil {
    panic(err)
}
fmt.Printf("%+v\n", u)
```

If we don't use a unique email address every time we run our program you will eventually get an error like `pq: duplicate key value violates unique constraint "uix_users_email"` which means that you are trying to write an email address that already exists in our database.

To counter this, it would be nice to have a function that will ask the user for a name and email address before creating the user object. In Listing 8.29 I have put all of the code from the last few sections together; I have also provided a `getInfo()` function which will ask the user for their name and email address, but I won't be going over the details of what is going on here. Just be aware that it will ask you for your name and email address every time you run the program and will use those values to create a new user.

```
$ atom exp/main.go
```

Listing 8.29: Creating a dynamic record with GORM

```
package main
```

```

import (
    "bufio"
    "fmt"
    "os"

    "github.com/jinzhu/gorm"
    _ "github.com/lib/pq"
)

const (
    host      = "localhost"
    port      = 5432
    user      = "postgres"
    password  = "your-password"
    dbname    = "lenslocked_dev"
)

type User struct {
    gorm.Model
    Name string
    Email string `gorm:"not null;unique_index"`
}

func main() {
    psqlInfo := fmt.Sprintf("host=%s port=%d user=%s "+
        "password=%s dbname=%s sslmode=disable",
        host, port, user, password, dbname)
    db, err := gorm.Open("postgres", psqlInfo)
    if err != nil {
        panic(err)
    }
    defer db.Close()
    db.LogMode(true)

    db.AutoMigrate(&User{})

    name, email := getInfo()
    u := &User{
        Name:  name,
        Email: email,
    }
    if err = db.Create(u).Error; err != nil {
        panic(err)
    }
    fmt.Printf("%+v\n", u)
}

func getInfo() (name, email string) {
    reader := bufio.NewReader(os.Stdin)
    fmt.Println("What is your name?")
    name, _ = reader.ReadString('\n')
}

```

```
name = strings.TrimSpace(name)
fmt.Println("What is your email?")
email, _ = reader.ReadString('\n')
email = strings.TrimSpace(email)
return name, email
}
```

Run the code in Listing 8.29 a few times to create a few users in your database. We will be using these in the next section when we start to query our database for specific users.

```
$ go run exp/main.go
```

Source code

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

Completed - book.usegolang.com/8.4.5

Changes - book.usegolang.com/8.4.5-diff

8.4.6 Querying a single record with GORM

GORM offers a wide variety of ways to query for records, but they all are essentially just helpers for writing SQL, so they are pretty easy to get the hang of.

Box 8.4. You may want to enable logging here

In [Section 8.4.4](#) we discussed how to enable logging with GORM which causes it to start outputting logs with information about every SQL query it attempts to execute.

It is helpful to enable logging in this section as it will help clarify what each piece of code is doing to see the SQL that it generates. As a result, I suggest you enable logging with GORM for the rest of this section and testing out each example to see what SQL is being executed.

In general I prefer to keep this enabled in all development environments so that I can easily debug things.

The first example we are going to look at is one that uses the [DB.First\(\)](#) method to tell our database that we want to retrieve the first record. This command can be used in conjunction with several other queries with GORM, but it doesn't have to be.

This is much easier to understand after looking at some examples, so we are going to jump right into an example.

Listing 8.30: Querying for the first user

```
var u User

db.First(&u)
if db.Error != nil {
    panic(db.Error)
}
fmt.Println(u)
```

In [Listing 8.30](#) we are telling GORM that we would like the first user in our database to be retrieved and stored in the `u` object. By passing in an address to a user object, GORM knows to query the users table, and knows to put the resulting record into the object passed in as the first argument.

If there was an error it would be present on our `db` object, so we need to check to see if `db.Error` has a value before using our user object. We will be doing this with every example moving forward, and it is a general pattern you can follow when using GORM. Instead of returning errors it frequently attaches them to the `db` object.

Box 8.5. Where did the extra SQL come from?

If you have enabled logging you might be asking yourself “Where did all of the extra code in my SQL query come from?”. We didn’t write anything about the `deleted_at` field being NULL in our query, and we definitely didn’t specify an order for the SQL query.

By default, GORM assumes that you use the `deleted_at` field to mark a record as deleted rather than actually deleting the record from your database. As a result, it will ignore any record that has a value set for this field by default.

GORM also sets a default order for queries to help provide a little more consistency with our queries. As a result, when you call the `First()` method you can rest assured that it will always return the record with the lowest ID unless your query specifies a different order to sort the results.

Listing 8.31: Querying for the first user with `id=1`

```
var u User
id := 1

db.First(&u, id)
if db.Error != nil {
    panic(db.Error)
}
fmt.Println(u)
```

Our second example in Listing 8.31 is very similar to our first except we are

now providing a second argument to the `First()` method - an ID.

GORM allows us to provide an ID as the second argument to the `First()` method as a handy way of querying for objects with a specific ID without writing out a full `WHERE` statement. We will end up using this a good bit in our code.

Listing 8.32: Querying with the `Where()` method

```
var u User
maxId := 3

db.Where("id <= ?", maxId).First(&u)
if db.Error != nil {
    panic(db.Error)
}
fmt.Println(u)
```

[Listing 8.32](#), our third example, demonstrates how to chain the `Where()` method along with the `First()` method to create a `WHERE` query. In this example we are looking for any user whose ID is less than or equal to the provided `id`, which happens to be 3 in this case.

GORM offers several helper methods like `Where()` that can be chained together to create more complex queries, and you can find examples of most of them in the [GORM querying documentation](#).

When using these queries, you might have noticed that we use the `?` character instead of `$1` and `$2` like we were using with the `database/sql` package. They end up doing the same thing, but because we don't have numbered parameters we need to include the same argument multiple times if we intend to use it in multiple spots in our query.

Listing 8.33: Querying with an existing user

```
var u User
u.Email = "jon@calhoun.io"
```

```
db.Where(u).First(&u)
if db.Error != nil {
    panic(db.Error)
}
fmt.Println(u)
```

In addition to querying with strings, you can also query by passing a model that matches what you are looking for to the `Where()` method. In Listing 8.33 we are searching for a user with the email address “`jon@calhoun.io`” by setting the email address on our `User` object and then passing that object into our `Where()` method call.

This can be useful for building a more complex query because you can have different parts of your code set different attributes of the user object before finally executing your query, but it can also be confusing at times because an attribute you weren’t expecting to be set could end up affecting your query.

There are a lot of other ways to query with GORM, and we will be covering several of them throughout this book, but rather than learning them all upfront I suggest you wait until you need a new query and then check out the GORM docs to figure out how to create what you need. This allows you to be productive immediately, and also ensures you are only learning things you will actually use in practice.

Source code

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

Completed - book.usegolang.com/8.4.6

Changes - book.usegolang.com/8.4.6-diff

8.4.7 Querying multiple records with GORM

Querying for multiple records with GORM is nearly identical to querying for a single records, except we will be using the `Find()` method instead of the `First()` method, and we will be passing in a slice of users instead of a single user.

Listing 8.34: Querying for multiple users

```
var users []User
db.Find(&users)
if db.Error != nil {
    panic(db.Error)
}
fmt.Println("Retrieved", len(users), "users.")
fmt.Println(users)
```

Behind the scenes the major difference between `Find()` and `First()` is that `First()` will automatically add a `LIMIT 1` to the query because it is aware that we only want one item returned.

Source code

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

Completed - book.usegolang.com/8.4.7

Changes - book.usegolang.com/8.4.7-diff

8.4.8 Creating related models with GORM

When we were learning about the `database/sql` package before we created an additional table, the orders table. In this section we are going to recreate that

table using GORM and fill it with some fake data that we will use in the next section to demonstrate how to query for related records using GORM.

The first thing we need to do is create the `Order` type. Doing this will be very similar to our `User` type; We will be embedding the `gorm.Model` type and declaring other fields we need inside of the type.

```
type Order struct {
    gorm.Model
    UserID      uint
    Amount      int
    Description string
}
```

The only part in this code that might warrant a second glance is the `UserID` being of the type `uint`, which is an unsigned integer. We are doing this because we are never using negative IDs in our database, so we can reflect that in our Go code by using the `uint` type instead of `int`.

We will be creating a few orders, so we are going to write a function to make this a little faster. We will simply pass the data needed to create an order into the function and let it handle checking for errors and saving to the database.

```
func createOrder(db *gorm.DB, user User, amount int, desc string) {
    db.Create(&Order{
        UserID:      user.ID,
        Amount:      amount,
        Description: desc,
    })
    if db.Error != nil {
        panic(db.Error)
    }
}
```

With these two pieces we are ready to put it all together and look up a user, then create several orders for that user. The final code can be found in [Listing 8.35](#). You should update your `exp/main.go` file with this code and then run it to create a few orders to use in the next section.

Listing 8.35: Creating the orders table

```
package main

import (
    "fmt"

    "github.com/jinzhu/gorm"
    _ "github.com/lib/pq"
)

const (
    host      = "localhost"
    port      = 5432
    user      = "postgres"
    password = "your-password"
    dbname   = "lenslocked_dev"
)

type User struct {
    gorm.Model
    Name string
    Email string `gorm:"not null;unique_index"`
}

type Order struct {
    gorm.Model
    UserID     uint
    Amount     int
    Description string
}

func main() {
    psqlInfo := fmt.Sprintf("host=%s port=%d user=%s "+
        "password=%s dbname=%s sslmode=disable",
        host, port, user, password, dbname)
    db, err := gorm.Open("postgres", psqlInfo)
    if err != nil {
        panic(err)
    }
    defer db.Close()

    db.LogMode(true)
    db.AutoMigrate(&User{}, &Order{})

    var user User
    db.First(&user)
    if db.Error != nil {
        panic(db.Error)
    }
}
```

```

createOrder(db, user, 1001, "Fake Description #1")
createOrder(db, user, 9999, "Fake Description #2")
createOrder(db, user, 8800, "Fake Description #3")
}

func createOrder(db *gorm.DB, user User, amount int, desc string) {
    db.Create(&Order{
        UserID:      user.ID,
        Amount:      amount,
        Description: desc,
    })
    if db.Error != nil {
        panic(db.Error)
    }
}

```

Source code

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

Completed - book.usegolang.com/8.4.8

Changes - book.usegolang.com/8.4.8-diff

8.4.9 Querying relational data with GORM

We now want to start querying for related data using GORM. Specifically, we would like to be able to look up a user and load the orders linked to that user at the same time. To do this we need to update our **User** type to inform GORM that our user type is related to the **Order** type. We need to add the attribute **Orders []Order** to the **User** type to make this happen.

```

type User struct {
    gorm.Model
    Name  string
    Email string `gorm:"not null;unique_index"`
}

```

```
    Orders []Order
}
```

After that we simply need to inform GORM that we want to preload the orders for a user we are looking up. To do this we use the `Preload()` method on the `gorm.DB` type.

Listing 8.36: Preloading user orders

```
package main

import (
    "fmt"

    "github.com/jinzhu/gorm"
    _ "github.com/lib/pq"
)

const (
    host      = "localhost"
    port      = 5432
    user      = "postgres"
    password = "your-password"
    dbname   = "lenslocked_dev"
)

type User struct {
    gorm.Model
    Name  string
    Email string `gorm:"not null;unique_index"`
    Orders []Order
}

type Order struct {
    gorm.Model
    UserID     uint
    Amount     int
    Description string
}

func main() {
    psqlInfo := fmt.Sprintf("host=%s port=%d user=%s "+
        "password=%s dbname=%s sslmode=disable",
        host, port, user, password, dbname)
    db, err := gorm.Open("postgres", psqlInfo)
    if err != nil {
```

```
    panic(err)
}
defer db.Close()

db.LogMode(true)
db.AutoMigrate(&User{}, &Order{})

var user User
db.Preload("Orders").First(&user)
if db.Error != nil {
    panic(db.Error)
}
fmt.Println("Email:", user.Email)
fmt.Println("Number of orders:", len(user.Orders))
fmt.Println("Orders:", user.Orders)
}
```

The code in Listing 8.36 will also work with the `Find()` method and will preload the orders for every user that ends up being returned in the query. This is useful for when you want to find a specific subset of data.

Source code

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

Completed - book.usegolang.com/8.4.9

Changes - book.usegolang.com/8.4.9-diff

8.5 Exercises

This chapter covered a boatload of material, but most of it was meant to give you background information so that you have an understanding of what is going on behind the scenes when we use GORM.

As a result, the exercises are going to focus mostly on GORM, but that doesn't mean you should forget about or ignore the other sections of this chapter. It is important to have an understanding of the SQL being executed behind the scenes when you use GORM so you can make educated decisions. For example, having an understanding of the underlying SQL queries used most often in your application will make it easier to determine which fields need to be indexed and which do not.

8.5.1 Ex1 - What changes won't the AutoMigrate function provided by GORM handle for you?

GORM's **AutoMigrate** function will handle a lot for you, but there are a few things it will not handle. Take a minute to make sure you understand what it will and will not handle.

8.5.2 Ex2 - What is **gorm.Model** used for?

What fields are provided by the **gorm.Model**? What is each field used for?

8.5.3 Ex3 - Experiment using a few more GORM methods.

Go to the docs for GORM:

- <http://jinzhu.me/gorm/>

Then take a few moments to experiment with different ways to query for records. Experiment with things like **Not()** and other similar functions.

8.5.4 Ex4 - Experiment with query chaining

GORM's method chaining is how more complicated queries are created when using GORM. Most of these are designed to look and feel very similar to SQL, so if you have any experience using SQL you should be able to read the queries with ease.

Try chaining a few of the querying methods provided by GORM together. Try to predict what the resulting SQL will be before running the query, and then run your code with logging enabled. Did it match your expectations?

8.5.5 Ex5 - Learn to execute raw SQL with GORM

While GORM can be helpful, there will often be times when writing raw SQL queries is the best route for creating a complicated query. Take a few minutes to learn how to execute raw SQL queries using GORM.

- <http://jinzhu.me/gorm/advanced.html#sql-builder>

Chapter 9

Creating the user model

In this chapter we are going to get back to our web application where we will be taking what we learned about databases in the last chapter and incorporating it into our model layer.

Throughout this chapter we are going to be writing code that will eventually be a part of our application, but we won't actually be tying it into our controllers until next chapter.

9.1 Defining a **User** type

The first thing we need to do is define what our user resource. To define the resource we need to decide what data we want to store with each user object.

While it might seem imperative to get this right on the first try, very few developers will actually achieve this. The truth is, it is nearly impossible to predict how your requirements will change over time, and as your requirements change so will the structure of your resources.

Rather than spending a long period of time contemplating how to design our

user resource, we are instead going to pick a pretty basic set of attributes to store on the user resource, and then over time we will update our user resource. For example, this first version won't have any reference to a password, but we will eventually need to store something so that we can verify a user is authorized to log into an account.

The first version of our user resource is going to consist of the following data:

- **id** - a unique identifier, represented as a positive integer
- **name** - the user's full name
- **email** - the user's email address
- **created_at** - the date that the user account was created
- **updated_at** - the date that the user account was last updated
- **deleted_at** - the date that the user account was deleted

The **deleted_at** attribute might seem funny, but we are going to use it to "delete" accounts without actually removing them from our database. This is useful because a user might have their account hacked and deleted and ask us to recover it a few days later. By using a **deleted_at** attribute we are able to pretend like an account is deleted for a set time period (maybe a week or two) and then permanently delete the user resource after that two weeks have passed.

GORM is designed to ignore models with a **deleted_at** attribute by default, so we won't have to customize our code at all to make this work.

Create a folder named **models** and then create the file **users.go** in it so that we can start by creating our user type.

Listing 9.1: Create the users model file

```
$ mkdir models
$ atom models/users.go
```

All of our files in the **models** directory are going to be a part of the models package, so we will start our file off by stating the package. After that we will import GORM and then create a new type like we did in [Section 8.4.2](#).

Box 9.1. You may need to reset your database

If you were following along and running the code from the last chapter you likely already have an orders and users table in your database. It is a good idea to reset your database at this point and start from scratch to avoid any issues. The easiest way to do this is to connect to Postgres then drop your database and recreate it.

First open up Postgres. You may need to use a username and password depending on how you installed Postgres.

```
$ psql
```

Next we want to drop the table and recreate it. Again, you might need to follow slightly different steps depending on how you initially created your table.

```
DROP DATABASE lenslocked_dev;
CREATE DATABASE lenslocked_dev;
```

If all went according to plan you should now have a fresh database to work with. Just make sure you DO NOT run any of the experimental code from before anymore otherwise it could alter your existing database. Alternatively, you could update your experimental code to use a different database such as “lenslocked_exp”.

If you recall from before, the **gorm.Model** type will include the **id**, **created_at**, **updated_at**, and **created_at** fields for us, so the only ones we need to man-

ually declare for now are **Name** and **Email**.

We are also going to add in a tag for our **Email** field to let GORM know that we don't want this field to be null, and that we want to index it with a unique constraint. This will ensure that all of our users have an email address, and that no two users share the same email address.

Listing 9.2: Creating the **User** type

```
package models

import "github.com/jinzhu/gorm"

type User struct {
    gorm.Model
    Name  string
    Email string `gorm:"not null;unique_index"`
}
```

Now that we have a data type available, let's look at how we can write some code to make interacting with the database easier.

Source code

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

Completed - book.usegolang.com/9.1

Changes - book.usegolang.com/9.1-diff

9.2 Creating the **UserService** interface and querying for users

In [Section 8.4.5](#) we explored using GORM to write records to our SQL database, but we left out a lot of details at the time to make things simpler. As a result, we were able to interact directly with the a **gorm.DB** object and demonstrate how GORM can be used to interact with a database, but we won't to interact with GORM directly in most of our application.

For starters, we might eventually want to change our application to use the **database/sql** package, or even another database altogether. If we were to scatter the **gorm.DB** object across all of our code it would be very hard to change in the future. It would also be very hard to test our application without setting up a real database, and we would end up with a lot of duplicate code.

Rather than using GORM everywhere in our code, we are instead going to write a layer of abstraction that defines what methods our controllers will be using, but hides those implementation details. This will allow us to change our database implementation details in the future without needing to update any controller or view code.

Box 9.2. What is an abstraction layer?

In programming, an abstraction layer is a way of hiding implementation details.

Take a vehicle for example - when you put your foot on the gas pedal your expectation is that the engine will provide power to the wheels and that your car will move. The actual implementation details of this are irrelevant. You don't care if your car is electric, gas, or a hybrid; as long as the gas pedal still causes the engine to provide power and accelerate the vehicle you should still be able to drive it.

Abstraction layers in programming are very similar. Most of your code won't actually care how data is saved or what database is being used. All that will matter is

that if it calls the **Save** function it expects the data provided to be saved. By writing code this way, we can avoid writing everything all at once and instead write more modular pieces one at a time.

The abstraction layer for our users database is going to be called the **UserService**, and it is going to be a type that provides methods for querying, creating, and updating users. We know this layer is going to need access to a **gorm.DB** pointer, so we can start there when defining it.

```
$ atom models/users.go
```

```
type UserService struct {
    db *gorm.DB
}
```

We know we need to open a connection to our database with GORM, so our next step will be to write a function to do this and then setup and return a **UserService** object. When writing code that creates a type, I typically name the function **NewXXX** where “XXX” is the type it instantiates, so we will name our new function **NewUserService**.

Inside of this function we will be writing code nearly identical to the code we wrote in the last chapter when connecting to our database with GORM, but we will expect the end user to pass us a string defining how we should connect to the database. We also need to add the Postgres dialect to our imports. You need to do these even if you are using an auto-import tool because this package isn’t used directly, but loads the Postgres driver for us.

```
import (
    "github.com/jinzhu/gorm"
    _ "github.com/jinzhu/gorm/dialects/postgres"
```

```
)
// ...

func NewUserService(connectionInfo string) (*UserService, error) {
    db, err := gorm.Open("postgres", connectionInfo)
    if err != nil {
        return nil, err
    }
    db.LogMode(true)
    return &UserService{
        db: db,
    }, nil
}
```

Box 9.3. Why aren't we defering **db.Close()**?

Deferred function calls are executed when a function exits, so if were to defer closing the DB inside of our **NewUserService** function it would end up closing the database connection right before returning the new user service. Our user service wouldn't be very useful without an active database connection, so this is likely a bad idea.

Since we aren't closing the database connection right when we open it, it is a good idea to also write a method that will let your **UserService** users close the connection when they are done with it.

```
// Closes the UserService database connection
func (us *UserService) Close() error {
    return us.db.Close()
}
```

Finally, we are going to add two methods to our user service. The first is going to be a method that allows us to retrieve a user from the database using the ID of the user, and we will name the method **ByID**. The second is a function we won't

use right away, but will allow us to destroy our existing tables and then recreate them. We will name this function **DestructiveReset**, and while it won't be very useful in a production environment, it will prove useful in a development environment when we want a way to quickly and easily reset our database.

Let's start with **ByID**, which will return a pointer to a user or an error. Errors will vary from a simple, "we couldn't find a user with that ID" to a variety of other errors that the GORM package might return. For instance, if our database crashes for some reason we will likely get an error back and won't be able to return a user.

Most of the errors will signify that there is an underlying problem with our database or our connection to it and we will simply return the errors as they occur, but in the case of not finding a record we want to return a custom error. We will do this so that any code using the **UserService** can differentiate between not finding a record and some other issue occurring behind the scenes. We are going to name this error **ErrNotFound** and will start by writing the code for it.

```
import (
    "errors"

    "github.com/jinzhu/gorm"
    _ "github.com/jinzhu/gorm/dialects/postgres"
)

var (
    // ErrNotFound is returned when a resource cannot be found
    // in the database.
    ErrNotFound = errors.New("models: resource not found")
)
```

Next we will write the **ByID** method.

```
// ByID will look up a user with the provided ID.
// If the user is found, we will return a nil error
// If the user is not found, we will return ErrNotFound
// If there is another error, we will return an error with
// more information about what went wrong. This may not be
// an error generated by the models package.
```

```
//  
// As a general rule, any error but ErrNotFound should  
// probably result in a 500 error.  
func (us *UserService) ByID(id uint) (*User, error) {  
    var user User  
    err := us.db.Where("id = ?", id).First(&user).Error  
    switch err {  
        case nil:  
            return &user, nil  
        case gorm.ErrRecordNotFound:  
            return nil, ErrNotFound  
        default:  
            return nil, err  
    }  
}
```

A lot of the code here should look familiar. For example, we used the **Where** method on a GORM database connection in the last chapter. What we might not have done last chapter was access the **Error** field of the **gorm.DB** object, which is where any errors encountered during a GORM query are placed. We do this to make sure we capture any errors that occur while querying our database.

Once we have the error we write a **switch** statement where we try to determine what to return based on the error. If there isn't an error its value will be nil, so we can simply return a pointer to the user we queried with. On the other hand, if there is an error we don't want to return a user and will instead return an error.

In this situation we first look for GORM's **ErrRecordNotFound**¹ which signifies that our query executed correctly, but we couldn't find a user with the provided query. While we could technically return this error directly, then our controllers would need to know to look for this GORM specific error. Instead we opt to return our own **ErrNotFound** error so that users of our models package don't need to know anything about GORM.

If the error is not a record not found error we will simply return the error and nil for the user. This is shown in the **default** block of the switch statement.

The last piece of code we need to write is the **DestructiveReset** method.

¹<https://godoc.org/github.com/jinzhu/gorm#pkg-variables>

This will do two things:

1. Call the **DropTableIfExists**² method on the users table.
2. Rebuild the users table using the **AutoMigrate** function.

```
// DestructiveReset drops the user table and rebuilds it
func (us *UserService) DestructiveReset() {
    us.db.DropTableIfExists(&User{})
    us.db.AutoMigrate(&User{})
}
```

Again, we won't use this reset function in production, but it will be valuable when working in development environments.

If you would like to see how the code we wrote is used, or if you simply want to test your code, you can find an updated copy of **exp/main.go** in [Listing 9.3](#).

Listing 9.3: Demoing our UserService

```
package main

import (
    "fmt"

    "lenslocked.com/models"

    _ "github.com/lib/pq"
)

const (
    host      = "localhost"
    port      = 5432
    user      = "postgres"
    password  = "your-password"
    dbname    = "lenslocked_dev"
)

func main() {
```

²<https://godoc.org/github.com/jinzhu/gorm#DB.DropTableIfExists>

```
psqlInfo := fmt.Sprintf("host=%s port=%d user=%s "+
    "password=%s dbname=%s sslmode=disable",
    host, port, user, password, dbname)
us, err := models.NewUserService(psqlInfo)
if err != nil {
    panic(err)
}
defer us.Close()
us.DestructiveReset()

// This will error because you DO NOT have a user with
// this ID, but we will create one soon.
user, err := us.ByID(1)
if err != nil {
    panic(err)
}
fmt.Println(user)
```

Source code

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

Completed - book.usegolang.com/9.2

Changes - book.usegolang.com/9.2-diff

9.3 Creating users

If you ran the source code for `exp/main.go` in the last section, you probably noticed that it isn't very useful yet. We don't have any users, and we haven't written a method to create them, so trying to look one up is a pretty pointless.

In this section we are going to write a `Create` method on our user service that will accept a pointer to a user object, save it to the database, backfill data like the ID and creation date, and then return. If our create method has an error

it will return an error, otherwise it will return nil. It doesn't need to return a user object because it accepts a pointer to one and can instead just update the provided user.

This will all go inside of the users model.

```
$ atom models/users.go
```

```
// Create will create the provided user and backfill data
// like the ID, CreatedAt, and UpdatedAt fields.
func (us *UserService) Create(user *User) error {
    return us.db.Create(user).Error
}
```

Now let's test it out. Open up your experimental source file where we will create a user.

```
$ atom exp/main.go
```

```
// Create a user
user := models.User{
    Name: "Michael Scott",
    Email: "michael@dundermifflin.com",
}
if err := us.Create(&user); err != nil {
    panic(err)
}

// NOTE: You may need to update the query code a bit as well
foundUser, err := us.ByID(1)
if err != nil {
    panic(err)
}
fmt.Println(foundUser)
```

If you haven't already, you will also need to call the **DestructiveReset** method as it is what sets up the users table for the first time. We will eventually provide another way to do this, but for now it does the trick.

Source code

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

Completed - book.usegolang.com/9.3

Changes - book.usegolang.com/9.3-diff

9.4 Querying by email and DRYing up our code

Typically, I wouldn't recommend adding more methods to a service than you immediately need. Doing so is an easy way to end up with a "fat" service that has several legacy methods that rarely (if ever) get used, yet they end up needing supported because everyone is afraid of removing them. Not to mention the fact that nobody really needed to write them in the first place.

Despite that, I know for a fact that we are going to need a method to query for users by their email address in the near future because we will need this in order to let users log in, so we are going to go ahead and write that method next. While we are at it, we are going to DRY up our code a bit so that we can avoid writing the same bits of code too frequently.

Note: DRY is an acronym discussed in [Section 7.4.5](#), but is often used as a verb meaning to refactor your code in order to make it less repetitive.

We will start by refactoring our **ByID** method. Specifically, both the call to the **First** method (followed by a call to the **Error** field) used to retrieve the first result, as well as the following switch statement we wrote before are both things we are likely to repeat in the future. In fact, we know that if we are searching for a user by ID, email address, or really by any individual field that we will only want to retrieve the first one. We also know that in those situations we are likely going to want to return our own **ErrNotFound** error when a record isn't

found.

To avoid having to repeat this code, we are going to move it to its own function which we will call **first**. We also won't be exporting this function as this is only used internally. Open up the users model source file.

```
$ atom models/users.go
```

Next we write the **first** function. In this particular case we aren't using a switch statement like before, but the resulting code is very similar. We are also opting to use an **interface{}** for our type so that this function can be used with other models in the future, reducing the amount of code we need to write further.

```
// first will query using the provided gorm.DB and it will
// get the first item returned and place it into dst. If
// nothing is found in the query, it will return ErrNotFound
func first(db *gorm.DB, dst interface{}) error {
    err := db.First(dst).Error
    if err == gorm.ErrRecordNotFound {
        return ErrNotFound
    }
    return err
}
```

Now we need to update the **ByID** function to utilize our new function. To do this we are going to:

1. Remove the switch statement.
2. Create the initial query using GORM, and then save it to a variable which can be passed into **first**. Notice that we DO NOT call the **First** method on this query because this will be handled by our new **first** function.

3. Pass the DB query and a pointer to the user to our **first** function and save any errors returned to a variable.
4. Return the user if there aren't any errors. Otherwise only return the error.

Listing 9.4: Using the new **first** function in **ByID**

```
func (us *UserService) ByID(id uint) (*User, error) {
    var user User
    db := us.db.Where("id = ?", id)
    err := first(db, &user)
    if err != nil {
        return nil, err
    }
    return &user, nil
}
```

Box 9.4. Is the if statement required here?

Technically we could return both the **user** and **err** variables in our code in Listing 9.4 and everything would still work. The only real difference is that when an error does occur the user pointer that we return won't be nil but will instead point to a zero-value user.

Even though the code we wrote is longer, I tend to write code this way because it helps prevent bugs. For example, someone might not fully read our documentation and could write code that only checks to see if the returned user is nil before proceeding. In those situations they would have buggy code because the zero-value user we would be returning IS NOT nil.

```
// An example of potentially buggy code - DO NOT code this
user, _ := us.ByID(123)
if user == nil {
    panic("error getting the user")
}
// Otherwise proceed with our user even though there could
// be an error!
```

While the bug we are preventing is not one we are directly causing, and it is a mistake on another developer's part to not check for errors, sometimes making your API more accident-proof can be useful.

Next we will write a method to query users by their email address. We will call this **ByEmail**, and it will function pretty much identically to **ByID**. The only real difference will be the query we pass into the **Where** method.

```
// ByEmail looks up a user with the given email address and
// returns that user.
// If the user is found, we will return a nil error
// If the user is not found, we will return ErrNotFound
// If there is another error, we will return an error with
// more information about what went wrong. This may not be
// an error generated by the models package.
func (us *UserService) ByEmail(email string) (*User, error) {
    var user User
    db := us.db.Where("email = ?", email)
    err := first(db, &user)
    return &user, err
}
```

Finally, we will update our experimental **main.go** source file to test out our new method.

```
$ atom exp/main.go
```

```
// Update the call to ByID to instead be ByEmail
foundUser, err := us.ByEmail("michael@dundermifflin.com")
if err != nil {
    panic(err)
}
fmt.Println(foundUser)
```

When you run this code with logging enabled you should see an SQL query like the one being logged just before it is run by GORM.

```
$ go run exp/main.go
```

Note: The logs won't have newlines between them like the SQL below, but they should have the same query.

```
SELECT * FROM "users"
WHERE "users"."deleted_at" IS NULL
AND ((email = 'michael@dundermifflin.com'))
ORDER BY "users"."id" ASC
LIMIT 1
```

Source code

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

Completed - book.usegolang.com/9.4

Changes - book.usegolang.com/9.4-diff

9.5 Updating and deleting users

In addition to needing a way to query users, we are also going to need a way to delete and update users. Updating a user is pretty simple - we just need a user object with all of the new data passed in and we can tell GORM to save the new model to the database.

```
$ atom models/users.go
```

```
// Update will update the provided user with all of the data
// in the provided user object.
func (us *UserService) Update(user *User) error {
    return us.db.Save(user).Error
}
```

Deleting a user is a little trickier, because GORM's delete method works in one of two ways. The first is what we would normally expect. If we provide GORM with a user object that has an ID GORM will only delete a user with the same primary key.

The second is a bit trickier. If we DO NOT provide an ID (eg it is **0**), then GORM will delete ALL users. We aren't ever going to do this, so we are going to write a bit of code to prevent this from accidentally happening. We will start with an error for when the ID provided is invalid.

```
var (
    // ...

    // ErrInvalidID is returned when an invalid ID is provided
    // to a method like Delete.
    ErrInvalidID = errors.New("models: ID provided was invalid")
)
```

Next we will write a delete method that returns **ErrInvalidID** if the ID is zero, otherwise it will construct a user object and then tell GORM to delete that particular user. Our method will also return an errors encountered along the way.

```
// Delete will delete the user with the provided ID
func (us *UserService) Delete(id uint) error {
    if id == 0 {
        return ErrInvalidID
    }
    user := User{Model: gorm.Model{ID: id}}
    return us.db.Delete(&user).Error
}
```

Note: Because the `gorm.Model` type is embedded into our `User` type we can access those fields like they were part of the `User` type - eg `user.ID` - but when constructing a new user we are explicitly creating a `gorm.Model` and assigning it to the embedded `Model` field inside our new `User`.

Finally we are going to test out our two methods inside of our experimental program.

```
$ atom exp/main.go
```

To test that a user is updated we are going to take the user our code already creates and we are going to update the name field. This isn't an exhaustive test of all fields, but it should at least give us confidence that our code appears to be working as intended. If all goes according to plan, the same user will be queried after it is updated and we will see the updated information printed out.

```
// ... Code to create a user remains above here

// Update a user
user.Name = "Updated Name"
if err := us.Update(&user); err != nil {
    panic(err)
}

foundUser, err := us.ByEmail("michael@dundermifflin.com")
if err != nil {
    panic(err)
}
// Because of an update, the name should now
// be "Updated Name"
fmt.Println(foundUser)
```

Testing delete is very similar. We must first call the `Delete` method, then verify that it worked by querying for a user. This time we will query for the user using the same ID we just deleted, and we will verify that a `ErrNotFound` error is returned, indicating that the user no longer exists in our database.

```
// Delete a user
if err := us.Delete(foundUser.ID); err != nil {
    panic(err)
}
// Verify the user is deleted
_, err = us.ByID(foundUser.ID)
if err != models.ErrNotFound {
    panic("user was not deleted!")
}
```

Box 9.5. Using errors to dictate flow control

In many languages it is considered a bad practice to use errors to handle the flow control. There are several reasons for this, but the biggest two are cost and readability of code.

Exceptions in most languages are expensive operations that eat up a good bit of CPU and memory resources because they need to calculate things like a stack trace. Exceptions can also be hard to follow because they might escape multiple levels of function calls before they are handled. For example, if `main()` called `a()` which called `b()` and it finally threw the exception, we don't know if that will be handled by the calling method `a()`, or if it will be handled by `main()`. Throw in a few more layers of function calls and this quickly becomes hard to debug and maintain.

Errors are very different in Go. They are not your traditional exceptions, but instead are values that can be returned, used to dictate logic, and aren't incredibly expensive. As a result, it isn't uncommon or frowned upon to use them to determine the flow of your program.

See <https://blog.golang.org/errors-are-values> for more info.

After running your code read the logs and look for what happens when we execute a delete.

As mentioned in [Box 8.5](#), GORM uses a `deleted_at` field by default for deleting records, so rather than finding a `DELETE` SQL statement you will instead see an `UPDATE`.

Source code

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

Completed - book.usegolang.com/9.5

Changes - book.usegolang.com/9.5-diff

9.6 AutoMigrating and returning errors from DestructiveReset

When we move over to our production environment we aren't going to want to destroy our database every time we need to migrate it, so we are going to also add an `AutoMigrate` function to our user service. Once again, this will be added to the users model source file.

```
$ atom models/users.go
```

The code for this is going to look familiar because it is simply a subset of `DestructiveReset`. The only real difference is that we are going to return an error if there is one.

```
// AutoMigrate will attempt to automatically migrate the
// users table
func (us *UserService) AutoMigrate() error {
    if err := us.db.AutoMigrate(&User{}).Error; err != nil {
```

```
    return err
}
return nil
}
```

While we are adding better error handling, we will update the **DestructiveReset** method as well to return an error when there is one. We can also use the new **AutoMigrate** function we just wrote rather than interacting directly with GORM for that part of our code, so if this changes in the future we won't need to update the code in two places.

```
func (us *UserService) DestructiveReset() error {
    err := us.db.DropTableIfExists(&User{}).Error
    if err != nil {
        return err
    }
    return us.AutoMigrate()
}
```

We are now ready to start using our user service inside the users controller.

Source code

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

Completed - book.usegolang.com/9.6

Changes - book.usegolang.com/9.6-diff

9.7 Connecting the user service and controller

The user model wouldn't be very helpful unless we actually utilized it in our code, so that last thing we are going to cover in this chapter is how to use our user service inside of our users controller. Afterwards, when we submit the sign up form it will interact with our database using our user service. We still won't be validating our users, displaying useful error messages, hashing passwords correctly, or really anything that we need to do to truly create a user resource, but those can come later once we have all the major pieces connected.

We will complete this code in roughly 3 steps:

1. Update the forms to take the name field into account.
2. Update our `main.go` source file to construct a database connection string and setup a new user service.
3. Update the users controller to accept a user service in its constructor and utilize it in the `Create` method.

We will cover each of these in subsections below.

9.7.1 Adding the name field the sign up form

Our sign up form doesn't have any notion of a user's name, so we need to first add that to the HTML form.

```
$ atom views/users/new.gohtml
```

The password field and email address fields will remain unchanged. We simply need to add a new field with the type `text` for the name field.

```
 {{define "signupForm"}}
<form action="/signup" method="POST">
  <div class="form-group">
    <label for="name">Name</label>
    <input type="text" name="name" class="form-control" id="name" placeholder="Your full name">
  </div>
  <!-- ... Everything else remains unchanged -->
</form>
{{end}}
```

After this you should start (or restart) your server to verify that the form was updated and include the field correctly. This is shown in Figure 9.1.

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

Next we need to update our users controller's **SignupForm** field to let it know about the new name field.

```
$ atom controllers/users.go
```

The code change is only one new line of code thanks to the **schema** package we are using.

```
type SignupForm struct {
  Name      string `schema:"name"`
  Email    string `schema:"email"`
  Password string `schema:"password"`
}
```

You can also update your **Create** handler in the users controller to print out the name to ensure it is being parsed correctly.

[H]

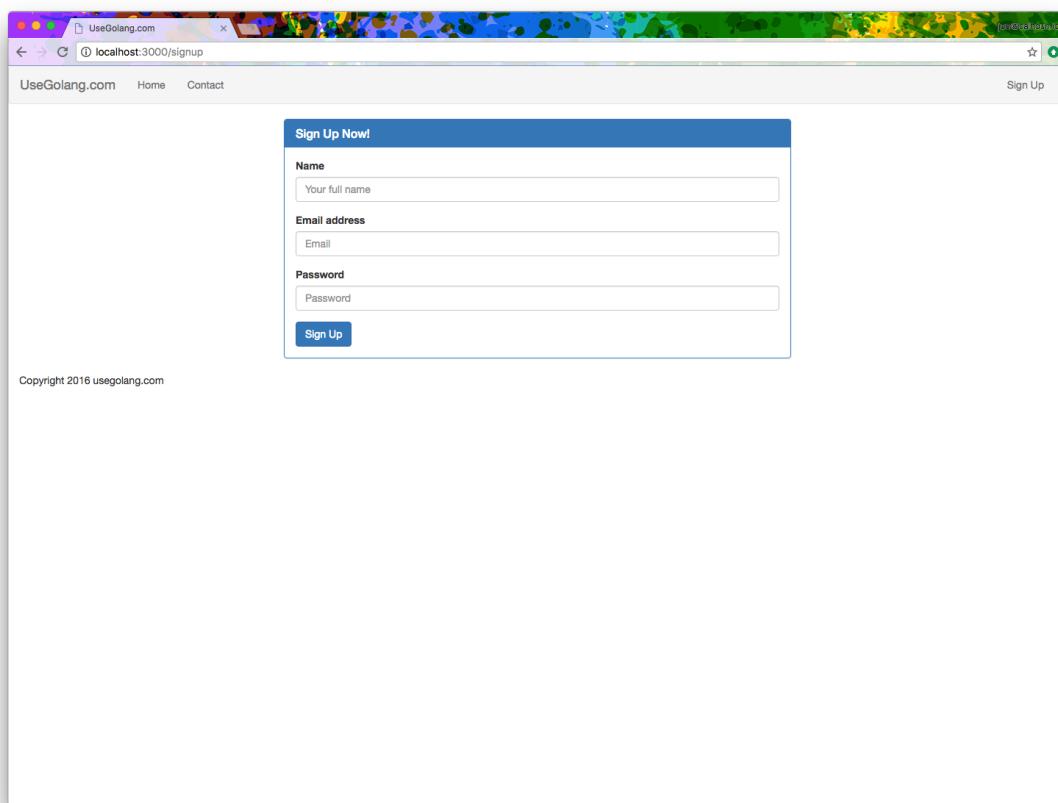


Figure 9.1: Sign up form with the name field

```
fmt.Fprintln(w, "Name is", form.Name)
```

Restart your server and verify that this all works before proceeding.

Source code

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

Completed - book.usegolang.com/9.7.1

Changes - book.usegolang.com/9.7.1-diff

9.7.2 Setting up a user service in our web application

Our next step is going to be setting up a user service when our application starts so that we can provide this service to any parts of our web application that need it.

While it might be tempting to push this logic off to the pieces that need a user service, doing so would have a few repercussions. For starters, several pieces of code would need to know about how to construct a Postgres connection string, would have to write custom logic to close the user service when done with it, and we would have several connections to our database with no real benefits.

Instead, we are going to create a single copy of the user service, defer closing it until our entire application is shutting down, and then share that user service with our user controller and any other pieces of code that need access.

We will be working in our web application's main package now, so open up the source file.

```
$ atom main.go
```

We haven't added any variables related to our database to this source file yet, so we are going to copy over the constants we have in our experimental `main.go` source file. Later we will explore how to move these into a config file, but for now this will work well enough. This is shown in [Listing 9.5](#), but you may need to tweak some of the values.

Listing 9.5: Add your database info to `main.go`

```
const (
    host      = "localhost"
    port      = 5432
    user      = "postgres"
    password  = "your-password"
    dbname   = "lenslocked_dev"
)
```

Box 9.6. Don't ship to production with passwords in code

We have been storing our password and other information required to connect to Postgres as constants in our application so far, and while that is fine for developing applications, you shouldn't ship to a production server or commit to git with passwords stored this way.

Eventually we will pull these constants out of our code and provide them via flags to our application when we start it up, but in the meantime just be aware that this isn't a great way to do things, but we are doing it to simplify our code a bit until we start talking about how to get it production ready.

Next we need to create our connection string. Again you can copy that from the experimental code from before. This will go inside of the `main` function.

```
func main() {
    // Create a DB connection string and then use it to
    // create our model services.
    psqlInfo := fmt.Sprintf("host=%s port=%d user=%s "+
        "password=%s dbname=%s sslmode=disable",
        host, port, user, password, dbname)

    // ... the controllers and routing code is unchanged
}
```

After that we need to create a user service using this connection string. Right after creating the user service we will also want to check for an error, defer closing it until our main function exits, and call the **AutoMigrate** function we wrote earlier to make sure our database is migrated properly.

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

    "lenslocked.com/controllers"
    "lenslocked.com/models"

    "github.com/gorilla/mux"
)

// ...

func main() {
    // ... the psqlInfo code comes first

    us, err := models.NewUserService(psqlInfo)
    if err != nil {
        panic(err)
    }
    defer us.Close()
    us.AutoMigrate()

    // ... the controllers and routing code is unchanged
}
```

We won't use the user service we setup quite yet, but go ahead and make sure your code compiles and works. In the next section we will look at using this inside of the users controller.

Source code

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

Completed - book.usegolang.com/9.7.2

Changes - book.usegolang.com/9.7.2-diff

9.7.3 Using the users service in our users controller

We are going to update our users controller to accept a **UserService** instance when it is constructed, and then we will use that user service inside of our **Create** method to actually create a database backed user.

Let's start by opening up the users controller source code.

```
$ atom controllers/users.go
```

Next we will add the **UserService** to the **Users** type inside of our controllers package so that all of our handler methods can access the user service when they need it. We will also need to add the models package to our imports in the process.

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

    "lenslocked.com/models"
    "lenslocked.com/views"
)

// ...

type Users struct {
    NewView *views.View
```

```
    us      *models.UserService
}
```

Our **NewUsers** function doesn't currently assign anything to this field, so let's fix that next. Rather than constructing the service, we are going to accept it as an argument to the function.

```
func NewUsers(us *models.UserService) *Users {
    return &Users{
        NewView: views.NewView("bootstrap", "users/new"),
        us:       us,
    }
}
```

Next we need to update our **main.go** source code. Right now it doesn't pass any arguments when it calls the **NewUsers** function, but we just updated the function to require a user service.

```
$ atom main.go
```

```
func main() {
    // ... most of this doesn't change

    // Find this line, and update it
    usersC := controllers.NewUsers(us)

    // ...
}
```

We can now continue updating the users controller. Specifically, we are going to use the user service in our **Create** handler to actually store a user in the database.

```
$ atom controllers/users.go
```

Next we update our code to create a `models.User` using the sign up form. We won't be using the password input just yet because we don't have a proper way to store passwords, but we will fix in the next few chapters. In the meantime we will just store the user's name and email address to verify everything is working as intended.

```
func (u *Users) Create(w http.ResponseWriter, r *http.Request) {
    var form SignupForm
    if err := parseForm(r, &form); err != nil {
        panic(err)
    }
    user := models.User{
        Name:  form.Name,
        Email: form.Email,
    }
    if err := u.us.Create(&user); err != nil {
        http.Error(w, err.Error(), http.StatusInternalServerError)
        return
    }
    fmt.Fprintln(w, "User is", user)
}
```

If we happen to get an error while creating a user we will also return a status code indicating that something went wrong on our server. Typically I wouldn't recommend rendering raw error messages to the end user, but this code is temporary and will only be seen by developers so I am adding it in case we have an issue and need to debug.

Restart your server and test it out. If all goes well, submitting the sign up form should create a user object and run some SQL like the example below.

```
INSERT INTO "users" ("created_at", "updated_at",
    "deleted_at", "name", "email")
VALUES ('2017-07-24 15:25:45',
    '2017-07-24 15:25:45', NULL, 'Michael Scott',
    'michael2@dundermifflin.com')
RETURNING "users"."id"
```

Note: If you see an error like the below, it indicates that you tried to create a user that already exists in your database.

```
pq: duplicate key value violates unique constraint "uix_users_email"
```

Source code

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

Completed - book.usegolang.com/9.7.3

Changes - book.usegolang.com/9.7.3-diff

9.8 Exercises

The exercises for this chapter are once again going to involve code that we won't be keeping, but are instead meant to help reinforce what we just learned and coded. If you do write and run any of this code, be sure to reset your database to reflect the user table we will be using in the rest of the book.

9.8.1 Ex1 - Add an **Age** field to our user resource

Adding a new field to a resource involved a few steps. Try to walk through them all to see if you can figure out what all code you would need to add in order to let a user provide their age when creating an account and have it persisted in the database.

9.8.2 Ex2 - Write a method to query the first user with a specific age

Once you have added the age field to your users model, write a method that will allow you to query users with a specific age. This should be attached to the **UserService** like our other query methods, and should be pretty similar to our **ByID** method.

9.8.3 Ex3 - [HARD] Write a method to query many users by age range

Our last exercises is a little harder only because you will need to write a bit more code and dig into GORM more.

Check out the last chapter and combine the information there demonstrating how to query multiple users with your knowledge of how GORM's **Where** method works to create a **InAgeRange** method on our user service.

The **InAgeRange** method should accept two arguments, a minimum age and a maximum age, and then it should return all users within that age range. If no users are found, you should return an empty slice of users, and you should also return an error when there is one. You DO NOT need to return an error when there are no users found - an empty slice will signify this.

Chapter 10

Building an authentication system

Authentication is arguably the most important and most sensitive part of your web application.

Not only can a poorly executed authentication strategy leave your user data at risk, you could also potentially leak passwords that are being used on other websites. Yikes!

While this might seem scary at first, the truth is that implementing a secure authentication service isn't really that hard, but it is **very important that you do not deviate from industry standards** when designing your authentication system.

Developers with good intentions deviating from the industry standard is what leads to major security issues, and you don't want to see your website on the front page of the New York Times with a headline like "Data breach could put your accounts at risk."

To help make sure this doesn't happen, we will spend the first half of this chapter learning about how to properly store a password without writing any code, and

then once we know what we are about to do we will walk through implementing each of the pieces we discussed.

10.1 Why not use another package or service?

A fairly common question I receive is, “Why don’t we just use another package or service that does all this for me?”

The short answer is, without truly understanding what goes into a secure authentication system you might unknowingly write code that causes a security leak. The only real way to avoid this is education. Developers need to be aware of what security measures are necessary to properly store passwords, and only then can they make decisions with confidence that they aren’t making an application insecure.

Another reason for this 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. There is no shortage of random criteria that can be added to your authentication system.

Not only will you need to customize your authentication system over time, but these customizations often make it impossible to just plug an off-the-shelf solution into your web application. That means customization will be necessary, and, as I said before, customizing without knowing what may or may not make the entire system insecure is a bad idea.

In addition to benefiting from the knowledge you will gain building your own authentication system, you very well may end up saving time by writing your own authentication system. While this may seem crazy at first, customizing a package someone else created means learning how it works so that you can add in your tweaks. This takes time, and often times you can build an entire authentication system from scratch in a similar amount of time, but you will

definitely understand how your system works because you wrote it.

And finally, using services like Auth0 and Stormpath add an additional cost to your bottom line. I'm not saying that they aren't worth that additional cost. They definitely are in some situations, but many people reading this book are looking to build a small side project application, and it is hard to keep a side project running when it costs you hundreds of dollars every month.

10.2 Secure your server with SSL/TLS

If your website stores any sort of user data, you should strongly consider getting a certificate for your site so that it can be served with **https** instead of **http**. Once you have a certificate, you should redirect all of your web traffic to the **https** version of your site. This is especially true if you are going to allow users to create accounts and provide you with a password, as otherwise their password would be sent to your server without any encryption.

We won't actually get to this step until later in the book when we prepare for production and deploy our application, but this chapter would feel incomplete if I didn't at least mention that SSL is a requirement for a proper authentication system.

10.3 Hash passwords properly

The most common mistake made when building a custom authentication system is to choose a bad way to store a user's password in your database.

As a general rule, **if you can decrypt the password, you are making a very bad mistake**. Your web application shouldn't be able to decrypt a user's password ever, period. It doesn't matter what your use case is, or what super awesome service you are working on; If you can decrypt a user's password, then an

attacker could decrypt it as well.

Now that might seem confusing at first. If you can't figure out what a user's password actually is, how can you verify their password when they log in?

That, my friends, is where hash functions come into the equation.

10.3.1 What is a hash function?

A hash function is a function that can be used to take in data of any arbitrary size and convert it into data of a fixed size.

The act of applying a hash function is often called "hashing", and the values returned by a hash function are often referred to as hash values or hashes.

Hashes are useful in a wide variety of programming applications, and I definitely recommend doing some researching and reading up on them in your spare time, but for our purposes we are going to briefly look at a simpler example, and then focus exclusively on [bcrypt](#), a hash function designed explicitly for hashing passwords.

For our simple example, imagine a function that takes in any string value and translates it into a number between 0 and 4. One way to implement this is to just count the number of letters in the string and then use the modulus (%) operator to get a number between 0 and 4.

Listing 10.1: A fake hash function

```
// Do not code this
func hash(s string) int {
    return len(s) % 5
}
```

While this might not be a great hashing function, it does illustrate several key points that we need to discuss.

The first is that two values might be converted to the same hash value. For example, if we provided the input “jon” it would return 3, but if we provided the input “password” it would also return 3 (**8 % 5 = 3**).

This is inevitable because we are saying that you can put a string of any size into our function, and we are limited to just 5 possible output values.

The second key point is that it is impossible to take a hash value and convert it back into the original string. Looking back at the same example, if we had the output 3, it is impossible to determine if the original input was “jon”, “password”, or some other string entirely.

10.3.2 Store hashed passwords, not raw passwords

How does this help us with passwords? Well, instead of storing the password that the user types in, we are instead going to store a hash of their password. That way 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.

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, so when a user tries to log in we can hash the password they typed in and then ask “is the hash from the password they just typed in the same as the hash we stored?”

If it is, the user provided the correct password. If it isn’t, the user typed the wrong password.

We can effectively validate a user’s password, but we have limited our knowledge so that we don’t have any way of remembering what their password is, and since we don’t know what their password is, a hacker won’t be able to figure it out either.

But that raises yet another question. Didn’t we say earlier that two inputs can hash to the same value? What if two different passwords get hashed to the same

value?

This is a real possibility with hashes, but we will be using a much better hashing function that has so many possible outputs that is **extremely** unlikely that this would ever happen. You probably have a better chance of being struck by lightning after buying the winning lottery ticket on the third Tuesday of January.

I haven't actually done the math to verify that claim, but my point is that accidentally guessing a password that hashes to the same value as a user's password, but *isn't* their password, is not really worth worrying about.

Box 10.1. Encryption functions are NOT hashing functions

At some point you may learn about encryption functions like [AES](#) and think they would be a good fit here, but I want to once again reiterate that encryption and hashing are two different things, and state that encryption is not appropriate for an storing passwords.

The reason for this is pretty simple. 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, and once they do they will be able to decrypt every password in your database.

10.3.3 Salt and pepper passwords

We have one last thing left to discuss before we start storing passwords in our database, and that is adding salt and pepper to our passwords. And before anyone asks, no, I haven't accidentally mixed up a cookbook manuscript with my web development book.

What is a password salt?

Salting a password is the act of adding some random data to a password before hashing it. For example, if you typed in the password “abc123”, a program might salt the password by adding “ja08d” to the password, making the string “abc123ja08d” which would be hashed and stored in the password hash field instead of the original.

In the future when a user logs in and types their password “abc123” we would add their salt, “ja08d”, to the password, and then hash it to determine if they provided us with the correct password.

Earlier I told you the it is impossible to go from a hashed value back to the original value, so this might seem like a waste of time at first, but I promise you it isn’t. Password salts are necessary to prevent hackers from using rainbow tables to crack your users’ passwords.

What are rainbow tables?

The goal when designing a password system is to design one such that even if an attacker gets a copy of your database, they won’t be able to actually figure out what each user’s password is. That way, even if your company does lose access to a server, you don’t have to worry about your user’s having all of their accounts on other sites stolen. You should still take every precaution to secure your own server from attackers, but this is our final fail-safe.

As we discussed earlier, it isn’t possible to take a hash value and reverse it into the original password, so at first it would appear that we are covered, but unfortunately hackers are a pesky bunch, and over time have developed new ways of trying to determine passwords, even if they are hashed.

Hackers discovered that while it isn’t possible to reverse a password hash, it is possible to generate a very large list of possible passwords and then hash each of these. Now they don’t actually have to reverse a hash; Instead they can just

check to see if their hash value matches the one you generated.

Let's look at an example to clarify. Imagine that I had a really long list of common passwords. My list might even have some random gibberish passwords that I think are likely to be used.

```
password
abc123
abcd1234
ak sdfj
```

My list might have several million passwords in it, but for this example we will just use 4 passwords. Now if I know you are using a specific hashing function, I could then run that hashing function on each of my passwords and store the results.

```
fake-hash-value-for-password
fake-hash-value-for-abc123
fake-hash-value-for-abcd1234
fake-hash-value-for-ak sdfj
```

This is called a rainbow table, and after stealing your database I would take all of the hashes in your database and see if any matches the values in my rainbow table. If one did match, I would know what their original password was by looking at my original list of passwords and determining which generated that hash.

The problem with this approach is that rainbow tables take a while to create. Hashing millions of possible passwords takes time, and hackers might really have dictionaries with billions of possible passwords. As a result, this attack really only works if I can use the same rainbow table on every password stored in a database.

That is where salts come into play. By adding a different random string to each password, the rainbow table becomes useless. The hash of “password-ja08d” is

different than the hash of “password-salt2”, so if every user has a different salt, the attacker would need to create a custom rainbow table for each user. Ouch!

One important thing to note is that salts aren’t private data. An attacker can gain access to a user’s salt, but because each user should have a different salt, they would still need to generate a new rainbow table for each individual user, making it impractical.

What is a password pepper?

A pepper is very similar to a salt, but rather than being user-specific it is application specific. In our application we would have a universal pepper value that is similar to a salt. It is just some random string that we can append to passwords before hashing them.

Unlike a salt, a pepper is never stored in your database. That way if an attacker does manage to gain access to your database but not your application they will only have access to hashed passwords and salt values, but they won’t actually know what pepper value you used, making it much harder for them to discover what the underlying passwords were.

In my opinion, this is one of the weaker precautions out of all of the ones we will be using, but it is incredibly easy to implement so it is worth adding anyway.

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 the two are so similar and “salt & pepper” is a popular duo (at least in the United States).

10.4 Implementing password hashing for our users

Now that you understand how we are going to be storing passwords, let's go ahead and look at how to implement everything we have discussed. We are going to start by adding fields to our user model to store our password. After that we will learn how to hash passwords using **bcrypt**, a password hashing function. Once that is done we will need to update some of our controller code to pull the password from the sign up form, and then finally we will learn how to salt and pepper our password when hashing them.

10.4.1 Adding password fields to the user model

Our first changes will be made to the user model.

```
$ atom models/users.go
```

While we are only accepting a single password input from users when they sign up, we are going to add two fields to our user model. The first field will be named **Password**, and it will be used to store the raw (unhashed) password. That means that we will NEVER save this field in the database, and GORM provides us with a way to ensure this via the `-` struct tag.

Note: We should also avoid logging the user's raw password, so keep this in mind when writing any logging code.

The second field will be named **PasswordHash**, and will only store hashed passwords. This field will be the one we store in our database.

Listing 10.2: Adding password fields to the user model

```
type User struct {
    gorm.Model
    Name      string
    Email     string `gorm:"not null;unique_index"`
    Password  string `gorm:"-"`  
    PasswordHash string `gorm:"not null"`
}
```

By separating our two password fields we are ensuring that we can't easily get the two mixed up. If we were to instead use a single password field it might be hard to determine whether or not the password has been hashed. This could lead to accidentally hashing a password that is already hashed, or worse yet - forgetting to hash a raw password.

Box 10.2. You may need to reset your database

If you have any users already stored in your database then you will need to reset your database rather than migrating it. This is due to the fact that any existing users will have no password hash set, but the new column is required, so there will be an error during that migration.

You can tell that this is happening when you see the following error when you start your application:

```
pq: column "password_hash" contains null values
```

Typically we would create a migration intended to handle this, or some other sort of fix, but since we are in development it is much easier to simply reset the database.

You can do this by running the **DestructiveReset** function we wrote as part of our user service earlier.

Source code

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

Completed - book.usegolang.com/10.4.1

Changes - book.usegolang.com/10.4.1-diff

10.4.2 Hash passwords with bcrypt before saving

We now have a field to store our hashed password in, but we don't have any code that hashes a raw password and stores the result in the user's **PasswordHash** field.

We are going to write this inside of our user service so that in the future any controllers, API endpoints, or anything else used to create a user doesn't have to think about the logic used to hash a user's password. Instead they simply need to pass a user with a raw password in, and a user with a hashed password will be saved to the database.

To do this we need a hash function, so we will start by getting the **bcrypt** package.

Listing 10.3: Installing bcrypt

```
$ go get -u 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.

You can check out bcrypt's docs here: <https://godoc.org/golang.org/x/crypto/bcrypt>

The primary functions we will be using in this package are:

- **GenerateFromPassword**¹
- **CompareHashAndPassword**²

The first is used to generate a password hash from an raw user password, and the second is used to compare a user-provided password with a hash we have stored to verify whether or not they match.

We are going to be using bcrypt inside of our users model, so we can start by opening up that source code.

```
$ atom models/users.go
```

We can't use the bcrypt package without first importing it...

```
import (
    "errors"

    "github.com/jinzhu/gorm"
    _ "github.com/jinzhu/gorm/dialects/postgres"
    "golang.org/x/crypto/bcrypt"
)
```

We won't be using the comparison function just yet, but we do want to write some code to hash our passwords before creating a user resource. To do this we will be using bcrypt inside of the **Create** method we wrote earlier in our user service.

¹<https://godoc.org/golang.org/x/crypto/bcrypt#GenerateFromPassword>

²<https://godoc.org/golang.org/x/crypto/bcrypt#CompareHashAndPassword>

Listing 10.4: Hashing the password with bcrypt

```
func (us *UserService) Create(user *User) error {
    hashedBytes, err := bcrypt.GenerateFromPassword(
        []byte(user.Password), bcrypt.DefaultCost)
    if err != nil {
        return err
    }
    user.PasswordHash = string(hashedBytes)
    user.Password = ""
    return us.db.Create(user).Error
}
```

There is a lot going on in Listing 10.4, so let's take a moment to review the code.

The first thing we are doing is calling `bcrypt.GenerateFromPassword()`. This function requires a byte slice as the first argument, but our user password field is a string. That means we need to convert our password string into a byte array. You can do this in Go by typing:

```
[]byte(user.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 racecar 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**³ constant that can be changed in the library when CPUs have a drastic improvement. We only need to update our bcrypt package to receive this update.

The **GenerateFromPassword** function returns a byte slice and an error. If we receive an error we can simply return it instead of saving our user, but typically this shouldn't happen. When we don't have an error we convert the byte slice into a string so that we can store the results in our user's **PasswordHash** field.

Note: Not all byte slices can be converted into a string because not all bytes map to valid characters. In this case it works out, but keep that in mind in the future.

Finally, we set the password to the empty string so that it is no longer accessible in our application and create our user. While it isn't necessary to set the user's password field to an empty string, I prefer to clear any references to cleartext password as soon as possible to minimize the chances of it being leaked in logs or anywhere else.

³<https://godoc.org/golang.org/x/crypto/bcrypt#pkg-constants>

Source code

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

Completed - book.usegolang.com/10.4.2

Changes - book.usegolang.com/10.4.2-diff

10.4.3 Retrieving passwords from the sign up form

We have a way to hash passwords, but that isn't very useful if we aren't parsing the passwords from the sign up form and setting them in our user model. Luckily this is a pretty quick change in the users controller.

```
$ atom controllers/users.go
```

We will be updating the **Create** method, and since our **SignupForm** already has the password field we only need to add it to the user model.

Listing 10.5: Use the sign up form password

```
func (u *Users) Create(w http.ResponseWriter, r *http.Request) {
    // ... This is unchanged
    user := models.User{
        Name:      form.Name,
        Email:    form.Email,
        Password: form.Password,
    }
    // ... This is unchanged
}
```

Now our password is being parsed from the signup form, added to the user model, and passed along to the call to create a user via the user service which

handles hashing the password, but we aren't quite done. We still need to discuss salting and peppering the password.

Source code

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

Completed - book.usegolang.com/10.4.3

Changes - book.usegolang.com/10.4.3-diff

10.4.4 Salting and peppering passwords

The last thing we need to do before we are ready to start creating users with a hashed password is to implement a salt and pepper. Luckily, we don't actually need to do anything to implement a salt because bcrypt handles this for us.

Box 10.3. How does bcrypt store the salt?

Whenever bcrypt creates a hash, it also adds some additional data to the resulting hash before returning it. For example, imagine we had the following hash generated by bcrypt:

```
$2a$10$N9qo8uLoickgx2ZMRZoMyeIjzAgcf17p921dGxad68LJZdL171hWy
```

This hash can be broken into several pieces:

```
$2a$  
10$  
N9qo8uLoickgx2ZMRZoMye  
IjzAgcf17p921dGxad68LJzdL171hWy
```

When you put all of these pieces together they create our original hash, but separately they each tell bcrypt something about the hash.

The first piece, **\$2a\$**, is used to tell bcrypt what format we are using and how to process the rest of the hash. That is why it comes first - we couldn't do much with the rest of the hash unless we knew its format.

The second piece is the cost of the hash. In this case we have **10\$** which means we are using a cost of 10. As of this writing, 10 is the default cost for the bcrypt package in Go.

After that we have a 22 character string - **N9qo8uLoickgx2ZMRZoMye**. This is the salt used in our password encoded in base-64.

Finally we have the last piece - **IjzAgcf....**. This is the actual hash generated by bcrypt using the plaintext password, the salt, and the cost discussed above.

Note: I mention how salting works in this book so that you are aware of it in the future, even if you do not need to implement it yourself.

On the other hand, peppering the password isn't handled by bcrypt so we are going to need to write some code for that. We will be doing this inside of the users model.

```
$ atom models/users.go
```

For now we are going to add the pepper as a package variable named **userPwPepper**. You can pick whatever value you want for your string, and before we deploy

we will move this out of the package and instead use a config file to specify our pepper. That will allow us to use a different, secret pepper in production while using a dummy value in development.

```
// Add this to your users model
var userPwPepper = "secret-random-string"
```

Next we need to add the pepper to our user provided password. To do this we will use the `+` operator and concatenate the user's password with our pepper, and then we will hash the resulting string.

We will be doing this inside of the `Create` method for now, but we will eventually need to move this code elsewhere so we can also allow users to update their password.

Listing 10.6: Add a pepper to the password when creating users

```
func (us *UserService) Create(user *User) error {
    pwBytes := []byte(user.Password + userPwPepper)
    hashedBytes, err := bcrypt.GenerateFromPassword(
        pwBytes, bcrypt.DefaultCost)
    if err != nil {
        return err
    }
    user.PasswordHash = string(hashedBytes)
    user.Password = ""
    return us.db.Create(user).Error
}
```

Congratulations! You just finished the final piece necessary to hash a user's password. If you want to give it a test run, restart your application and sign up for an account. We won't be able to authenticate your account just yet, but that will be our next task.

Note: If you created any users before this point you should reset your database, as their passwords likely aren't valid and we won't be able to authenticate them in the next section.

Source code

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

Completed - book.usegolang.com/10.4.4

Changes - book.usegolang.com/10.4.4-diff

10.5 Authenticating returning users

We have users stored in our database, and now it is time to code the second half of our authentication system - the code used to verify that a user is actually who they say they are.

The rest of this chapter focuses exclusively on allowing a user to fill out a log in form, parsing the data, and then verifying that the email address and password provided are correct. While we will eventually need a way to remember users after they log in, we will defer that until the next chapter.

10.5.1 Creating the login template

The first piece of code we are going to write is the log in form. It is going to be nearly identical to the sign up form with two major distinctions:

1. It will POST to `/login` instead of `/signup`.
2. We won't have an input for a user's name, but will instead only ask for their email address and password.

We are going to create the template inside of the users views directory.

```
$ atom views/users/login.gohtml
```

After we have the file created we need to fill it in with the code from Listing 10.7.

Listing 10.7: Creating the login template

```
 {{define "yield"}}
<div class="row">
  <div class="col-md-4 col-md-offset-4">
    <div class="panel panel-primary">
      <div class="panel-heading">
        <h3 class="panel-title">Welcome Back!</h3>
      </div>
      <div class="panel-body">
        {{template "loginForm"}}
      </div>
    </div>
  </div>
{{end}}
```

```
 {{define "loginForm"}}
<form action="/login" method="POST">
  <div class="form-group">
    <label for="email">Email address</label>
    <input type="email" name="email" class="form-control"
      id="email" placeholder="Email">
  </div>
  <div class="form-group">
    <label for="password">Password</label>
    <input type="password" name="password"
      class="form-control" id="password"
      placeholder="Password">
  </div>
  <button type="submit" class="btn btn-primary">Log In</button>
</form>
{{end}}
```

Because this code is virtually identical to the code we wrote in Section 7.1.2 we won't be covering what it does in detail here. If you are unfamiliar with any of the code, I suggest reviewing Chapter 7.

Source code

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

Completed - book.usegolang.com/10.5.1

Changes - book.usegolang.com/10.5.1-diff

10.5.2 Creating the login action

In order to render and parse our log in form we are going to need to update the users controller. Specifically, we are going to need to create a new view that renders our template, and we will need to create a method used to process the form when it is submitted.

Technically speaking, logging in is more akin to creating a session than it is to altering a user, so many developers will choose to place these views and actions on a sessions controller. While there isn't anything wrong with that approach, I personally prefer to place these actions on the users controller because sessions and users are so tightly linked, and we wouldn't be doing anything else with the sessions controller aside from logging users in or out.

Open up your users controller source code.

```
$ atom controllers/users.go
```

We will start by adding a view used to render the `login.gohtml` template, and we will name this view `LoginView`.

```
func NewUsers(us *models.UserService) *Users {
    return &Users{
        NewView:   views.NewView("bootstrap", "users/new"),
```

```

    LoginView: views.NewView("bootstrap", "users/login"),
    us:          us,
}
}

type Users struct {
    NewView  *views.View
    LoginView *views.View
    us        *models.UserService
}

```

After that we need to create an action used to parse the form when it is submitted. The process for doing this is very similar to the process used to create the **Create** method and the **SignupForm** types, but in this case we will be writing code used to parse a login form.

```

type LoginForm struct {
    Email    string `schema:"email"`
    Password string `schema:"password"`
}

// Login is used to process the login form when a user
// tries to log in as an existing user (via email & pw).
//
// POST /login
func (u *Users) Login(w http.ResponseWriter, r *http.Request) {
    form := LoginForm{}
    if err := parseForm(r, &form); err != nil {
        panic(err)
    }
    // We will eventually do something to see if the
    // information provided is correct.
}

```

In the next section we will finish the **Login** method by making a call to the user service's **Authenticate** method and verifying whether the information provided was valid, but for now we will leave a comment.

In the meantime we need to update our router to inform it of the new pages we added to our web application.

```
$ atom main.go
```

```
func main() {
    // ... nothing changes above here
    r.HandleFunc("/signup", usersC.New).Methods("GET")
    r.HandleFunc("/signup", usersC.Create).Methods("POST")
    // NOTE: We are using the Handle function, not HandleFunc
    r.Handle("/login", usersC.LoginView).Methods("GET")
    r.HandleFunc("/login", usersC.Login).Methods("POST")
    // ... nothing changes below here
}
```

Finally, we are going to update our navbar template to add a link to the log in page.

```
$ atom views/layouts/navbar.gohtml
```

```
<!-- Find the navbar-right section and add the login link -->


- Log In
- Sign Up

```

With those changes we are done adding the login page and are ready to move on to implementing the **Authenticate** method for our user service.

Source code

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

Completed - book.usegolang.com/10.5.2

Changes - book.usegolang.com/10.5.2-diff

10.5.3 Implementing the `Authenticate` method

Our authenticate method is going to do one thing - take in an email address and a cleartext password, and then return either a user that matches that information or an error explaining why a match couldn't be found. It will NOT be responsible for creating a session, altering cookies, or anything like that. That responsibility will instead lie on whatever code calls the authenticate method.

We will be writing this method in the users model source file.

```
$ atom models/users.go
```

We will start by defining a new error. We already have an error for when a resource can't be found, so if we are provided with an invalid email address we could return that error, but we still don't have any error that would be relevant if the password provided is invalid.

```
var (
    // ... Leave the existing errors unchanged

    // ErrInvalidPassword is returned when an invalid password
    // is used when attempting to authenticate a user.
    ErrInvalidPassword = errors.New(
        "models: incorrect password provided")
)
```

Now we have at least two potential errors we can return: ErrNotFound and ErrInvalidPassword. Let's use those to write a comment describing how the Authenticate method is going to work.

```
// Authenticate can be used to authenticate a user with the
// provided email address and password.
// If the email address provided is invalid, this will return
// nil, ErrNotFound
// If the password provided is invalid, this will return
// nil, ErrInvalidPassword
```

```
// If the email and password are both valid, this will return
// user, nil
// Otherwise if another error is encountered this will return
// nil, error
func (us *UserService) Authenticate(
    email, password string) (*User, error) {
    // ... we will implement this in a moment
    return nil, nil
}
```

In every case except for the very last one we will be returning an error defined by our **models** package. This is important because every case besides the last are uses cases where we should be able to handle the error and render an appropriate error message to our end user, but in the last case we might not be able to render a useful error for the end user and might instead need to interact to see what is wrong with our web application. For example, it might be experiencing an error because the database crashed.

Now that we have a rough understanding of what we want our **Authenticate** method to do we can implement it. We will start by looking up a user with the provided email address.

```
foundUser, err := us.ByEmail(email)
if err != nil {
    return nil, err
}
```

If you recall from earlier, when a user isn't found with the email address our **ByEmail** method will return the **ErrNotFound** error, so we don't need to actually check for that error in our code. We can simply return whatever error is returned by the **ByEmail** method.

Assuming we retrieve a user without any problems, our next step is to check to see if the provided password is correct. We will be using bcrypt to do this, using its **CompareHashAndPassword**⁴ function.

⁴<https://godoc.org/golang.org/x/crypto/bcrypt#CompareHashAndPassword>

The compare hash and password function accepts two parameters - the password hash and the cleartext password. Both of these need to be byte slices instead of strings, so we will need to convert our strings into byte slices. We will also need to add the pepper to our cleartext password because the hashed password stored in our database had it added before it was hashed.

The **CompareHashAndPassword** function will return an error if the passwords don't match for any reason, so we need to capture the return value as well.

```
err = bcrypt.CompareHashAndPassword(  
    []byte(foundUser.PasswordHash),  
    []byte(password+userPwPepper))
```

Once we have compared the passwords we need to check the error to determine what happened. There are three use cases we care about:

1. If the error is nil it means the password was correct.
2. If the error is the ErrMismatchedHashAndPassword⁵ error provided by the bcrypt package it means that the password we tried was incorrect. It DOES NOT mean that something went wrong.
3. Any other errors indicate that something went wrong. This probably means that the hashed password we have stored was corrupted somehow and shouldn't happen, but we will return the error if it ever does occur.

In code these three cases can be expressed with the following switch statement:

```
switch err {  
case nil:  
    return foundUser, nil  
case bcrypt.ErrMismatchedHashAndPassword:  
    return nil, ErrInvalidPassword  
default:  
    return nil, err  
}
```

⁵<https://godoc.org/golang.org/x/crypto/bcrypt#pkg-variables>

Putting this all together we get the implementation shown in Listing 10.8. I am not showing the comments in the listing for brevity, but you should keep the comments we wrote earlier in this section.

Listing 10.8: Fully implemented Authenticate method.

```
func (us *UserService) Authenticate(email, password string) (*User, error) {
    foundUser, err := us.ByEmail(email)
    if err != nil {
        return nil, err
    }

    err = bcrypt.CompareHashAndPassword(
        []byte(foundUser.PasswordHash),
        []byte(password+userPwPepper))
    switch err {
    case nil:
        return foundUser, nil
    case bcrypt.ErrMismatchedHashAndPassword:
        return nil, ErrInvalidPassword
    default:
        return nil, err
    }
}
```

That's it! We are ready to start using the authenticate method inside of our login action in the users controller.

Source code

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

Completed - book.usegolang.com/10.5.3

Changes - book.usegolang.com/10.5.3-diff

10.5.4 Calling `Authenticate` from our login action

The final piece to our authentication code is to call the `Authenticate` method from our login action using the data inside of the `LoginForm` that we parsed.

```
$ atom controllers/users.go
```

When calling the `Authenticate` method we also need to keep in mind that this could return either an error or a user. If there is an error we will want to check for: `ErrNotFound`, `ErrInvalidPassword`, or any other error.

In each of those error cases we will display a different error message, and if there isn't an error we will display the user object that was just authenticated.

```
func (u *Users) Login(w http.ResponseWriter, r *http.Request) {
    // ... Leave the existing code alone here
    user, err := u.us.Authenticate(form.Email, form.Password)
    switch err {
    case models.ErrNotFound:
        fmt.Fprintln(w, "Invalid email address.")
    case models.ErrInvalidPassword:
        fmt.Fprintln(w, "Invalid password provided.")
    case nil:
        fmt.Fprintln(w, user)
    default:
        http.Error(w, err.Error(), http.StatusInternalServerError)
    }
}
```

Note: We could use `http.Error` for every error case but will eventually replace all of this code so we can leave it as is for now.

Restart your server and test it all out. You may need to first create a new user via the sign up page, but once you do so you should be able to submit that user's email address and password on the log in page and see the user resource printed out to the screen.

Test out the potential errors as well. What happens if you plug in an invalid email address? What about a valid email address but an invalid password?

Source code

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

Completed - book.usegolang.com/10.5.4

Changes - book.usegolang.com/10.5.4-diff

10.6 Exercises

Before moving on to creating sessions, let's take a few moments to review some of the key concepts we covered in this chapter.

10.6.1 Ex1 - What is a hash function?

Try to explain in your own words what a hash function is.

10.6.2 Ex2 - What purpose does a salt and pepper serve?

We discussed using a salt and pepper - explain in your own words why these are important and what types of attacks they can be used to combat.

10.6.3 Ex3 - Timing attacks

We didn't cover timing attacks in this chapter because bcrypt handles this for us as part of its comparison function, but it is worth understanding what timing attacks are.

Do a little research on timing attacks and discover what they are.

After that, dig into the bcrypt package and learn why you don't need to worry about timing attacks when using the provided comparison function.

Hint: Look for the `ConstantTimeCompare` function call inside of `CompareHashAndPassword` if you get stuck.

Chapter 11

Remembering users

Up until now, our server has been stateless. That is, when a user visits our web application, it has no knowledge of who that user is or what pages the user has visited in the past. Instead, our server sees a single web request and nothing else.

The web request might be to create a new user account, to delete a gallery, or anything else. Whatever the specific details of the request are, the server can expect them all to be contained within that single web request.

One of the biggest advantages to this approach is that it allows developers to build highly scalable web applications. For example, if you are browsing Google.com, it doesn't matter which of their servers your web requests go to. Every request you send has everything that the server needs to respond to your request.

If instead the server had to remember what you were doing between every request, you would have to talk to the same server every time you talked to Google. What would happen if that server got overloaded with request? Or what if that server had a network outage? All of your work would be lost, which would really suck.

As a result, a stateless design is actually preferable for web applications, but how do we remember who a user is with a stateless server?

It turns out, the easiest way to do this isn't to "remember" who the user is between requests, but to instead have the user tell you who they are in every request. By having the user simply tell us who they are, we avoid needing to store any actual state, but this introduces a new problem - how do verify that the user is telling us the truth?

The rest of the chapter is mostly going to be dedicated to answering these questions; We will start by exploring how we can use cookies to store information that will be provided by the end user's browser on every request. After that we will look at ways to make the data that we store in the cookie a little more secure so that users can't lie to us about who they are, and then finally we will talk about some other security concerns that you will need to consider when using cookies.

Once we have covered everything we need to about cookies we will finally be prepared to utilize it all so that we can verify who a user is and whether or not they have access to a specific page.

11.1 What are cookies

At a high level, cookies are basically just data that is stored on a user's computer. When you visit a website that creates a cookie, it isn't actually storing that data on its own servers; The cookie file is saved on your computer. That is why you can clear your cookies in almost every browser - you are just deleting local files.

What makes cookies unique from other files on your computer is that every time you visit a website, your browser will automatically include the cookies that the website has permission to access. This is really handy, because it means that you don't need to type your password in every time you visit a web page. Instead information about who you are is stored in a cookie and can be saved

even after you close your browser. It is also why you get logged out of every website when you clear your cookies - the sites no longer have a way to tell who you are.

In the rest of this chapter we are going to focus on using cookies to remember who our user is, securing those cookies, and updating the rest of our code to limit user access to pages using data derived from our cookies.

11.2 Creating our first cookie

Creating cookies in Go is done by using the `net/http`¹ package. Doing so involves two steps:

1. Instantiate a `Cookie`², which is a type provided by the `net/http` package.
2. Call the `SetCookie`³ function provided by the `net/http` package, passing in both a `ResponseWriter` and the cookie we just created.

In practice this ends up looking roughly like the code below.

```
cookie := http.Cookie{
    Name:  "cookie-name",
    Value: "cookie-value",
}
// Assuming w is an http.ResponseWriter
http.SetCookie(w, &cookie)
```

Note: This is an example and shouldn't be added to your application just yet.

¹<https://golang.org/pkg/net/http/>

²<https://golang.org/pkg/net/http/#Cookie>

³<https://golang.org/pkg/net/http/#SetCookie>

Box 11.1. Invalid cookies may be silently dropped

If you are ever having trouble with cookies not saving, be sure to double check that the cookie is valid. According to the net/http docs for the **SetCookie** function, “Invalid cookies may be silently dropped.”

This means that you may not ever see an error when an invalid cookie is saved, but instead your code might continue along as if nothing went wrong. This is obviously less than ideal, so be sure to take extra care when creating cookies.

Cookies have quite a few details we can configure, but at their core they are basically just a key and a value associated to that key. We will eventually talk about some of those other details in [Section 11.4](#), where we discuss securing our cookies, but until that point just know that our cookies aren’t secure and could be altered by end users.

Open up the users controller, where we are going to create our first cookie.

```
$ atom controllers/users.go
```

Find the **Login** method; we are going to be adding code to this method to create a cookie storing a user’s email address after we have successfully authenticated them. As I said before, this is NOT secure, but will help give us an idea of what we are working towards.

The first thing we need to do is tell our code to return if we can’t verify the user’s email address and password. We don’t want to accidentally set a cookie when the user hasn’t provided us with correct information, and the easiest way to do that is to return from our function.

```
func (u *Users) Login(w http.ResponseWriter, r *http.Request) {
    form := LoginForm{}
    if err := parseForm(r, &form); err != nil {
        panic(err)
    }
    user, err := u.us.Authenticate(form.Email, form.Password)
    if err != nil {
        switch err {
        case models.ErrNotFound:
            fmt.Fprintln(w, "Invalid email address.")
        case models.ErrInvalidPassword:
            fmt.Fprintln(w, "Invalid password provided.")
        default:
            http.Error(w, err.Error(), http.StatusInternalServerError)
        }
    }
    return
}
fmt.Fprintln(w, user)
}
```

Our code here is roughly the same logic as what we were using before, but we have wrapped the **switch** statement inside of an **if** block that will return if there are any errors. We then moved the **Fprintln** used when a user is authenticated correctly to occur outside of this if block.

Next we want to create a **Cookie**, using the key "**email**" and the value will be the email address of the user returned by the **Authenticate** method. This will occur after the switch statement in our code.

```
cookie := http.Cookie{
    Name:  "email",
    Value: user.Email,
}
```

And finally we need to call the **SetCookie** function to persist the cookie.

```
http.SetCookie(w, &cookie)
```

Putting it all together we get the code in [Listing 11.1](#), and in the next section we will look at how to view the cookies created by our **Login** method.

Listing 11.1: Creating our first cookie

```
func (u *Users) Login(w http.ResponseWriter, r *http.Request) {
    form := LoginForm{}
    if err := parseForm(r, &form); err != nil {
        panic(err)
    }
    user, err := u.us.Authenticate(form.Email, form.Password)
    if err != nil {
        switch err {
        case models.ErrNotFound:
            fmt.Fprintln(w, "Invalid email address.")
        case models.ErrInvalidPassword:
            fmt.Fprintln(w, "Invalid password provided.")
        default:
            http.Error(w, err.Error(), http.StatusInternalServerError)
        }
        return
    }
    cookie := http.Cookie{
        Name:   "email",
        Value:  user.Email,
    }
    http.SetCookie(w, &cookie)
    fmt.Fprintln(w, user)
}
```

Restart your server and log in again. You should see your user printed out to the screen, and the cookie will be silently saved in the background. Even though we can't see the cookie, we will need one saved before we try to view it in the next section.

Source code

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

Completed - book.usegolang.com/11.2

Changes - book.usegolang.com/11.2-diff

11.3 Viewing cookies

Now that we have created a cookie we are going to look at a couple ways to view cookies stored on your computer along with their values. In doing this we will see that cookies are both clearly visible and editable by end users, so the way we use cookies needs to take this into account.

We will start by first looking at cookies inside of a browser, and then we will look at how to do this using Go code.

11.3.1 Viewing cookies in Google Chrome

Note: If you aren't using Chrome or if your version of Chrome is different this guide should still be useful as a guideline, but the exact steps likely won't be the same. We will also look into how to view a cookie using Go code in the next section, so this isn't mandatory.

We will be looking at our cookie first inside of Googel Chrome. For this to work, you must have logged in and created a cookie inside of Chrome as well, so if you haven't be sure to do that first.

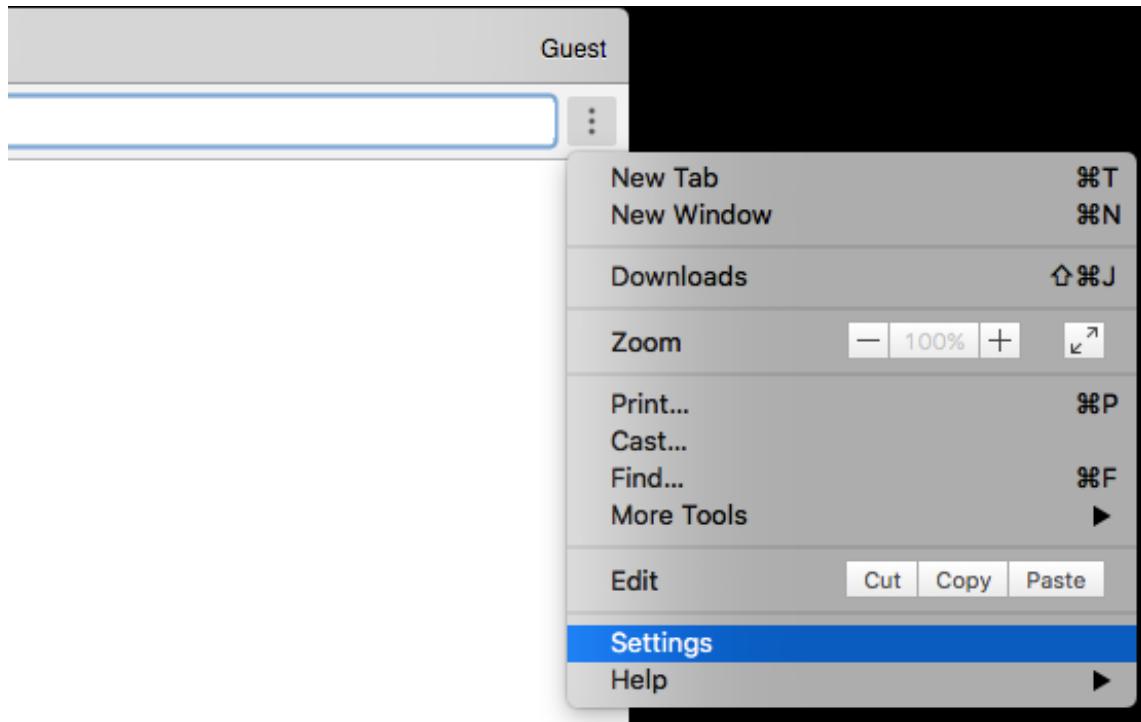


Figure 11.1: Opening Google Chrome Settings

The first thing you want to do in Chrome is to open your settings. This should be in the menu for Chrome, and you should also see three dots on the top right of the browser with a menu option for settings.

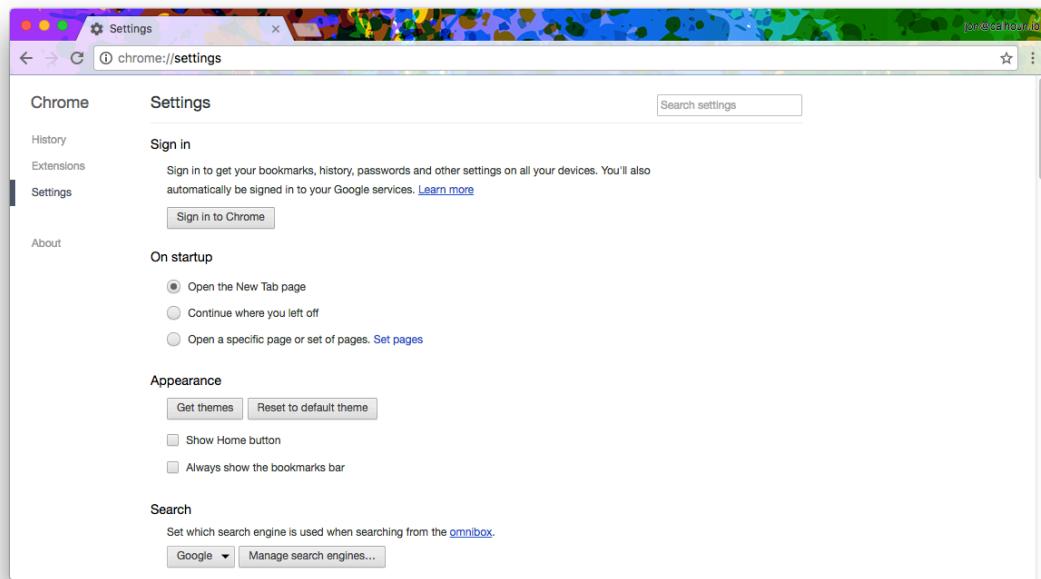


Figure 11.2: The Google Chrome settings page

Doing this should open up a page that looks pretty similar to Figure 11.2. Once on the settings page, type **cookie** into the search bar in the top right. This should update the page and you should see two buttons - one that reads “Content settings...” and another that reads “Clear browsing data...”.

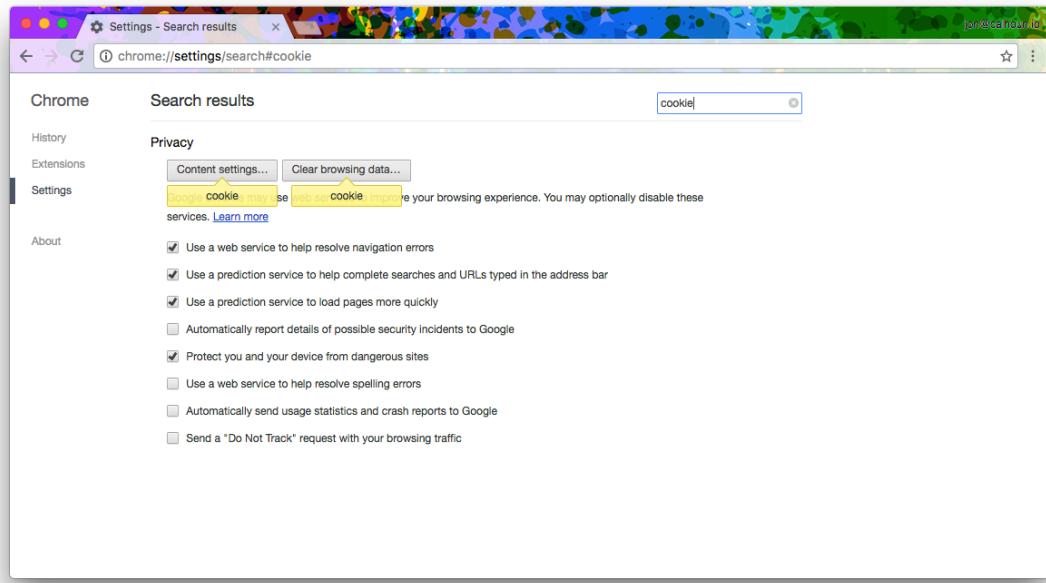


Figure 11.3: Chrome settings after searching "cookie"

Click on the button that reads “Content settings...”. This will open up a modal window that has a cookies section with a button that reads “All cookies and site data...”. This is shown in [Figure 11.4](#).

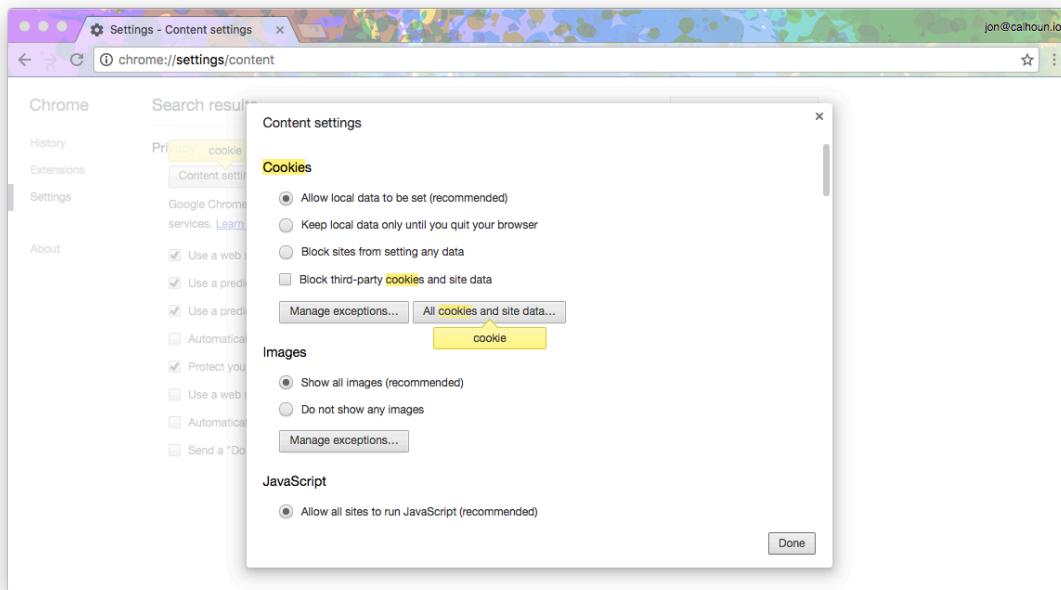


Figure 11.4: A modal showing cookie settings

Click on the button that reads “All cookies and site data...” and this will open up yet another modal that shows all of your cookies. Type **localhost** in the search bar in this new modal and it should filter your cookies to just ones that were created by localhost.

Select the cookie and look for a blue colored button that says “email”, then click on that button. The cookie should be expanded and you should see a bit more information about the cookie we just created.

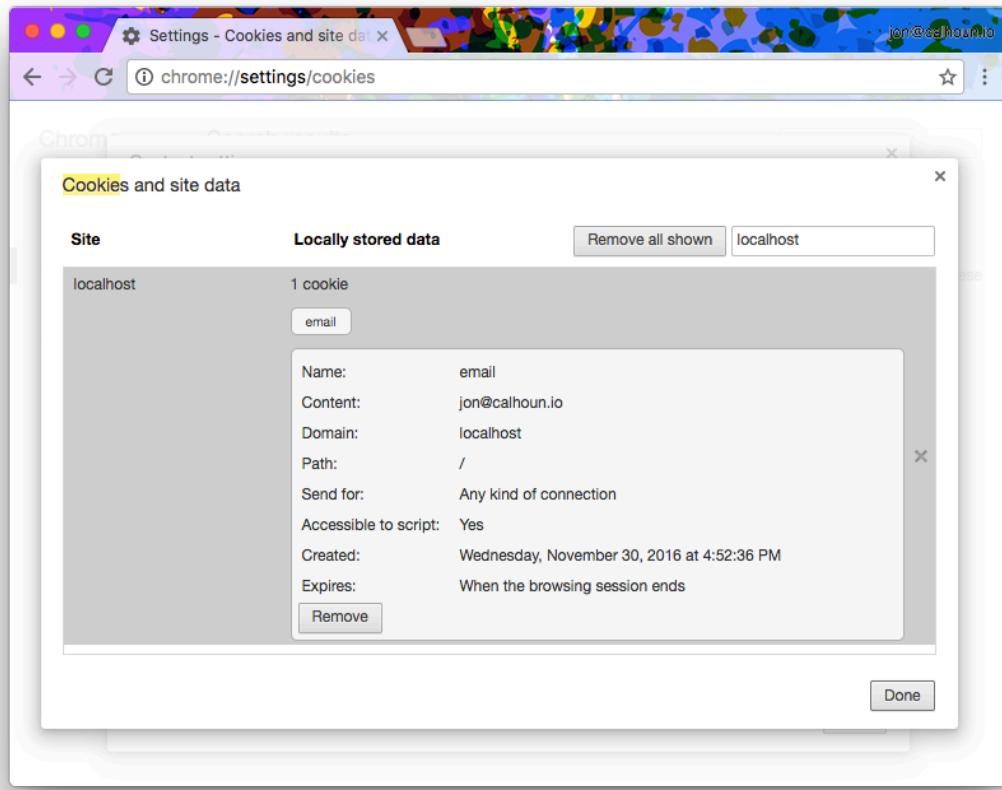


Figure 11.5: A modal showing the email cookie

A cursory examination of the cookie shows that this looks to be the right one. The name is “email”, like we expected, and the content is the email address we just signed in with.

On top of the information that we set, there appears to be some other information that we didn’t set. Most of these are set to default values, and are used to determine things like when the cookie expires, who can access the cookie, and whether or not the cookie requires a secure connection (SSL/TLS). We will explore some of these settings later in this book, but for now the defaults are fine.

11.3.2 Viewing cookies with Go code

Another way to read a cookie is to call the **Cookie**⁴ method implemented by the **http.Request** object. This gives us a way to access cookies that we have set in past web requests.

We are going to create an action on our users controller that will utilize the **Cookie** method and retrieve the cookie that we created in our **Login** handler. Once we have the cookie, we will then print it out to the screen so we can look at any of the information set on the cookie.

Open up the users controllers.

```
$ atom controllers/users.go
```

We are going to name our method **CookieTest**, so let's start by just writing out a stub for the method.

```
// CookieTest is used to display cookies set on the current user
func (u *Users) CookieTest(w http.ResponseWriter, r *http.Request) {
    // TODO: Implement this
}
```

Inside of our cookie test method we are going to call the **Cookie**⁵ method and pass it the name of the cookie we want. This is the **Name** field we defined when we initially created our cookie. If the cookie is located, we will get the cookie back and nil for an error, otherwise we will get the **ErrNoCookie** error which indicates that the cookie couldn't be located.

While not finding the cookie isn't necessarily an error in our web application, we are going to assume that it is for now and render an error page. Otherwise we will write the cookie's value to the response writer.

⁴<https://golang.org/pkg/net/http/#Request.Cookie>

⁵<https://golang.org/pkg/net/http/#Request.Cookie>

```
func (u *Users) CookieTest(w http.ResponseWriter, r *http.Request) {
    cookie, err := r.Cookie("email")
    if err != nil {
        http.Error(w, err.Error(), http.StatusInternalServerError)
        return
    }
    fmt.Fprintln(w, "Email is:", cookie.Value)
}
```

If we are going to use our new handler function we need to add it to our router. Open up your main.go source file and add a route for our cookie test. Let's use the path `/cookietest` and limit it to just the HTTP GET method.

```
$ atom main.go
```

```
func main() {
    // ... we are only adding the line below
    r.HandleFunc("/cookietest", usersC.CookieTest).Methods("GET")
    http.ListenAndServe(":3000", r)
}
```

Restart your server and log in if you haven't already. Then head over to <http://localhost:3000/> where you should now see your email address displayed on the screen.

```
Email is: jon@calhoun.io
```

We won't be shipping to production with this controller, but I find it useful to keep it around for a while as a way of verifying that our cookies are working as expected. Once you get the hang of everything you can delete it though.

Source code

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

Completed - book.usegolang.com/11.3.2

Changes - book.usegolang.com/11.3.2-diff

11.4 Securing our cookies from tampering

When we talk about cookie security, there are roughly five major attack vectors that we need to consider for our application.

1. Cookie tampering.
2. A database leak that allows users to create fake cookies.
3. Cross site scripting (XSS).
4. Cookie theft via packet sniffing.
5. Cookie theft via physical access to the device with the cookie.

In this section we are going to discuss the first of these attack vectors - cookie tampering - along with tactics we will be using to secure our cookies from tampering.

11.4.1 Digitally signing data

We mentioned it earlier, but one interesting side effect of cookies being stored on your user's computer instead of your web server is that they can be deleted. While this in itself isn't a security concern, what it implies is that we don't have ownership of the cookie. In other words, the end user has the freedom to edit those cookies and we can't do anything to stop them. This is known as cookie tampering.

Editing cookies stored on your computer is actually pretty easy to do, and there are several Chrome extensions that will allow you to do just that. Take [Edit-ThisCookie](#) for example - after installing this extension you could update the email address stored in your cookie in a matter of seconds.

If you are thinking “that doesn’t sound very secure...” then you are absolutely correct. As a general rule, all data stored in cookies shouldn’t be trusted unless you have done something to make it more trustworthy.

One way to do this is to store a [digital signature](#) of all of the data that you store in a cookie. The detail of how you do this can vary from case to case, but the general idea is always the same. Let’s imagine that you wanted to store the following data in a cookie.

```
{  
  "id": 123,  
  "email": "jon@calhoun.io"  
}
```

If that is all we decided to store, it would be pretty easy for an end user to alter the **id** or **email** and start accessing your website as another user. We don’t currently have any way to validate that the data is what we originally set in the cookie.

One way to solve this problem is to create a signature to also attach to the cookie using a hashing function like [HMAC](#). What makes this 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; You don’t need to worry about coding this, but look at how changing either of the secret key or the data being hashed will alter the resulting hash value.

```
HMAC("secret-key-1", { "id": 123 }) =  
60fc0ff9d3f4132cbe7b9dd12ef51837d8e...  
  
HMAC("secret-key-2", { "id": 123 }) =
```

```
b91c4b45a02dc9f92725d93ec4a7bfd6a4e...
```

```
HMAC("secret-key-1", { "id": 333 }) =  
78257ef3e303a46b934fccf7ed9c01d7f77...
```

What does this have to do with securing cookies?

When we digitally sign data, rather than just storing the data in the cookie, we would also store the resulting hash of that data. Now when a user returns to our website, they will not only be sending us a cookie with the data that we stored, but they will also be including a cookie with the signature that we created. If we want to validate that the data that the user has in their cookie is valid, we simply recreate the hash with the data in the user's cookie and make sure it matches the hash that we stored in the signature cookie.

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 new valid hash. When our code attempts to validate the data with the hash that the user provides it will not match the provided hash, and we will know that someone was tampering with the cookie data.

Box 11.2. JSON Web Tokens

If you have ever heard of [JSON Web Tokens \(or JWTs\)](#) they operate using a similar approach as what I described here. The data in the JWT is visibly by anyone, but editing the data isn't possible because not everyone knows the secret key used to generate the signature attached to the token.

11.4.2 Obfuscating cookie data

While digital signatures are incredibly neat and useful, we won't actually need them in our web application. I mention them in this book because it is possible that you will need them in the future, and I want you to be aware of safe ways to store data in cookies, but in the application we are building we will be taking other precautions to protect our sensitive data, and anything else that we store in cookies isn't sensitive enough to justify digitally signing it.

The approach we will be using is another one commonly used to prevent cookie tampering; We will be obfuscating our data so that it is nearly impossible for an attacker to fake the data.

The biggest issue with storing a user's ID or email address in their cookie is that it is really easy to figure out how to fake this. Take an ID for example - if your ID is 101, it is a pretty safe bet that there are users with IDs 1 through 100, and by changing your ID to any of these numbers you could attempt to take control of their account. Nearly every database creates IDs in sequence, so you don't have to try every number possible to find a user's ID. You only need to try numbers that are likely to have user IDs associated with them.

Similarly, email addresses are also easy to fake; You simply need to guess email addresses of other users and viola! You have access to an account that isn't yours.

Rather than storing data that is easy to fake, we are going to store a remember token that just looks like gibberish, and has no clear mapping to users. We are going to achieve this by generating a random string every time a user logs in, assigning that string to the user's remember token field. Then when the user logs in, we can find the user with the remember token, but the user won't be able to guess the remember token for other users.

Let's look at an example quickly. Imagine we have the following user:

```
User:  
ID: 1  
Email: jon@calhoun.io
```

The first thing we would do when this user logs in is create a random string for a remember token. We will be writing code to do this shortly, but for now assume that it is **0f008bc520e71**. Once we create this string, we would map it to a user and store that information in our database.

```
User:  
ID: 1  
Email: jon@calhoun.io  
RememberToken: 0f008bc520e71
```

The next thing we would do is set a cookie on the user with the name **remember_token** and the value **0f008bc520e71**. When the user logs in, we now have a way to look up the user in our database without using anything easily faked like an email address or an ID.

11.5 Generating remember tokens

In order to create remember tokens we need to do a few things, but the very first we need is a way to generate our remember tokens. As we mentioned in Section 11.4.2, we need to generate these randomly so that they can't be easily faked and lead to users pretending to be someone they are not. In this section we will focus on writing a package that will allow us to generate random strings that we can use as remember tokens.

To do this, we are going to need to cover some more advanced topics that might not be easy to grasp at first. Unfortunately, this isn't something we can easily simplify. Cutting any corners in this section could ultimately leave your application open to attacks, so we have to make sure our remember tokens are truly secure.

We will be writing this code in its own package, which we will be calling **rand**, so let's start by creating that directory and a source file.

```
# Make sure you are in the root directory
$ cd $GOPATH/src/lenslocked.com
$ mkdir rand
$ atom rand/strings.go
```

Our **rand** package will essentially be a wrapping around the **crypto/rand**⁶ package, and in it we are going to write the functions that make it easier for us to use the **crypto/rand** package.

Box 11.3. Wrapping packages

Wrapping packages is a pattern used by many Go developers as a way to take a more general package, like the **crypto/rand** package, and simplify it for their application by handling a lot of the application-specific logic inside of their wrapping.

In our particular case, creating a custom **rand** package means we can individually test the package and then easily use it throughout our code without worrying about the details of how we generate a random string.

Once you have created your directory and source file, we will start by declaring our package and stating our imports.

```
package rand

import (
    "crypto/rand"
    "encoding/base64"
)
```

⁶<https://golang.org/pkg/crypto/rand/>

There are multiple `rand` packages in Go, so be sure that your code is using the `crypto/rand` package if you are using an auto-importer! The `math/rand` package is NOT a secure way to generate remember tokens, but the `crypto/rand` package is because it is based on a cryptographically strong pseudo-random generator. This is done differently for each operating system, but the docs state:

On Linux, Reader uses `getrandom(2)` if available, `/dev/urandom` otherwise. On OpenBSD, Reader uses `getentropy(2)`. On other Unix-like systems, Reader reads from `/dev/urandom`. On Windows systems, Reader uses the `CryptGenRandom` API.

Understanding exactly what this means isn't necessary, but the short version is that `crypto/rand` package will help us generate random values safe for authentication purposes.

While we are looking at the documentation for the `crypto/rand` package⁷, we might also notice that it doesn't provide us with an obvious way to generate random strings. Instead, we are going to need to leverage the `Read`⁸ function (shown below) to create our strings.

```
func Read(b []byte) (n int, err error)
```

The `Read` function works by taking in a byte slice and filling it with random values. It then returns two values: an integer and an error. The integer will be equal to the length of the byte slice if there weren't any errors, but it could be smaller if there was an error. In that case it represents how many bytes were written before there was an error.

⁷<https://golang.org/pkg/crypto/rand/>

⁸<https://golang.org/pkg/crypto/rand/#Read>

Box 11.4. What is a byte?

A byte is a data type that stores 8 bits. In other words, a byte can be written as an 8 digit number consisting of only **0**s and **1**s. Learning about binary is outside the scope of this book, but it is important to know that a byte has 256 total possible values because of this.

While we can't use this directly to create a string, what we can do is fill a byte slice with random data and then encode that byte slice as a string using the **base64**⁹ package, then use the resulting base 64 encoded string.

Box 11.5. Base 64 encoding

When we generate a random byte slice using the `crypto/rand` package it isn't guaranteed that every byte will map to a valid UTF-8 character. When this happens you would get what looks like invalid characters in your string and it might not be persisted correctly in cookies and some databases.

This means we can't simply convert our random byte slice generated by the **Read** function into a string using conversion. Instead we need to use an encoding scheme that will take arbitrary binary data (like our byte slice) and encode it as a string. Base 64 is one such encoding scheme that is commonly used in programming. With base 64 encoding we can encode binary data into a string and then decode it back into the original byte slice anytime we want, although in this particular use case we won't have any reason to decode the data.

Let's start by writing a helper function that instead of requiring us to pass in

⁹<https://golang.org/pkg/encoding/base64/>

a byte slice will instead return one of a predefined length, or an error if one occurs.

```
// Bytes will help us generate n random bytes, or will
// return an error if there was one. This uses the
// crypto/rand package so it is safe to use with things
// like remember tokens.
func Bytes(n int) ([]byte, error) {
    b := make([]byte, n)
    _, err := rand.Read(b)
    if err != nil {
        return nil, err
    }
    return b, nil
}
```

The code we just wrote takes in an integer, **n**, and first creates a byte slice of that length. It then calls the **Read** function from the crypto/rand package, checks for errors, and returns the byte slice if there are no errors.

Next we need to write a function that will generate a random string using a specific number of bytes. This will utilize the **Bytes** function we just wrote, so the only new code will revolve around encoding the random byte slice using the **base64** package and then returning the results.

```
// String will generate a byte slice of size nBytes and then
// return a string that is the base64 URL encoded version
// of that byte slice
func String(nBytes int) (string, error) {
    b, err := Bytes(nBytes)
    if err != nil {
        return "", err
    }
    return base64.URLEncoding.EncodeToString(b), nil
}
```

As you can see, our code calls the **Bytes** function and captures the results. If there is an error, it returns an empty string and the error. Otherwise we use the base64 package to encode the byte slice into a string and return the results.

When encoding with the base64 package there are a few variables¹⁰ made available that offer specific encodings. We will be using the **URLEncoding** variable because I find this to be the safest option when you don't need a specific encoding.

Finally we are going to create a **RememberToken** function that simplifies the code for us a bit by using a constant number of remember token bytes. This isn't absolutely necessary, but I like to do it so that developers don't need to think about how many bytes they need in their remember tokens and accidentally use a number that is too small.

```
// I tend to put constants near the top of the file
const RememberTokenBytes = 32

// RememberToken is a helper function designed to generate
// remember tokens of a predetermined byte size.
func RememberToken() (string, error) {
    return String(RememberTokenBytes)
}
```

Note: We will discuss why we use 32 bytes in the next section.

Finally, we are going to test this all out. Open up the experimental main package we created earlier in the book and update the source code to test our our new package.

```
$ atom exp/main.go
```

```
package main

import (
    "fmt"
    "lenslocked.com/rand"
)
```

¹⁰<https://golang.org/pkg/encoding/base64/#pkg-variables>

```
func main() {
    fmt.Println(rand.String(10))
    fmt.Println(rand.RememberToken())
}
```

Run your code and verify that it is generating random strings and remember tokens as expected.

```
$ go run exp/main.go
```

You should see output similar to the example below, but because our strings are being randomly created yours will be different.

```
SJAh-V-ibhLyxQ== <nil>
scw6IJBlu3YPwNfj2taQw0h171yV_uaCmvtwtH8rIU8= <nil>
```

*Note: The **<nil>** in each of these is the error returned by our functions.*

Source code

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

Completed - book.usegolang.com/11.5

Changes - book.usegolang.com/11.5-diff

11.5.1 Why do we use 32 bytes?

It is important to note that in Box 11.4 we mentioned that a byte has 256 total possible values. As a result, if we were to update our remember token function

to only use one byte, it would only be able to create 256 unique tokens - one for each unique byte. If we were to increase the number of bytes used to 2 the number of unique tokens we could create would increase to 65,536 (256 x 256).

More generally, when we are using N bytes in our remember token function the total number of unique tokens we can create is equal to 256^N . When using 32 bytes this translates to $256 \times 32 = 1e77$ unique tokens, which is an incredibly large number. Do we really need that many potential tokens to choose from?

There are a lot of factors that go into picking the number of bytes to use for our remember token, but in my opinion the simplest way to understand this is to look at a simple game as an example.

Imagine that you and a friend are playing a game where you pick a number between 1 and 10, and your friend tries to guess the number that you picked.

Assuming you pick randomly, your friend has a $1/10$ chance of guessing the number correctly on the first try. If they get it wrong and guess again, they would have a $1/9$ chance of getting it right the second time.

More generally put, if your friend has N guesses before he loses, he has a $N/10$ chance of getting it right. So if they were able to guess 10 times, they would have a 100% (10/10) chance of getting it right. With 5 guesses they would have a 50% (5/10) chance of getting it right.

Now imagine that a few extra friends came over and wanted to join into the game, so you decide to change the rules a bit. There will still only be one person guessing, but everyone else will pick a unique number between 1 and 10 that nobody else has picked yet. For example, if you had four people playing in total you would have one person guessing, three picking numbers, and their choices might be 1, 3, and 5.

Now that you have 3 friends, it doesn't make sense to guess a single person's number. Instead, you decide that the guesser wins whenever he picks *anyone's* number correctly.

This drastically changes the odds of the game!

Now rather than having a **1/10** chance of getting a number right on the first guess you have a **3/10** chance. If you guessed three different numbers you would have over a 70% chance of guessing someone's number.

Guessing a remember token is similar to this game. When an attacker tries to guess a valid remember token and you only have one user you don't need a lot of unique values to make it challenging, but as you gain more users the odds of them guessing any single user's remember token increases drastically.

To make matters worse, attackers using computers to generate remember tokens can guess *much* faster than people can. Rather than taking 1 guess every few seconds, an attacker is likely to be making 10,000 or more guesses every second. Yikes!

Putting this all together, we need to pick a large enough set of remember tokens that an attacker can't easily guess a remember token even when they are guessing for many users and guessing thousands of times per second. 32 bytes allows us to choose from enough unique remember tokens that this isn't likely to happen.

11.6 Hashing remember tokens

While our remember tokens are hard to guess, we still have another issue to address. If we were to store raw remember tokens in our database and an attacker gained access to our database they would be able to manually create cookies and impersonate any of our users.

We obviously don't want this to happen, so rather than storing raw remember tokens in our database we are going to store the hash of each remember token similar to what we do for passwords, but rather than using bcrypt we are going to use another hashing function named HMAC.

Box 11.6. Why can't we use bcrypt for remember tokens?

In [Box 10.3](#) we discussed how bcrypt automatically adds a salt to our hash. This is great when hashing passwords where we intend to first lookup a user via their email address and then verify the hash, but for remember tokens this presents a problem.

When a user logs in via a remember token we won't be looking them up via their email address, but will instead be looking them up via their remember token. This means that we need to be able to generate the hash stored in our database *before* we do a query, but that isn't possible with bcrypt unless you know the salt that was used.

This presents us with a catch-22¹¹ where we can't look up a user account without knowing the salt, but we can't know the salt without first looking up the user account. As a result, we will be using the HMAC hashing algorithm instead.

Note: Technically, we could make bcrypt work by storing two cookies - one with a user's email address or ID that we use to look up their account, and then another with their remember token. We won't be doing that in this course because I find that it makes transitioning to multiple-sessions per user harder in the future.

11.6.1 How to use the `hash` package

Before we can jump into writing our own code, we must first learn how to use the `Hash`¹² interface provided by the `hash` package in Go.

¹²<https://golang.org/pkg/hash/#Hash>

```

type Hash interface {
    // Write (via the embedded io.Writer interface) adds more
    // data to the running hash. It never returns an error.
    io.Writer

    // Sum appends the current hash to b and returns the
    // resulting slice. It does not change the underlying hash
    // state.
    Sum(b []byte) []byte

    // Reset resets the Hash to its initial state.
    Reset()

    // Size returns the number of bytes Sum will return.
    Size() int

    // BlockSize returns the hash's underlying block size.
    // The Write method must be able to accept any amount
    // of data, but it may operate more efficiently if all
    // writes are a multiple of the block size.
    BlockSize() int
}

```

There are different ways to use this interface, but assuming we have some data we want to hash and an implementation of the **hash.Hash** interface, we would write code similar to below.

```

var data []byte
var h hash.Hash
h.Reset()
h.Write(data)
hashedData := h.Sum(nil)

```

We call **Reset** to ensure that the hash has been flushed of any data that was previously written to it. We then **Write** our own data to the hashing function so it knows what we want hashed, and finally we call **Sum** to request that the hashing function calculate a hash value for us, and we store the result in **hashedData**.

Now that we know roughly how to use the **hash.Hash** interface, we are going to look at how the **crypto/hmac** package works.

11.6.2 Using the `crypto/hmac` package

We discussed HMAC in [Section 11.4.1](#), but the general idea behind HMAC is that it uses a secret key to sign data, and then uses a hashing function to hash the signed data. In doing this, it makes it harder to figure out what data was used to construct the hash unless you know both the hashing function used and the secret key.

Doing this manually is possible, but Go provides us with the `crypto/hmac`[^crypto_hmac] package to make our lives a little easier. Inside of this package there is the `New` function, which takes in these two pieces of information - a function that returns a `hash.Hash` implementation and the secret key - and then returns an implementation of the `hash.Hash` interface.

```
func New(h func() hash.Hash, key []byte) hash.Hash
```

It is important to note that we have to provide a function that returns a hash here, not a hash itself.

What this means is that we can use the `crypto/hmac` package to construct a `hash.Hash` implementation, and then use it exactly like we discussed in [Section 11.6.1](#).

Looking at the rest of the docs for the `crypto/hmac` package, there is an example in the overview that illustrates how to use the `New` function provided by the `crypto/hmac` package:

```
mac := hmac.New(sha256.New, key)
```

In this line of code we are stating that we want to create an HMAC hash that is backed by the secret key stored in the `key` variable, and we intend to use the SHA256 hashing function once the data is digitally signed. The SHA256

hashing function here comes from the `crypto/sha256`¹³ package.

We are going to be writing code similar to this in our own application in order to hash our remember tokens in the next section.

11.6.3 Writing our own `hash` package

Much like our `rand` package, we are going to write our HMAC hashing code as a wrapper around a few existing packages. We will mostly be building a wrapper around the `hash` package we mentioned in Section 11.6.1 and the `crypto/hmac` package mentioned in Section 11.6.2, but we will be using a few other packages as well.

Because the primary package we will be wrapping is the `hash` package, and the primary purpose of our code will be to hash strings, we will name our package `hash`.

```
$ mkdir hash
$ atom hash/hmac.go
```

We will start off by stating our package name and declaring our imports.

```
package hash

import (
    "crypto/hmac"
    "crypto/sha256"
    "encoding/base64"
    "hash"
)
```

We touched on most of these packages in the past two sections, but not the `base64` package. When hashing data we get a byte slice as the result, and we

¹³<https://golang.org/pkg/crypto/sha256/>

will be using the `base64` package much like we did earlier - to base 64 encode those byte slices into strings.

Next we want to define our own `HMAC` type. We know that the `crypto/hmac` package's `New` function returns a `hash.Hash`, so we are going to embed that inside of our `HMAC` type, and then we are going to write a function that takes in a secret key and constructs an HMAC.

```
// NewHMAC creates and returns a new HMAC object
func NewHMAC(key string) HMAC {
    h := hmac.New(sha256.New, []byte(key))
    return HMAC{
        hmac: h,
    }
}

// HMAC is a wrapper around the crypto/hmac package making
// it a little easier to use in our code.
type HMAC struct {
    hmac hash.Hash
}
```

The last piece we need to write is a function that will take an input string, hashes it, and then returns the result as a string.

In [Section 11.6.1](#) we looked at how to use the `hash` package to generate a hash, and then finally we will use the `base64` package to encode the resulting byte slice and return it.

```
// Hash will hash the provided input string using HMAC with
// the secret key provided when the HMAC object was created
func (h HMAC) Hash(input string) string {
    h.hmac.Reset()
    h.hmac.Write([]byte(input))
    b := h.hmac.Sum(nil)
    return base64.URLEncoding.EncodeToString(b)
}
```

Box 11.7. Base64 encoding isn't the same as hashing.

It might appear that using a Base64 encoding scheme is similar to hashing, but it **is not the same as hashing**. When you Base64 encode something, anyone who has access to a Base64 decoder can determine your original data by simply running a decoder on the data.

You can try this for yourself. Head over to <https://www.base64decode.org/> and try decoding the data **R28gUm9ja3Mh**. You should get the string **Go Rocks!** after decoding the data. This works because Base64 just converts data to a different representation and back again - it doesn't actually hash or encrypt data.

Finally, let's test our package out. Open up your experimental **main.go** source file.

```
$ atom exp/main.go
```

Replace the contents of your source file with the following code.

```
package main

import (
    "fmt"

    "lenslocked.com/hash"
)

func main() {
    hmac := hash.NewHMAC("my-secret-key")
    // This should print out:
    // 4waUFc1cnuxoM2oUOJfpGZLGP1asj35y7teuweSFgPY=
    fmt.Println(hmac.Hash("this is my string to hash"))
}
```

Run it and verify that your output matches the output below. If you change the secret key your output will change, but as long as you have the same secret key it should be the same.

```
$ go run exp/main.go
```

```
4waUFc1cnuxoM2oUOJfpGZLGP1asj35y7teuweSFgPY=
```

Source code

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

Completed - book.usegolang.com/11.6.3

Changes - book.usegolang.com/11.6.3-diff

11.7 Hashing remember tokens in the user service

We have all the tools in our application to generate and hash remember tokens. The next step towards completing our authentication system is to update our user service to utilize these pieces when updating and creating users, and then to expose a function that allows us to lookup a user by their remember token.

This section is broken into several smaller subsections, and then we will test all of the changes in the final section using our experimental **main.go** source file.

11.7.1 Adding remember token fields to our User type

The first thing we need to do is update our user model. This is going to be very similar to how we updated it for the password fields - we need one field to store the raw remember token, and we need another to store the hashed value. And much like when creating our password fields, we also want to ensure that the raw remember token is never persisted in the database, but we do want to persist the hashed token.

First we need to open up our users model source file.

```
$ atom models/users.go
```

Next we need to add the fields to our **User** type.

```
type User struct {
    gorm.Model
    Name      string
    Email     string `gorm:"not null;unique_index"`
    Password  string `gorm:"-"`
    PasswordHash string `gorm:"not null"`
    Remember  string `gorm:"-"`
    RememberHash string `gorm:"not null;unique_index"`
}
```

The only piece different from how we setup our password fields is that we are going to add a unique index to our **RememberHash** field. This is done so that we can quickly lookup users via their remember token, but also to ensure that two users can't accidentally be given the same remember token.

Source code

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

Completed - book.usegolang.com/11.7.1

Changes - book.usegolang.com/11.7.1-diff

11.7.2 Setting a remember token when a user is created

Now that the **RememberHash** field is required for every user we need to update our code to set a remember token when users are created. Without this, we wouldn't be able to create users without manually setting a remember hash which is a pretty poor experience.

We are going to update our user service to automatically set the remember token whenever a user is created, and then later we will update our code to hash a remember token if it is set and store this in the user's **RememberHash** field, making it easy to update this field without knowing any of the specifics about how it is hashed.

Once again we will be working inside of the users model source file.

```
$ atom models/users.go
```

The first thing we need to add is an import for the **rand** package we created earlier.

```
import (
    "errors"

    // This is the only new import
    "lenslocked.com/rand"

    "github.com/jinzhu/gorm"
    - "github.com/jinzhu/gorm/dialects/postgres"
    "golang.org/x/crypto/bcrypt"
)
```

After that we need to find the **Create** method and add a snippet of code to check if the remember field is empty, and if so to set a remember token value. We won't be doing this when the remember token has a value set so that we don't overwrite any tokens that are already set.

```
func (us *UserService) Create(user *User) error {
    // ... this all stays the same

    if user.Remember == "" {
        token, err := rand.RememberToken()
        if err != nil {
            return err
        }
        user.Remember = token
    }
    // TODO: Hash the token and set it on user.RememberHash

    return us.db.Create(user).Error
}
```

At this point we would like to hash the remember token and store it in the **RememberHash** field, but we don't have access to an instance of the **HMAC** type we created earlier. We could create one every time we create a user, but that could prove to be problematic long term.

For starters, we would need to store the secret key we use in HMAC hashing somewhere that our **Create** method could access it every time we wanted to create an instance of the **HMAC** type. On top of that, we would be creating the same instance over and over again with no real benefit.

Instead, we are going to simply store an instance of the **HMAC** type as a field in our **UserService** so that all of our methods have access to it without needing to know what secret key was used or having to reinitialize the same thing over and over again. We will cover this in the next section.

Source code

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

Completed - book.usegolang.com/11.7.2

Changes - book.usegolang.com/11.7.2-diff

11.7.3 Adding an HMAC field to the UserService

We will still be working in the users model, so navigate to that source file if it isn't open already.

```
$ atom models/users.go
```

In order to access the **HMAC** type we created earlier we will need to import the **hash** package we created, so let's start there.

```
import (
    "errors"

    "lenslocked.com/hash"
    "lenslocked.com/rand"

    "github.com/jinzhu/gorm"
    _ "github.com/jinzhu/gorm/dialects/postgres"
)
```

Next we want to add the **hmac** field to the **UserService** type. We are going to make this an unexported field because we don't want it being changed or accessed by any code outside of our models package.

```
type UserService struct {
    db    *gorm.DB
    hmac hash.HMAC
}
```

The `hmac` field won't be very useful without being set, so our next step is to update the `NewUserService` function to handle doing this for us. In order to make this happen, we are going to need to add yet another package-level constant, but we will fix that before shipping our code to production. For now you need to be aware that this isn't a good practice with real secret keys because it could lead to security issues if your code repository was leaked.

```
// I prefer constants near the top of the source file
const hmacSecretKey = "secret-hmac-key"

func NewUserService(connectionInfo string) (*UserService, error) {
    db, err := gorm.Open("postgres", connectionInfo)
    if err != nil {
        return nil, err
    }
    db.LogMode(true)
    hmac := hash.NewHMAC(hmacSecretKey)
    return &UserService{
        db:   db,
        hmac: hmac,
    }, nil
}
```

In this code we create an HMAC object by calling `NewHMAC` with our secret key and then assign that object to the `hmac` field of the `UserService` we are creating.

Now moving forward we will have a way to hash our remember tokens.

Source code

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

Completed - book.usegolang.com/11.7.3

Changes - book.usegolang.com/11.7.3-diff

11.7.4 Hashing remember tokens on create and update

With all of the pieces put in place, we can now update the **Create** and **Update** methods on the UserService to hash a remember token if it is set.

Once again we will be working in the users model source file.

```
$ atom models/users.go
```

We will start with the **Create** method - find the comment we wrote earlier that reads:

```
// TODO: Hash the token and set it on user.RememberHash
```

Replace that comment with the line below.

```
user.RememberHash = us.hmac.Hash(user.Remember)
```

Next up is the **Update** method. We don't want to update the remember hash if the remember token wasn't updated, so we need an if statement in this situation. We don't need this in the **Create** method because we always create a new remember token when creating a user.

```
func (us *UserService) Update(user *User) error {
    if user.Remember != "" {
        user.RememberHash = us.hmac.Hash(user.Remember)
    }
    return us.db.Save(user).Error
}
```

With that last code change we are ready to start persisting remember tokens, but before we can make much use of them we need to write a function that allows us to lookup a user based on their remember token.

Source code

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

Completed - book.usegolang.com/11.7.4

Changes - book.usegolang.com/11.7.4-diff

11.7.5 Retrieving a user by remember token

We are still working in the users model, so make sure you have it open.

```
$ atom models/users.go
```

The next piece of code we are going to write is a new method on the UserService that allows us to pass in a raw remember token, and then returns either a user that has that remember token or an error. In order for this to work, our method is going to need to first hash the raw remember token so that it matches what we store in our database, and then query the database for a user with that remember token hash.

We have already seen how to do each of these individual pieces, so all that is left to do is put them all together.

```
// ByRemember looks up a user with the given remember token
// and returns that user. This method will handle hashing
// the token for us.
```

```
// Errors are the same as ByEmail.
func (us *UserService) ByRemember(token string) (*User, error) {
    var user User
    rememberHash := us.hmac.Hash(token)
    err := first(us.db.Where("remember_hash = ?", rememberHash), &user)
    if err != nil {
        return nil, err
    }
    return &user, nil
}
```

Note: It might seem a little odd that we are using `remember_hash` in our `Where()` method, but GORM uses the snake case¹⁴ of all fields when creating the columns in the database tables, so this is indeed the correct name to use.

We are returning the same errors as `ByEmail` because we are using the same `first` function. This makes it easy for us to return errors like `ErrNotFound` when a record isn't found, making it clear to users of the `ByRemember` method that we didn't run into an error but were simply unable to find a user with that remember token.

11.7.6 Resetting our DB and testing

The final piece of our puzzle is a test. This won't be a traditional test file, but we need to at least verify that our code works as we intended.

While doing this we are also going to reset our database using the `DestructiveReset` method, as any existing users will not have a valid `RememberHash` field and we haven't created a migration to handle this. Since our users are pretty useless without this field, it is simpler to just reset the database and start from scratch.

Note: If you forget to do this, you will probably see the following message in your logs:

¹⁴https://en.wikipedia.org/wiki/Snake_case

```
(pq: column "remember_hash" contains null values)
```

Interacting with our database requires us to copy a little bit of code from our main source file back into our experimental one. Specifically, we need the set of constants that we use to connect to our database. We will then need to rebuild our database connection string, setup a user service instance, and then interact with it.

Given that this is all either code we have seen or code we just wrote I am going to just show you the source code first and then walk over it some.

Open up your experimental source code.

```
$ atom exp/main.go
```

Then replace its contents with the following source, noting that you might need to update the constants to match your specific setup.

```
package main

import (
    "fmt"

    "lenslocked.com/models"
)

const (
    host      = "localhost"
    port      = 5432
    user      = "jon"
    password = "your-password"
    dbname   = "lenslocked_dev"
)

func main() {
    psqlInfo := fmt.Sprintf("host=%s port=%d user=%s "+
        "password=%s dbname=%s sslmode=disable",
        host, port, user, password, dbname)
    us, err := models.NewUserService(psqlInfo)
```

```

if err != nil {
    panic(err)
}
defer us.Close()
us.DestructiveReset()

user := models.User{
    Name:      "Michael Scott",
    Email:     "michael@dundermifflin.com",
    Password:  "bestboss",
}
err = us.Create(&user)
if err != nil {
    panic(err)
}
// Verify that the user has a Remember and RememberHash
fmt.Printf("%+v\n", user)
if user.Remember == "" {
    panic("Invalid remember token")
}

// Now verify that we can lookup a user with that remember
// token
user2, err := us.ByRemember(user.Remember)
if err != nil {
    panic(err)
}
fmt.Printf("%+v\n", *user2)
}

```

In this code we:

1. Reset our users table
2. Create a new user and allow our user service to set the remember token
3. Verify that the remember token was set, and panic otherwise
4. Search for a user via the remember token to verify the ByRemember method works and to verify that our remember token was persisted correctly

You can run it out to test it yourself by running:

```
$ go run exp/main.go
```

Source code

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

Completed - book.usegolang.com/11.7.6

Changes - book.usegolang.com/11.7.6-diff

11.8 Remembering users

In this section we are going how to use cookies to remember who a user is between requests, and how to use that information to restrict a user from access a page whenever they are not logged in.

We will start by writing a method that allows us to “sign in” a user. In other words, it will generate a remember token for the user if a raw one isn’t available, update the user account in our database with the new remember token, and finally it will store the remember token in a cookie so that we know who that user is on subsequent web requests. We will then call this method anytime a user creates a new account or logs in.

After that we will alter our cookie test function so that it looks up this cookie and redirects the user to the login page if they not logged in, otherwise it will print out the user’s account information to demonstrate that we now have access to the user account.

11.8.1 Storing remember tokens in cookies

The sign in method is going to be part of the users controller because we will be using it when users log in and create accounts.

```
$ atom controllers/users.go
```

Our sign in method is going to need to be provided two arguments: the current http ResponseWriter, and the user we want to sign in. We need the user so that we can update their remember token, and we need the response writer so that we can set the user's cookie.

```
// signIn is used to sign the given user in via cookies
func (u *Users) signIn(w http.ResponseWriter,
    user *models.User) error {
    // TODO: Implement this
}
```

While it is unlikely, we could run into an error when attempting to update our user with a new remember token so we will need to return those errors when they occur, hence the **error** return type.

Our implementation will start with a check to see if a raw remember token is set. If one is not set, we have no way to reverse our hashes so we will generate a new remember token. Otherwise we will use the remember token that is already set for the user.

When we do generate a remember token we will also want to update the user using our user service, so we will also make a call to the **Update** method and return any errors if there are any.

```
if user.Remember == "" {
    token, err := rand.RememberToken()
    if err != nil {
```

```

        return err
    }
    user.Remember = token
    err = u.us.Update(user)
    if err != nil {
        return err
    }
}

```

After exiting this if statement we know that the **user** variable should have a value in the **Remember** field that isn't an empty string. If it was an empty string our code inside the if statement would have generated a new remember token, and otherwise we know one was already set.

We are going to use that remember token to create a cookie and set it using the **http** package along with the response writer passed into our **signIn** method. Finally we will return **nil** because we didn't have any errors if we made it this far.

```

cookie := http.Cookie{
    Name:  "remember_token",
    Value: user.Remember,
}
http.SetCookie(w, &cookie)
return nil

```

Putting it all together we get the method shown in Listing ??.

Listing 11.2: Complete **signIn** method

```

// signIn is used to sign the given user in via cookies
func (u *Users) signIn(w http.ResponseWriter,
    user *models.User) error {
    if user.Remember == "" {
        token, err := rand.RememberToken()
        if err != nil {
            return err
        }
        user.Remember = token
        err = u.us.Update(user)
    }
}

```

```

    if err != nil {
        return err
    }
}

cookie := http.Cookie{
    Name: "remember_token",
    Value: user.Remember,
}
http.SetCookie(w, &cookie)
return nil
}

```

Our next step is to update the log in and sign up methods to call this method after a user successfully creates an account or logs in. We will also go ahead and redirect users to the cookie test page so that we can verify a cookie was properly set without having to manually type in the URL.

Updating the **Create** method is easier as we only need to add some new code and remove the single **fmt.Fprintln** at the end of the method.

```

func (u *Users) Create(w http.ResponseWriter, r *http.Request) {
    // ... This code remains unchanged

    // Delete this line:
    // fmt.Fprintln(w, "User is", user)

    // Add the following at the end of the Create method
    err := u.signIn(w, &user)
    if err != nil {
        // Temporarily render the error message for debugging
        http.Error(w, err.Error(), http.StatusInternalServerError)
        return
    }
    // Redirect to the cookie test page to test the cookie
    http.Redirect(w, r, "/cookietest", http.StatusFound)
}

```

In the **Login** method we will need to remove any code used to create a cookie and the print statement at the end of the method. We don't need to manually create the cookie anymore because it is handled by the **signIn** method.

```
func (u *Users) Login(w http.ResponseWriter, r *http.Request) {
    // ... Most of the code remains unchanged

    // Delete this code
    // cookie := http.Cookie{
    //     Name: "email",
    //     Value: user.Email,
    // }
    // http.SetCookie(w, &cookie)
    // fmt.Fprintln(w, user)

    // Add this code at the end of the method
    err = u.signIn(w, user)
    if err != nil {
        http.Error(w, err.Error(), http.StatusInternalServerError)
        return
    }
    http.Redirect(w, r, "/cookietest", http.StatusFound)
}
```

Finally, restart your server and try creating a new account and signing in. At this point we just want to verify that there aren't any errors, as we won't be able to properly test our remember token cookies until we update our cookie test.

Note: The cookie test method currently reads an older cookie, so if you are wondering why this doesn't match any new accounts you create or log in as this is probably why. We will fix this in the next section.

Source code

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

Completed - book.usegolang.com/11.8.1

Changes - book.usegolang.com/11.8.1-diff

11.8.2 Restricting page access

Next we need to write some code to restrict a user's access when they try to visit pages and either aren't logged in. Long term we will want to redirect them to the login page when this happens, but for now we are going to do this by showing an error message so that we can verify our code is working as expected.

As luck would have it, our cookie test already has about half of the code we are going to need to do this. In the `CookieTest` method we already handle reading a cookie and checking for errors, so the first thing we want to do is update this code to read the new remember token cookie we created.

Open the users controller and head to the cookie test method.

```
$ atom controllers/users.go
```

Update the `CookieTest` method to read the “`cookie_test`” cookie instead of the “`email`” cookie.

```
func (u *Users) CookieTest(w http.ResponseWriter, r *http.Request) {
    cookie, err := r.Cookie("remember_token")
    if err != nil {
        http.Error(w, err.Error(), http.StatusInternalServerError)
        return
    }
}
```

Next we are going to use this cookie's value to search for a user in our database. To do this, we will use the `us` field of our users controller, which is a `UserService` instance. We will be using the `ByRemember` method on the user service that we wrote earlier in this chapter.

Add this code after the cookie is retrieved and the error is checked. In short, add it right below the code we just wrote.

```
user, err := u.us.ByRemember(cookie.Value)
if err != nil {
    http.Error(w, err.Error(), http.StatusInternalServerError)
    return
}
fmt.Fprintln(w, user)
```

If there is NOT an error when calling the `ByRemember` method we will have successfully retrieved a user from our database, so we will print that user out so we can manually verify it is the correct user.

If there is an error, our if statement will execute and we will print out an error message telling us a little more information about what happened. In production this isn't ideal because the error message might contain sensitive information, and because a better user experience would involve redirecting them to the login page, but for development this is fine. We will eventually fix this code before shipping to production.

Restart your server and verify that logging in and creating an account will both set a new remember token cookie and that you are properly recognized in the updated cookie test code.

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

If you want to verify the page works without a cookie being set or with an invalid one, try editing or deleting your cookie or visiting the cookie test page in an incognito browser window.

Source code

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

Completed - book.usegolang.com/11.8.2

Changes - book.usegolang.com/11.8.2-diff

11.9 Securing our cookies from XSS

XSS stands for cross site scripting[^xss], and it is a type of attack where nefarious parties attempt to inject JavaScript code into a web application and have it run as if the web application provided it.

For example, I might set my username as the following on a website:

```
<script>alert("hi!");</script>
```

While this isn't a very great username, if the website doesn't render it correctly it could end up being parsed by the browser as HTML which would mean the JavaScript alert would get run.

For something like an alert this is a minor annoyance, but if the JavaScript code were to create web requests to our server the user's cookies would be attached to those requests by default. This means that the JavaScript could impersonate the user without the user ever knowing.

In this case my username might instead be something like:

```
<script>
// code to make web requests to give me access to all of
// your resources
</script>
```

Now when you visit my user profile, the code would get executed and I would have access to all of your resources. Maybe those are private documents, private

code, or something else entirely. The important part here is that an attacker was able to make web requests on your behalf.

We are using the `html/template` package to mitigate this, so it isn't likely this will happen to us, but it is always best to be as safe as possible. If you don't plan on using a JS framework, or if you don't intend to make web requests with your JavaScript, you really don't need to give scripts access to your cookies, especially your authentication cookies. Instead, it would be wise to limit cookies to only browser-initiated web requests, which we can do!

Box 11.8. Only loosen security when necessary

While it might be tempting to skip this in case you decide to use a JavaScript framework in the future, I am going to urge you not to do that.

In general, it is much safer to err on the side of caution and keep as many restrictions as possible on things like cookies until it becomes absolutely necessary to remove a restriction. There are [countless examples](#) on the internet of sites that weren't hacked because of a single vulnerability, but because the stars aligned and three or four things that are insignificant on their own all happened to lead to the perfect storm for an attacker.

Our application won't have any need for accessing cookies with scripts so we are going to restrict our cookies. If that ever changes, it is incredibly easy to update and you would only need to update a few lines of code.

We will be making our changes to the `signIn` method of our users controller, so let's start by opening the source file.

```
$ atom controllers/users.go
```

Find the cookie we construct in the `signIn` method and update it to include the value `true` for the `HttpOnly` field.

```
cookie := http.Cookie{
    Name:      "remember_token",
    Value:     user.Remember,
    HttpOnly:  true,
}
```

This will tell our cookie that it is not accessible to scripts. If you were to view the cookie in Google Chrome like we did earlier, you would see something like this in the cookie:

```
Accessible to script: No (HttpOnly)
```

This means that the cookie isn't accessible for scripts run in the browser.

Source code

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

Completed - book.usegolang.com/11.9

Changes - book.usegolang.com/11.9-diff

11.10 Securing our cookies from 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, and this occurs when someone manages to intercept your web requests and steal data from them before forwarding them along to where they were intended to go. This one is tricky because you might not even realize someone stole your data if they forward your request along and you continue to get a response as expected.

The second, theft via physical access, is exactly what it sounds like. It occurs when someone has physical access to your computer, laptop, or some other hardware and steals a cookie from it.

The only real solution to the first type of theft is to make sure packet sniffing isn't a viable option for attackers, which means making sure your application is served over SSL/TLS. This is why so many people are pushing for always using **https** on your website these days - it is relatively easy to do now, can make your website faster, and helps prevent these hard-to-detect attacks.

If your website doesn't have **https** (the **s** is the important bit) in the URL bar when people visit it, you are not secure from this. We will discuss how to do this later when we talk about deploying our application to production, but keep this in mind if you ever consider alternative deployment methods than what we discuss in this book as you will want to make sure your site is secure.

Box 11.9. A cookie theft example

While theft of cookies via packet sniffing might not seem like a serious concern, a developer named Eric Butler shed a lot more light on the issue when he published a Firefox extension named [Firesheep](#) which demonstrates just how easy this attack is to administer.

With the extension installed, all anyone needs to do is connect to a busy wifi network. If anyone else on that network visits a site that Firesheep recognizes and does so over an insecure connection, Firesheep would then show the user's name and the service they connected to on the left. After that, all an attacker needs to do

to log in as that user is double click on the user and Firesheep would instantly log them in as that user.

Lastly, we have physical theft which we really can't prevent on our end. If someone manages to access someone's physical hardware, it is virtually impossible to detect, but what we can do is make it easy for a user to invalidate all existing sessions. Luckily, our application already does this by changing the **RememberToken** every time a user logs in. All a user needs to do is log in with a new device and those cookies will be invalidated.

Note: There are alternatives to this approach that are just as secure, but most of these involve using session expirations and are a little more complicated.

11.11 Preventing CSRF attacks

CSRF attacks are a potential attack that occurs when a website creates a form that submits false data to a different website. For example, a nefarious site might have a form that submits a fake money transfer to your banking website.

This works because from the server's perspective it looks like you initiated the web request yourself because you clicked a button and submitted the form.

To prevent this we will eventually implement something known as CSRF tokens, but that won't happen until later in the book when we prepare our application for production. I am telling you about it now so you understand why you shouldn't deploy your application until then.

11.12 Exercises

At this point most of your authentication system should be in place, so it is time to review a few exercises to test your understanding of the material.

11.12.1 Ex1 - Experiment with redirection

In our cookie test we show an error when the user isn't logged in. Try to update it or another handler function to redirect the user to the login page instead when this happens, much like we will in our final version of the application.

11.12.2 Ex2 - Create cookies with different data

Cookies can be used for storing all sorts of data so long as it isn't sensitive data. For example, if you need to redirect a user to the login page, you could store the page they were coming from and send them back to that page after they log in. Try creating a few different cookies and validating that they are being set correctly.

11.12.3 Ex3 - [HARD] Experiment with the hash package

We used the hash package in the standard library a bit, but we didn't get to explore it too much. Try experimenting with different portions of the standard library's hash package and get a better sense for what is available there.

Chapter 12

Normalizing and validating data

In this chapter our primary focus is going to be validating and normalizing our data. In the process we will also look at cleaning up our code a bit and coming up with a nice code structure, but this will all be done in an effort to improve our validation code.

Before we can really jump into writing any code we need to first discuss what validation and normalization are, so we will start there.

What are validation and normalization?

Validation can mean different things to different people, but in the context of web development it most commonly refers to verifying that data is in an acceptable state. For example, we might want to verify that email address is valid, or that it is not taken by another user. Or we might want to verify that every user has a password with at least eight characters in it.

Similarly, normalization is a term that can mean different things depending on

the context; when we look at normalizing our data we are talking about making sure our data is in the format we expect, and if not attempting to transform it into that format. For example we might verify that a phone number matches the format we expect before persisting it in our database, or we might convert all email addresses to their lowercase equivalent.

The reasons for using validation is much more obvious than normalization. We clearly need to verify that data is valid, but why do we care about the format of phone numbers? Does it really matter?

Imagine we were storing each user's phone number in our database. If we didn't normalize the data, we might end up with a table with data similar to below.

user_id	phone_number
1	(123) 456-7890
2	123-456-8989
3	1234569999

Now imagine that we wanted to search for a user by their phone number. Perhaps we received a support call, and we need to know which account is linked to that phone number. When we go to search we run into a problem - how do we search for a phone number when they are all stored in a different format? Despite that phone number "1234567890" existing in our database, searching for it in that format wouldn't return any result because it is in a different format.

If we were to instead normalize our data, we could write code that takes in any format like above and trims out everything except for the digits. That would leave us with a database that is much easier to search.

user_id	phone_number
1	1234567890
2	1234568989
3	1234569999

It also makes other features, like email formatters, easier to write, because they can always assume the data is in a consistent format. Converting “1234567890” to “(123) 456-7890” is fairly easy to do, but if we had to deal with many different formats and convert each to the “(###) ###-####” format it would become much harder.

Note: When we talk about normalization we ARE NOT talking about database normalization that you might hear about when learning SQL. That form of normalization revolves around avoiding duplicate/unsynced data in your database. We won't be covering that here.

While validation and normalization are not the same, the two are very tightly linked. For starters, it is very hard to separate the two entirely. Whenever we want to verify an email address isn't taken we need to first normalize it, then search our database for a user with that email address. If we didn't normalize first we might end up trying to create two users with the same email address. In this case we are doing normalization first, then validation, but there are also cases where we must validate data first, then normalize it. One instance of this is when validating a password has a minimum length; we can't tell how long an original password is once it is hashed (normalized), so we need to validate it first, then normalize it.

This tends to lead to a common pattern where we don't perform one then the other, but instead create a sort of sandwich:

1. Some normalization
2. All validations
3. Remaining normalizations

As a result of the tight dependency between the two, we will be keeping all of this code together in our application.

12.1 Separating responsibilities

At this point our models package doesn't have any real isolation. That is, our UserService type contains pretty much all of our code related to interacting with users regardless of what it is.

We have some code in the UserService that handles reading and writing to our DB using GORM. We have some code that does basic validations, like verifying that an ID is not 0 when we go to delete a user. We even have some normalization happening when we hash our users' passwords prior to storing them in the database.

Short term this was a good approach because we weren't doing a lot. Our UserService was relatively small, so splitting it up too much wouldn't have been very useful, but as we begin to add more validations and expand our application we will find that this stops being the case. Instead, we are going to want to separate our code so that we can focus on smaller pieces at a time and write less error-prone code.

As we start to split our code into "layers", each will have its own isolated set of responsibilities. This isn't exactly a new concept, and in fact we have been doing that already in our application by using MVC. Models, views, and controllers each have different responsibilities and by separating this code we don't have to keep our entire application in our head as we make changes, we only need to think about the piece we are editing.

We will be using the same idea, except our layers this time will all be within the same package - the models package. We will be doing this using different struct types, each with their own set of responsibilities like interacting with the database or validating data.

Note: While this isn't the only design option, it is the one that I find fits most situations well which is why we are using it in this course.

Box 12.1. There isn't a "one size fits all" design pattern

It would be great if there were, but unfortunately there is no such thing as “one size fits all” when it comes to code design. What might be ideal for one application could easily turn out to be horrific for another. This sentiment applies to nearly all design patterns, including MVC.

While most of the things we learn in this book are meant to be applicable in most use cases, keep in mind that it is okay to try other code structures and design patterns if they don’t seem to be meshing well with your project. I do suggest that you try these patterns first, especially on your first pass through the course, but experimenting is a great way to expand your knowledge. Even if a new pattern doesn’t work well, it isn’t a waste of time as long as you learn from the experience.

12.1.1 Rethinking the models package

In our current code, our **models** package exports two primary types.

```
type User struct { ... }
type UserService struct { ... }
```

The first, **User**, is used to represent the data we store in our database. It doesn’t have methods attached to it, and is instead passed into other functions and methods.

The second, **UserService**, provides us with methods to create, update, and otherwise interact with users. In short, this defines everything that can be done with a user. When someone is just getting started with our models package, they will likely want to start there to discover what operations are available to them.

[H]

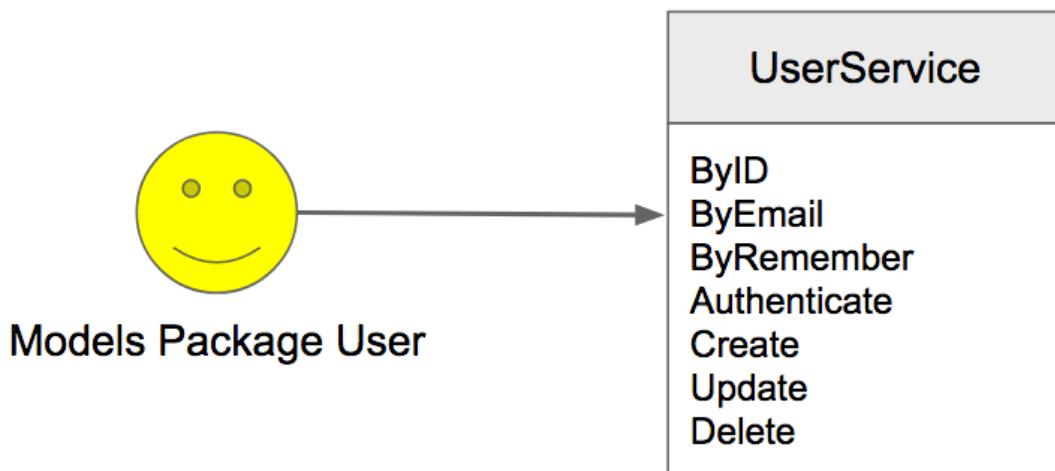


Figure 12.1: Our current models package design

With our current design, the `UserService` is a struct type and it implements every method that it exposes. That is, none of this work is deferred to say a database layer, or a validation layer. Our `UserService` handles all of this.

As we mentioned earlier, we are going to start splitting this into many layers. In doing this, our user service will no longer be responsible for doing all of this work but will instead defer much of it to other layers. For example, if we introduced a `UserReaderWriter` layer we might end up with a design like the one shown in [Figure 12.2](#).

In this design our `UserService` is still a collection of all of the actions possible to take on a user, and any users of our models package would still interact directly with the `UserService`, but that `UserService` doesn't implement all of that code on its own. Instead, when a user calls a method like **ByID** our `UserService` might pass that method call off to the `UserReaderWriter` layer which is responsible for interacting with the database. Other methods, like `Authenticate`, might be

[H]

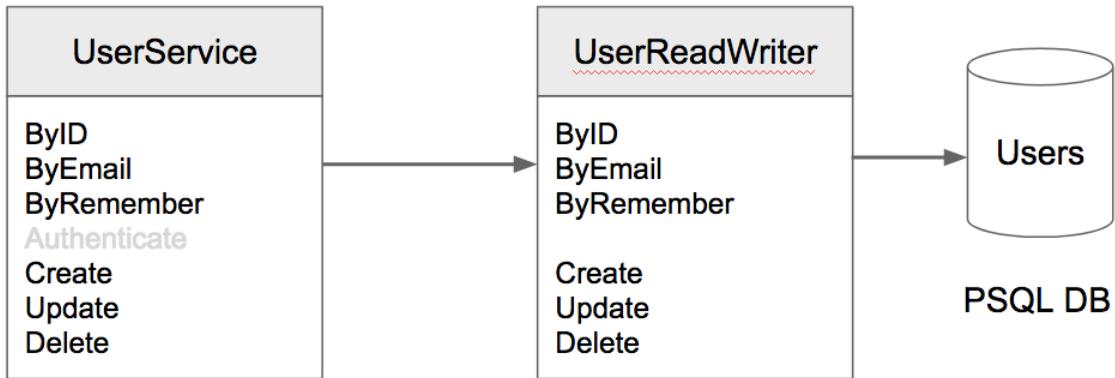


Figure 12.2: Potential models package design

implemented by the `UserService` or another type altogether.

Where the code gets implemented isn't important to our end user. What is important is that we provide our user with a single, easy-to-use `UserService` that defines all of the user related actions they can take perform.

The primary reason for splitting up this code is to make it easier for us to manage our code. Instead of needing monstrous `UserService`, we are instead able to break our code up into smaller pieces that are easier to think about and test.

Another benefit to this approach is that we can change implementation details, possibly adding new layers, new validations, or caching, and any code that uses our `UserService` should continue to work without any changes because we aren't changing the `UserService` itself.

The design we will be using is slightly different from [Figure 12.2](#); we will be breaking our code into three layers of responsibility.

1. The database interaction layer. This is responsible for reading and writing data from the database, but DOES NOT handle validation or anything else.

2. The validation and normalization layer. This is responsible for verifying and transforming data into the state that our database layer expects, and returning errors when data is invalid.
3. The topmost layer that developers interact with. This layer will serve two purposes; first it will define the contract to users of our `models` package. That is, it will define all of the methods that they can use to interact with users. Its second purpose will be to expose some common functionality. For example, authenticating a user can be done using nothing but the other two layers, but we will wrap this common functionality in an `Authenticate` method here.

If we were to start splitting our code into these layers now, we might end up with code that looks something like this:

Listing 12.1: An example of layers with structs

```
type UserService struct {
    userVal
}
func (us *UserService) ByID(id uint) (*User, error) {
    return us.userVal.ByID(id)
}

type userVal struct {
    userGorm
}
func (uv *userVal) ByID(id uint) (*User, error) {
    // Validate the ID first...
    if id <= 0 {
        return nil, errors.New("invalid ID")
    }
    // If the ID is valid, pass it to the userGorm layer
    return uv.userGorm.ByID(id)
}

type userGorm struct {
    db *gorm.DB
}
func (ug *userGorm) ByID(id uint) (*User, error) {
    // We DO NOT need to validate data because it is already
    // validated
    var user User
    return user, nil
}
```

```
err := first(ug.db.Where("id = ?", id), &user)
if err != nil {
    return nil, err
}
return &user, nil
}
```

Note: This code is for illustration purposes, and you do not need to add it to your application.

The topmost layer is defined by the `UserService`. Below that we have (2), the validation layer, which is defined by the `userVal` type. Then below that we have (3), the database interaction layer, which is defined by the `userGorm` type.

When we call a method like `ByID`, the method may end up traversing through all of these layers, but each of these layers isn't required to actually take any action. For example, the `UserService` doesn't actually modify the data or do any real work. It simply passes the data along to the validation layer, which in turn validates the data and then calls the database interaction layer.

This change might seem insignificant at first, but it can have a major impact on your development. For example, now when we write our database interaction layer we can always assume that the data we have is 100% valid; if it wasn't, the validation layer would have rejected it already.

Being able to make assumptions like this makes writing code easier and faster because we get to focus on a single thing at a time. We don't need to ask ourselves what a valid ID is, or what error we should return if an ID isn't valid. We simply need to create a database query and execute it.

An additional perk is that as we can easily change out our database implementation or add caching without changing most of our code. All we would need to do to add caching to our application is add a new layer between the validation and database layer that acts as a cache.

Unfortunately this approach has a flaw - by using static types our implementation becomes very brittle. We can't optionally choose to use some layers in

some cases, and exclude them in others. In order to do this we are going to introduce a few new interfaces to our application, but first we will take a moment to explore some of the pros and cons of interfaces and how chaining works.

12.1.2 Static types vs interfaces

Note: None of the code in this section will actually be used. It is instead meant for illustration purposes.

In this section we are going to take a break to discuss some of the pros and cons of using interfaces, as well as what interface chaining is and some of its benefits. If you are familiar with interfaces you can skip this section. If you are completely unfamiliar with interfaces, I suggest you check out the “Using interfaces in Go” section in Appendix 18.1, then come back here.

In Listing 12.1 we looked at how we might define our layers using structs, and we spoke briefly about how this design makes for a brittle implementation. We will start this section by exploring what exactly makes this implementation brittle and then we will discuss how interfaces can remedy these issues.

We will start by looking at the code necessary to instantiate the structs we looked at in Listing 12.1.

```
db, _ := // connect to a GORM DB
UserService{
    userVal: userVal{
        userGorm: userGorm{
            db: db,
        },
    },
}
```

At first glance there isn’t anything wrong with this code. We simply declare each of our layers and construct them inside of one another. Where this starts to become brittle is if we wanted to add a new layer, or remove an existing layer.

For example, let's say we wanted to add a cache between the `userVal` and `userGorm` layers. How would we go about doing that?

We would need to completely change our `userVal` type in order for this to work, updating our code to instead be something like this:

```
type userVal struct {
    userCache
}
func (uv *userVal) ByID(id uint) (*User, error) {
    // Do validations first...

    // We now use the cache instead of the DB layer
    return uv.userCache.ByID(id)
}

type userCache struct {
    userGorm
    // plus other caching data
}
func (uc *userCache) ByID(id uint) (*User, error) {
    // I am intentionally skipping any caching logic here for
    // simplicity.
    return uc.userGorm.ByID(id)
}
```

We had to completely overhaul the `userVal` type to instead have an embedded `userCache` instead of our database layer, and now our code only works if we have a cache layer present. We cannot use it without a cache.

We could fix this by adding a bunch of optional code that checks for a cache, but that would end up adding a lot of complexity to our code and it would become chaotic to maintain.

```
func (uv *userVal) ByID(id uint) (*User, error) {
    // Do validations first...

    // use the cache if we have one
    if uv.userCache != nil {
        return uv.userCache.ByID(id)
    } else { // otherwise use the DB layer
        return uv.userGorm.ByID(id)
    }
}
```

```
    }
```

Rather than doing this, we are going to look at how interfaces, along with interface chaining, allow us to write this code in a way that each of our layers doesn't actually care about which layer comes after it. All each layer will have to do is perform its own set of work, and then call the next layer.

Box 12.2. Interface chaining

Interface chaining is pretty much exactly what it sounds like - it is writing code that effectively creates a “chain” of interface implementations.

For example, imagine we started with the **Speaker** interface:

```
type Speaker interface {
    Speak() string
}
```

To implement this interface, we simply need to create a type with the **Speak** method.

```
type Dog struct{}

func (d Dog) Speak() string {
    return "woof"
}
```

In order to chain this, we would simply need to create a second Speaker that is meant to wrap an existing speaker and create a chain.

```
type PrefixSpeaker struct {
    s Speaker
}

func (ps PrefixSpeaker) Speak() {
    return "prefix: " + ps.s.Speak()
}
```

With this we could now create a chaining effect where we wrap a Dog with a PrefixSpeaker which would add a prefix to any strings returned by the Speak method.

```
speaker := PrefixSpeaker{
    s: Dog{},
}
// This prints out "prefix: woof"
fmt.Println(speaker.Speak())
```

What is happening is that our initial call to the Speak method calls the PrefixSpeaker's method, but inside of this method the Dog type's Speak method is called, causing a sort of chain of method calls.

While our example is fairly trivial, method chaining can be very powerful in Go. It is used extensively in packages like `io` and `net/http`, and we will be using it later in this course to create middleware that verifies a user is logged in and has permission to access a gallery before calling the controller to render that gallery.

To hear more about interface chaining, check out this talk by Francesc Campoy - <https://www.youtube.com/watch?v=F4wUrj6pmSI&feature=youtu.be&t=1337>

Box 12.3. Embedding interfaces

Related to interface chaining is embedding. There are two types of embed-

ding in Go - embedding interfaces when defining an interface, and embedding fields in a struct. You can read more about both of these in Effective Go - https://golang.org/doc/effective_go.html#embedding.

What we are going to discuss in this section revolves around embedding fields inside of a struct. Specifically, we are going to look at embedding an field with an interface type inside of a struct.

```
type Speaker interface {
    Speak() string
}

type DemoSpeaker struct {
    // Speaker is embedded in the DemoSpeaker
    Speaker
}
```

Embedding is occurring here because we didn't define a field name for the **Speaker** field. Instead, the field name is defaulted to the same name as the type. That means that right now if we had a DemoSpeaker instance, the following two lines of code would effectively be the same.

```
demoSpeaker.Speak()
demoSpeaker.Speaker.Speak()
```

As long as we haven't defined a **Speak** method on the DemoSpeaker, the first line will be the same as the second in our code. But more importantly, by embedding the Speaker type, our DemoSpeaker now implements the Speaker interface without us actually defining the Speak method. That means the following code would be valid.

```
var speaker Speaker = DemoSpeaker{}
```

By embedding interfaces in a struct, you can ensure that the struct implements the entire interface even if it doesn't implement any of the methods on its own. Throughout the rest of this chapter you will see how this affects our code and allows us to easily insert new layers into models package design.

In order to use interface chaining and embedding in our own code, we first need to define an interface. We will define the real interface we use in the next section, but for now let's look at an example using a simple **UserReader** interface.

```
type UserReader interface {
    ByID(id uint) (*User, error)
    ByRemember(token string) (*User, error)
}
```

With that interface defined, we could update our UserService to embed this type.

```
type UserService struct {
    UserReader
}
```

In fact, we could do this for every layer we want to add to our models code.

```
type userValidator struct {
    UserReader
}

type userCache struct {
    UserReader
}

type userGorm struct {
    db *gorm.DB
}
```

```

}

func (ug userGorm) ByID(id uint) (*User, error) {
    var user User
    // pretend to get a user w/ db
    return &user, nil
}

func (ug userGorm) ByRemember(token string) (*User, error) {
    var user User
    // pretend to get a user w/ db
    return &user, nil
}

```

Now our userValidator implements the UserReader interface without writing any extra code because it embeds it.

```

uv := userValidator{}
uv.ByID() // This is the same as uv.UserReader.ByID()
          // The method call is deferred to the embedded type

```

Note: This is NOT the same as inheritance in some languages.

Because we embedded our interfaces, we can now introduce each of these layers without writing all of the code for them. That means we can slowly add validations for a single method, like ByRemember, while the ByID method will continue to work because it will simply be deferred to the embedded type.

And because we used interfaces instead of static types, we have the freedom to pick and choose which layers we want to use in our code without the brittleness we discussed earlier. For example, we could create a user service without a validator and one with.

```

usWithoutVal := UserService{
    UserReader: userGorm{db},
}

usWithVal := UserService{
    UserReader: userValidator{

```

```
UserReader: userGorm{db},  
},  
}
```

In the next section we will get back to writing code, and we will start by creating an interface that defines all of the ways we can interact with the users table in a database. We will then add a few layers to our UserService, setting ourselves up to start organizing our code properly.

If you are a little lost right now, try going through the rest of the chapter before giving up. Much of what we are discussing is hard to understand conceptually until you see it in action, but once you start to see the code it all becomes much clearer.

Box 12.4. Testing is simpler with interfaces

Using embedded interfaces and many layers also makes testing much simpler, and while we won't be writing tests in this course I do think it is important to learn to design code that is testable.

The many layers make testing easier because we can break our tests up. When we test our database layer, we don't need to concern ourselves with invalid data. We simply need to test that this code works when passed valid data.

Similarly, our validation layer only needs to concern itself with validating data, and we don't really need to worry about interacting with a real database. In fact, we could stub out the database later by writing a fake UserReader implementation that pretends to read and write from a database, and then use that when testing our validation layer. The code might look something like the code below.

```
var testUser User
type testUserReader struct {}

func (testUserReader) ByID(id uint) (*User, error) {
    if id <= 10 {
        return &testUser, nil
    }
    return nil, ErrNotFound
}
```

Now when we write tests we would use this UserReader implementation instead of the real one. This is fairly simplistic example, but stubbing out code for tests like this is very helpful when writing maintainable tests in a larger application.

Box 12.5. When should I use interfaces?

Knowing when to use an interface for these purposes isn't always obvious. This can be especially confusing if you are coming from a language where you often define many interfaces upfront.

While I can't give you any specific formula for determining when you should use an interface, what I can tell you is that interfaces in Go are very different from other languages and are fairly easy to add in later. As a result, I typically recommend starting with a struct, and then converting it into an interface later when you see enough evidence that an interface would be easier to use.

12.2 The UserDB interface

In this section we are going to start implementing what we have discussed so far in this chapter. We will start by creating the **UserDB** interface, which is a subset of the UserService that interacts with a database, and then we will be updating our code so that the three layers we discussed are present and ready to be implemented.

We aren't actually going to move our code into the correct layer in this section, but instead will just be laying the groundwork so that we can organize our code and add new validations throughout the rest of this chapter.

We are going to be working in the users model, so let's start by opening up that source code.

```
$ atom models/users.go
```

We are going to start by defining all of the methods we expect our UserDB interface to have. We have already written all of these methods, so we are really just declaring all of the methods we already wrote and adding a bit of documentation for them.

```
// UserDB is used to interact with the users database.
//
// For pretty much all single user queries:
// If the user is found, we will return a nil error
// If the user is not found, we will return ErrNotFound
// If there is another error, we will return an error with
// more information about what went wrong. This may not be
// an error generated by the models package.
//
// For single user queries, any error but ErrNotFound should
// probably result in a 500 error until we make "public"
// facing errors.
type UserDB interface {
    // Methods for querying for single users
    ByID(id uint) (*User, error)
    ByEmail(email string) (*User, error)
```

```
ByRemember(token string) (*User, error)

// Methods for altering users
Create(user *User) error
Update(user *User) error
Delete(id uint) error
}
```

Note: I tend to declare my exported types, especially interfaces, near the top of my source file, but you can put this anywhere in your source file.

The only method we will not be adding to this interface is the Authenticate method, as this does not require direct DB access. It can instead be built using our UserDB layer to look up the user via their email address, and then verify the user using the hashed password and bcrypt. As a result, it doesn't make sense to add this to the UserDB interface.

We are also going to add some utility methods to our UserDB interface - like the ability to close our UserDB connection, or the ability to automatically migrate the table based on the update User type. While not all of these will stay here forever, it is much easier to make this code change if we include these methods.

```
type UserDB interface {
    // ... Keep the existing methods in this interface and add
    //      the ones below.

    // Used to close a DB connection
    Close() error

    // Migration helpers
    AutoMigrate() error
    DestructiveReset() error
}
```

Note: We will eventually remove the AutoMigrate and DestructiveReset methods from the UserDB interface when we perform migrations, but for now it simplifies things to keep them here.

While we likely won't be using the UserDB outside the context of a UserService

and therefore do not need to export it, I prefer to do so here in order to export the documentation. You can see this in action by running the following in your console:

```
$ godoc -http=:6060
```

This will start up a godoc server on port 6060, allowing you to head over to localhost:6060 and view the documentation generated automatically from your source code and comments. You may need to manually navigate to your package¹, but after that you should see the UserDB interface we just defined in the docs.

While it generally isn't a great idea to export types you don't want used outside of your package, in this case we are only exporting an interface which means we aren't exporting an actual implementation. We won't be exporting any of our UserDB implementations because we don't want those used outside of our models package.

In order to write an implementation that is not exported, the type must start with a lower case letter. Our implementation is also going to be using GORM to implement all of the methods, so we are going to name it **userGorm**.

```
// userGorm represents our database interaction layer
// and implements the UserDB interface fully.
type userGorm struct {
    db    *gorm.DB
    hmac hash.HMAC
}
```

We know that we need a **gorm.DB** in order to interact with a database, and I also know that we will temporarily want the **hmac** field, so we are going to add both of those for now. We will eventually change this code a bit once we have our validation code finished.

¹localhost:6060/pkg/lenslocked.com/models/#UserDB

[H]

type UserDB

UserDB is used to interact with the users database.

For pretty much all single user queries: If the user is found, we will return a nil error. If the user is not found, we will return ErrNotFound. If there is another error, we will return an error with more information about what went wrong. This may not be an error generated by the models package.

For single user queries, any error but ErrNotFound should probably result in a 500 error.

```
type UserDB interface {
    // Methods for querying for single users
    ByID(id uint) (*User, error)
    ByEmail(email string) (*User, error)
    ByRemember(token string) (*User, error)

    // Methods for altering users
    Create(user *User) error
    Update(user *User) error
    Delete(id uint) error

    // Used to close a DB connection
    Close() error

    // Migration helpers
    AutoMigrate() error
    DestructiveReset() error
}
```

Figure 12.3: UserDB documentation

Next we are going to update our code so that all of the UserDB methods that were previously implemented as part of the UserService will instead be implemented with the `userGorm` type. To do this, we will change the receiver of each of our existing methods from UserService to userGorm.

Box 12.6. What is a receiver?

A receiver is the parameter to a method that comes *before* the method name. For example, in the following code the receiver is of the type Dog, and is assigned to the variable `d`.

```
type Dog struct {}

func (d Dog) Bark() {
    fmt.Println("woof!")
}
```

We will start with the `ByID` method, which currently starts out like below:

```
func (us *UserService) ByID(id uint) (*User, error) {
```

Right now this has a UserService pointer as its receiver type. We will update this to instead be a pointer to the userGorm type. In the code snippet below I have also renamed the receiver variable to `ug` instead of `us` and updated the code in the method to match this change, but you do not have to make this change in order for your code to work.

```
func (ug *userGorm) ByID(id uint) (*User, error) {
    var user User
    db := ug.db.Where("id = ?", id)
```

```
err := first(db, &user)
if err != nil {
    return nil, err
}
return &user, nil
}
```

We are going to continue performing similar modifications like this to the rest of the methods currently implemented by the UserService type and defined by the UserDB interface. That is, we need to update the following methods:

- **ByEmail**
- **ByRemember**
- **Create**
- **Update**
- **Delete**
- **Close**
- **DesctructiveReset**
- **AutoMigrate**

We do NOT need to update the Authenticate method, so be sure to leave that one unchanged.

I won't be showing code snippets for each of these changes in the book, but the process is the same as what we did for the ByID method. Simply find the receiver and update it to be the userGorm type instead of the UserService type.

If you would like to double check your code, I suggest referencing the source code diff provided at the end of this section.

Once you have completed all of the updates, `userGorm` should now implement the UserDB interface. We can verify this by adding the following to our source code:

```
var _ UserDB = &userGorm{}
```

We name this variable `_` because we won't be using it, and we don't want the compiler to complain about us creating a variable we don't use. We then set this variable's type to UserDB, and then assign a pointer to a userGorm as the value. Assuming our userGorm implements the UserDB interface, this code will compile correctly. If the userGorm does not implement the UserDB interface, we will get an error when we try to compile our code.

Note: Many developers may argue that this is really a test and should be stored in a test file, but I often find it useful to have a simple validation like this in my source code since it will inevitably result in a compilation error somewhere if this ever stops being true.

The next piece of code we are going to write is a function used to instantiate our userGorm instances. This is going to be nearly identical to the NewUserService function we currently have, so you can likely copy the code from there as a starting point. We mostly only need to alter the function name and the return value.

```
func newUserGorm(connectionInfo string) (*userGorm, error) {
    db, err := gorm.Open("postgres", connectionInfo)
    if err != nil {
        return nil, err
    }
    db.LogMode(true)
    hmac := hash.NewHMAC(hmacSecretKey)
    return &userGorm{
        db:   db,
        hmac: hmac,
    }, nil
}
```

We will clean this code up a bit more before deploying, but for now this should do. It is also important to note that we do not export this function either, as we wouldn't want any code outside of our models package instantiating userGorm instances with this function, which IS possible even if the userGorm type isn't exported.

Next we are going to update the UserService type and the NewUserService function to include an embedded **UserDB** field that will be implemented by the **userGorm** type. We can utilize the newUserGorm function we just wrote to make this a little easier on ourselves.

```
func NewUserService(connectionInfo string) (*UserService, error) {
    ug, err := newUserGorm(connectionInfo)
    if err != nil {
        return nil, err
    }
    return &UserService{
        UserDB: ug,
    }, nil
}

type UserService struct {
    UserDB
}
```

Now our UserService still has the same methods as before, but most of the implementation has been offloaded to the embedded UserDB. UserService is only responsible for implementing the Authenticate method.

Our last step in this section is to add the third and final layer, the validation layer. For now this won't actually do much, but we are going to add it so that when we start adding validation code we have a proper place to put it.

The source code for this is going to look very similar to some of the examples we discussed earlier in this chapter but didn't actually code. It will be an unexported type named **userValidator**, and it will start off with a single embedded field of the type UserDB.

```
// userValidator is our validation layer that validates
// and normalizes data before passing it on to the next
// UserDB in our interface chain.
type userValidator struct {
    UserDB
}
```

As we add validation and normalization code to our application we will add it to this type by defining any necessary UserDB methods and then adding the validation code before we perform our interface chaining to the underlying UserDB. For example, we might validate an ID with the code below.

```
func (uv *userValidator) ByID(id uint) (*User, error) {
    // Validate the ID
    if id <= 0 {
        return nil, errors.New("Invalid ID")
    }
    // If it is valid, call the next method in the chain and
    // return its results.
    return uv.UserDB.ByID(id)
}
```

Note: You DO NOT need to write this code, as we will be covering it in a later section. It is simply meant as an example of where we are going.

By doing this we can validate data before the next UserDB ever gets called, meaning we can validate data before our userGorm methods ever start to interact with our database.

The final step we need to take is to update our NewUserService function to use the userValidator layer.

```
func NewUserService(connectionInfo string) (*UserService, error) {
    ug, err := newUserGorm(connectionInfo)
    if err != nil {
        return nil, err
    }
    return &UserService{
```

```
UserDB: userValidator{  
    UserDB: ug,  
},  
, nil  
}
```

With that last change we are now ready to start implementing validations and moving code to its proper layer in our code.

In the next section we will define the UserService interface and perform some similar changes to transform our UserService from a struct into an interface, but after that we will be ready to start filling out each of these layers. We will start that process by first moving validations that exist into the validation layer, and then we will write many new validators and normalizers to transform our data into the format our database expects.

Source code

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

Completed - book.usegolang.com/12.2

Changes - book.usegolang.com/12.2-diff

12.3 The UserService interface

In this section we are going to transform our UserService into an interface. While this step isn't completely necessary for our code, it is a step I like to take because it allows for more flexibility in our code much like our UserDB interface offers. For example, it can make testing much simpler in the long term.

To do this, we are going to first declare the UserService interface. After that, we will rename our existing UserService struct type to `userService`. Notice that this name is nearly identical, except the first letter is lowercase. This means that this type won't be exported, and more importantly it is different from our UserService, so we are allowed to use both as unique type names in our code.

Note: You do not need to name this userService, and could really name it anything you want. We are just doing it to make it clear what interface this type implements.

Open up your users models source code.

```
$ atom models/users.go
```

We will start with the UserService interface declaration. This just defines methods we will be offering, so we will need the Authenticate method as well as all of the methods offered by the UserDB interface.

```
// UserService is a set of methods used to manipulate and
// work with the user model
type UserService interface {
    // Authenticate will verify the provided email address and
    // password are correct. If they are correct, the user
    // corresponding to that email will be returned. Otherwise
    // You will receive either:
    // ErrNotFound, ErrInvalidPassword, or another error if
    // something goes wrong.
    Authenticate(email, password string) (*User, error)
    UserDB
}
```

Next we need to update the existing type so that it is no longer be exported. We also need to add an embedded UserDB so that we implement the UserService interface entirely. Find the line that reads:

```
type UserService struct {
```

And update it to instead start with a lowercase letter so that it is not exported.

```
type userService struct {
    UserDB
}
```

We also need to update the receiver on any methods that the UserService was implementing before. Most notably, we need to update the Authenticate method.

```
// The (us *UserService) at the start becomes
// (us *userService) - notice the lowercase u in the type.
func (us *userService) Authenticate(email, password string) (*User, error) {
    // ... The implementation doesn't change at all.
}
```

Our NewUserService function is also broken now, so we need to update it to return an interface rather than a pointer to a specific type. We also need to change some of the code used to construct the user service we return. This is explained in the comments below.

```
// THIS NO LONGER RETURNS A POINTER! Interfaces can be nil,
// so we don't need to return a pointer here. Don't forget
// to update this first line - we removed the * character
// at the end where we write (UserService, error)
func NewUserService(connectionInfo string) (UserService, error) {
    ug, err := newUserGorm(connectionInfo)
    if err != nil {
        return nil, err
    }
    // We also need to update how we construct the user service.
    // We no longer have a UserService type to construct, and
    // instead need to use the userService type.
    // This IS still a pointer, as our functions implementing
    // the UserService are done with pointer receivers. eg:
    // func (us *userService) <- this uses a pointer
```

```
return &userService{
    UserDB: &userValidator{
        UserDB: ug,
    },
}, nil
}
```

Note: It is very easy to miss the fact that we changed the return type in the function above, but it is an important change to make.

Our next step might be to verify that our **userService** type implements the UserService interface. This can be done similarly to how we did this with the userGorm type.

```
var _ UserService = &userService{}
```

Lastly, we need to make a few tweaks to the users controller. It currently expects a pointer to a UserService, but we no longer need this to be a pointer because it is an interface.

Open up the users controller source file.

```
$ atom controllers/users.go
```

We are going to update the NewUsers function along with the Users type to both no longer expect a pointer to the UserService. That is, we will be removing a few asterisks (*) from our code.

```
func NewUsers(us models.UserService) *Users {
    return &Users{
        NewView:   views.NewView("bootstrap", "users/new"),
        LoginView: views.NewView("bootstrap", "users/login"),
        us:         us,
    }
}
```

```
}
```

```
type Users struct {
    NewView    *views.View
    LoginView  *views.View
    us         models.UserService
}
```

With that we should be done updating the UserService and our code should compile again. Go ahead and attempt to run your code to verify it works.

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

Source code

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

Completed - book.usegolang.com/12.3

Changes - book.usegolang.com/12.3-diff

12.4 Writing and organizing validation code

Now that we have our code separated into layers, we are ready to start moving our validation code into methods on the userValidator type. This will allow us to separate that code from the database interactions.

We will be doing both normalizations and validations inside the userValidator, so we will often call this layer the “validation” layer but it really include both of these tasks. As stated earlier in this chapter, this is done because the two are so tightly linked.

In the next few subsections we will start by moving our code to the userValidator in the most straightforward manner. After that we will look at how we can start to create reusable validation functions, which will enable us to spend the rest of this chapter simply writing the validation and normalization functions we will need for our users.

12.4.1 The straightforward approach

We are going to start by looking at the most straightforward approach of writing and organizing validation code. That is, we are going to simply move the code into the appropriate method on the userValidator type.

We will be working in the users model, so first let's open up our source file.

```
$ atom models/users.go
```

We are going to look over our source code for any code in userGorm methods that looks like normalization or validation code. We are then going to move that code to the same method on the userValidator type.

ByID and ByEmail don't have any validations yet, so we can safely skip those. ByRemember has code used to hash a remember token, which I consider to be a normalization. We are transforming the remember token into the final form we expect to persist in our database.

We are going to move this code out of userGorm's ByRemember method and into a newly created ByRemember method associated to the userValidator type. We will start by writing the new code, and then we will come back and remove the old code.

```
// ByRemember will hash the remember token and then call
// ByRemember on the subsequent UserDB layer.
func (uv *userValidator) ByRemember(token string) (*User, error) {
```

```

rememberHash := uv.hmac.Hash(token)
return uv.UserDB.ByRemember(rememberHash)
}

```

We have created a ByRemember method on the userValidator type and moved our hashing code to it, but we have an issue. Our userValidator doesn't currently have an **hmac** field to hash the token.

To fix this, we need to add the hmac field to our userValidator type.

```

type userValidator struct {
    UserDB
    hmac hash.HMAC
}

```

We need to make sure we instantiate this field as well. For now we will handle this inside of the NewUserService function.

```

func NewUserService(connectionInfo string) (UserService, error) {
    ug, err := newUserGorm(connectionInfo)
    if err != nil {
        return nil, err
    }
    // this old line was in newUserGorm
    hmac := hash.NewHMAC(hmacSecretKey)
    uv := &userValidator{
        hmac:   hmac,
        UserDB: ug,
    }
    return &userService{
        UserDB: uv,
    }, nil
}

```

Next we need to remove the normalization code from userGorm's ByRemember method. We will also want to update any comments and arguments we have to reflect what our code is actually doing.

```
// ByRemember looks up a user with the given remember token
// and returns that user. This method expects the remember
// token to already be hashed.
func (ug *userGorm) ByRemember(rememberHash string) (*User, error) {
    var user User
    err := first(ug.db.Where("remember_hash = ?", rememberHash), &user)
    if err != nil {
        return nil, err
    }
    return &user, nil
}
```

With those changes completed, our code should now be compiling again. Take a moment to verify that your code builds.

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

While our code is functionally the same, we should now have some method chaining occurring when we call the `ByRemember` function. When we first call it on the `userService` type the method call will be deferred to the `userValidator`'s `ByRemember` method. At that point our code will normalize the remember token by hashing it, and then it will call the next method in the chain - the `userGorm`'s `ByRemember` method.

As our validations become more complex, this separation will start to make it much easier to manage our code.

Source code

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

Completed - book.usegolang.com/12.4.1

Changes - book.usegolang.com/12.4.1-diff

12.4.2 A few more validation examples

In this section we are going to update the Create, Update, and Delete methods in order to show a few more examples of how we can separate our validation code from our database interactions. Once we have done that we will also be able to clean up the userGorm type a bit, as it will no longer need the `hmac` field it currently has.

We will start with userGorm's Create method, which currently has quite a bit of validation and normalization code in it. To update this method we are going to code a new Create method on the userValidator type that will instead contain all of this logic, and then remove the logic from the existing method.

```
// Create will create the provided user and backfill data
// like the ID, CreatedAt, and UpdatedAt fields.
func (uv *userValidator) Create(user *User) error {
    pwBytes := []byte(user.Password + userPwPepper)
    hashedBytes, err := bcrypt.GenerateFromPassword(pwBytes,
        bcrypt.DefaultCost)
    if err != nil {
        return err
    }
    user.PasswordHash = string(hashedBytes)
    user.Password = ""

    if user.Remember == "" {
        token, err := rand.RememberToken()
        if err != nil {
            return err
        }
        user.Remember = token
    }
    user.RememberHash = uv.hmac.Hash(user.Remember)
    return uv.UserDB.Create(user)
}

// And now the userGorm version becomes...
func (ug *userGorm) Create(user *User) error {
    return ug.db.Create(user).Error
}
```

Nearly all of the work we did here was moving code around, but that is okay. It is helping us get to a point where our code is properly split up.

Box 12.7. userGorm's Create method is simple now

It is important to note how simple this is making our userGorm methods as we make these changes, and as we continue it will likely get even simpler.

In addition to making it easier to manage and test our code, this also means that we could easily swap out our database implementation with another fairly easily. For example, if we decided we wanted to use the database/sql package that comes with the standard library instead of GORM, we would only need to replace the relatively simple userGorm type with a new type, and all of our remaining code - validation, normalization, etc - would remain unchanged.

This is why I say that you could easily swap out all of the GORM code used in this course with a standard library; doing so only involves changing a very minor portion of our code once we get our models package cleaned up.

Next up is the Update method. This is going to be very similar; we are going to create a new Update method with a userValidator receiver, then move our validation code over to it.

```
// Update will hash a remember token if it is provided.
func (uv *userValidator) Update(user *User) error {
    if user.Remember != "" {
        user.RememberHash = uv.hmac.Hash(user.Remember)
    }
    return uv.UserDB.Update(user)
}

func (ug *userGorm) Update(user *User) error {
    return ug.db.Save(user).Error
}
```

And our final example is the Delete method, with the same workflow as before.

```
// Delete will delete the user with the provided ID
func (uv *userValidator) Delete(id uint) error {
    if id == 0 {
        return ErrInvalidID
    }
    return uv.UserDB.Delete(id)
}

func (ug *userGorm) Delete(id uint) error {
    user := User{Model: gorm.Model{ID: id}}
    return ug.db.Delete(&user).Error
}
```

Now that we have moved all of the references to the `hmac` field from our user-Gorm code, we no longer need this field. We are going to update our code to remove this since it is no longer being used. While this isn't absolutely necessary to make our code compile, it is a good idea to do housekeeping like this as you develop.

First we need to remove the field from the `userGorm` type declaration.

```
type userGorm struct {
    db *gorm.DB
}
```

Next, we need to update the `newUserGorm` function and remove any code used to setup the `hmac` field.

```
func newUserGorm(connectionInfo string) (*userGorm, error) {
    db, err := gorm.Open("postgres", connectionInfo)
    if err != nil {
        return nil, err
    }
    db.LogMode(true)
    return &userGorm{
        db: db,
    }, nil
}
```

And with that we are done setting up our straightforward validations. We are now ready to look at how to create reusable validators, but first let's make sure our code is compiling correctly and working as expected. Restart your server and test out a few pages to make sure things work as expected. Try logging in again just to be safe.

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

Source code

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

Completed - book.usegolang.com/12.4.2

Changes - book.usegolang.com/12.4.2-diff

12.4.3 Reusable validations

We could continue to create validations like we did in the last two sections, but over time this is going to result in a lot of duplicate code and a lot of checking for errors. Check out the Create method for an example of this; if we want users to be able to update their password most of the code in the Create method will need to be copied over so that we can properly hash their new password.

We don't want to repeat ourselves over and over again, so a better way to do this would be to break these validations and normalizations into reusable functions. While we are at it, it would also be helpful to define a single format for all of our validations functions so that we could run them all at once, stopping as soon as any of the validations returns an error as an error means we can no longer proceed with the operation.

Let's start out by creating a reusable method that allows us to hash a user's password using bcrypt. This code will mostly be the same as code that already exists in the userValidator's Create method, but moved into its own method.

```
// bcryptPassword will hash a user's password with an
// app-wide pepper and bcrypt, which salts for us.
func (uv *userValidator) bcryptPassword(user *User) error {
    pwBytes := []byte(user.Password + userPwPepper)
    hashedBytes, err := bcrypt.GenerateFromPassword(pwBytes,
        bcrypt.DefaultCost)
    if err != nil {
        return err
    }
    user.PasswordHash = string(hashedBytes)
    user.Password = ""
    return nil
}
```

Rather than simply taking in a user's password and returning a hashed password, we are accepting a pointer to a user and returning only an error. That means any data changes need to occur directly on the provided user object, which works because we were provided with a pointer to a user.

What is especially nice about this format is that it is easy to standardize. We can now define a function type named **userValFn** that defines what we expect a user validation function to look like.

```
type userValFn func(*User) error
```

Function types are similar to struct and interface types, except they define a function format. What we are saying with the code we just wrote is that a userValFn is any function that accepts a pointer to a User and returns an error.

That means that all of our validation functions are going to need to accept a User pointer, do whatever work is necessary, and then return either nil if no errors were encountered, or an error if something went wrong during the validation.

What's the point? Why do we want a single format for our validation functions?

The upside to using this standardized approach is that we can now define a function that runs all of our validations sequentially, stopping and returning any errors encountered along the way. Once we have written this, it will make it much easier to start adding new validators to our code without needing to check each one individually for errors.

```
func runUserValFns(user *User, fns ...userValFn) error {
    for _, fn := range fns {
        if err := fn(user); err != nil {
            return err
        }
    }
    return nil
}
```

The `runUserValFns` function accepts a pointer to a user and any number of validation functions as its arguments, and then it iterates over each validation function using a `range`. As it iterates over each function, it calls that validation function passing in the user as the argument, and capturing the error return value.

If the return value turns out to be nil, it means we didn't receive an error and we can continue on to the next validation function which is what our code does. If the return value isn't nil, it means we received an error and we can stop running our validations and return that error. Finally, if all validations have run successfully we can return nil, signifying that we didn't encounter any errors running the validations.

Because the user is a pointer, any changes made to the data inside of it will be persisted between each function call allowing us to also capture any data normalization that happens along the way.

To use this function, we would need to write some code like below.

```
if err := runUserValFns(user,
    uv.bcryptPassword); err != nil {
    return err
}
```

Note: Do not code this - we will add code like this shortly.

While this doesn't seem that much simpler at first, it becomes much easier when we have many validations and normalizations to run. Rather than checking each individual validation function for an error, we could instead write something like below.

```
if err := runUserValFns(user,
    uv.validationOne,
    uv.validationTwo,
    uv.normalizationOne,
    uv.normalizationTwo,
    uv.validationThree,
    uv.validationFour); err != nil {
    return err
}
```

Note: Do not code this - we will add code like this shortly.

As our validations grow, we simply add them to a list that gets run in the order we list it, making it nice and easy to manage.

Now let's set this up inside of our Create and Update methods using the bcrypt-Password method we wrote earlier. We will start with Create, which is the simpler use case.

```
func (uv *userValidator) Create(user *User) error {
    if err := runUserValFns(user,
        uv.bcryptPassword); err != nil {
        return err
    }

    if user.Remember == "" {
```

```
    token, err := rand.RememberToken()
    if err != nil {
        return err
    }
    user.Remember = token
}
user.RememberHash = uv.hmac.Hash(user.Remember)
return uv.UserDB.Create(user)
}
```

And then the Update method...

```
func (uv *userValidator) Update(user *User) error {
    if err := runUserValFns(user,
        uv.bcryptPassword); err != nil {
        return err
    }

    if user.Remember != "" {
        user.RememberHash = uv.hmac.Hash(user.Remember)
    }
    return uv.UserDB.Update(user)
}
```

Code changes to the Update method aren't actually any harder, but they do present us with an issue; our bcryptPassword method ALWAYS hashes the password, but when we update a user there is a chance they won't be updating their password. When this happens, we need a way to tell the bcryptPassword method to skip over its normalization code.

We could do this inside of our Update method, but that would bring us right back to where we started. Adding new normalization and validation code wouldn't be as easy as we want it to be.

Rather than changing the Update method, we are going to instead update the bcryptPassword method to only run if the password is set. Checking this is easy to do - we simply need to see if the user's password is equal to the empty string. If it is, then the password wasn't updated. Otherwise it was and we can hash it.

```
func (uv *userValidator) bcryptPassword(user *User) error {
    if user.Password == "" {
        // We DO NOT need to run this if the password
        // hasn't been changed.
        return nil
    }

    // ... The rest remains the same
}
```

Now at this point you might be wondering, “Won’t this cause us issues in the Create method?” It is a valid question, but later we will see that when we need to ensure things like a password being set on methods like Create, it is often easier to do this in a separate validation rather than trying to jam too many validations into a single function. By splitting these up, we make each of our individual validations much easier to reuse across our application.

With the changes we just made we have laid the groundwork for quickly and easily writing validators and normalizers for our users model. No extra frameworks or third party libraries were needed at all. Instead, we were able to just define our own types and write a few utility functions to simplify the process.

We will spend the next section in this chapter moving all of our existing validation code into functions like the bcryptPassword function we just wrote, after which we will start to introduce new validations and normalizations. But first, let’s verify that our application is still working.

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

Source code

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

Completed - book.usegolang.com/12.4.3

Changes - book.usegolang.com/12.4.3-diff

12.5 Writing validators and normalizers

We've seen how we can move validation and normalization code into our user-Validator layer, and we've also seen how if we break these into individual methods they become much easier to reuse.

This section will be dedicated to taking any remaining validation code we have that isn't broken into a validation function and moving it into one. This will make it easier for us to reuse the validations, as well as giving us some practice with using our new validation functions.

12.5.1 Remember token normalizer

We are still working in the users model, so open up that source file if it isn't already.

```
$ atom models/users.go
```

The first piece we are going to move into a validation function is currently being used in three different methdos - **ByRemember**, **Create**, and **Update**.

In all three of these methods we check to see if a remember token is present, and if so we hash it using the HMAC algorithm. If we move this code to a single method on the userValidator we will be able to reuse it rather than maintaining the same code in all three locations.

We will start by writing the validation function.

```
func (uv *userValidator) hmacRemember(user *User) error {
    if user.Remember == "" {
        return nil
    }
    user.RememberHash = uv.hmac.Hash(user.Remember)
    return nil
}
```

Like all of our other validation functions, this will accept a pointer to a user as its only argument and return an error if the validation fails. Inside the function we check to see if the remember token was set before proceeding. If it was not set, we terminate early returning nil. Otherwise we hash the remember token that is present.

If a remember token is already hashed our code will still continue to work as expected so long as we don't put an invalid value in the `Remember` field on the user. That is why we have separated the Remember and RememberHash fields on our user type; if we instead used a single field it would be very hard to determine if the field was previously hashed or not.

Next we must update the methods that need this normalization - ByRemember, Create, and Update. We will start with ByRemember, which is the trickiest scenario. It is tricky because this method doesn't have a User defined, so we need to create a user variable and set the Remember field to the token we want normalized. Our normalizer will then set the RememberHash field on the user we created, allowing us to pass that into the next layer as the hashed token.

```
func (uv *userValidator) ByRemember(token string) (*User, error) {
    user := User{
        Remember: token,
    }
    if err := runUserValFns(&user, uv.hmacRemember); err != nil {
        return nil, err
    }
    return uv.UserDB.ByRemember(user.RememberHash)
}
```

Note: Notice that we create a user, set the token to the Remember field, and then

use the `RememberHash` that gets set by the `hmacRemember` normalizer. This is how we get around the fact that our `ByRemember` method doesn't have a `User` object passed in even though our validation functions require one.

The `Create` method also requires a few extra tweaks. Most notably, we need to make sure our code that generates a default remember token gets run before we try to hash the token. Later when we port that code to a validation function it will simplify this even further, but for now it just means moving the code to the top of our method before we call the `runUserValFns` function.

```
func (uv *userValidator) Create(user *User) error {
    if user.Remember == "" {
        token, err := rand.RememberToken()
        if err != nil {
        }
        user.Remember = token
    }

    err := runUserValFns(user,
        uv.bcryptPassword,
        uv.hmacRemember)
    if err != nil {
        return err
    }
    return uv.UserDB.Create(user)
}
```

Finally we have the `Update` method.

```
func (uv *userValidator) Update(user *User) error {
    err := runUserValFns(user,
        uv.bcryptPassword,
        uv.hmacRemember)
    if err != nil {
        return err
    }
    return uv.UserDB.Update(user)
}
```

Source code

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

Completed - book.usegolang.com/12.5.1

Changes - book.usegolang.com/12.5.1-diff

12.5.2 Ensuring remember tokens are set on create

The next normalization we need to create will be used to verify that remember tokens are given a default value if none exists. This is necessary because we can't save users in our database without a remember token.

Our normalizer will only need to be run when we are creating new users, but we will create it in the same way we have been creating all our validation functions for consistency.

```
func (uv *userValidator) setRememberIfUnset(user *User) error {
    if user.Remember != "" {
        return nil
    }
    token, err := rand.RememberToken()
    if err != nil {
        return err
    }
    user.Remember = token
    return nil
}
```

In this code we first check to see if a remember token is set. If it is, it means we don't need to set a new value so we return nil.

While it might be tempting to validate the length or size of the token and set a new one if it is too short, I would recommend avoiding this because it could

lead to other odd bugs. It also unnecessarily complicates this normalization function.

Assuming a remember token hasn't been set, our code continues on and generates a remember token using the `rand` package we created. If it runs into any errors, those are returned, otherwise nil will be returned after the new remember token is set on the user.

With the `setRememberIfUnset` function we can now update our `Create` method on the `userValidator` to run this validation. Be sure to do this *BEFORE* the `hmacRemember` function, otherwise your code will have a bug. We need to set the default value before we can hash it.

```
func (uv *userValidator) Create(user *User) error {
    err := runUserValFns(user,
        uv.bcryptPassword,
        uv.setRememberIfUnset,
        uv.hmacRemember)
    if err != nil {
        return err
    }
    return uv.UserDB.Create(user)
}
```

With that we should have a completely cleaned up `Create` method; that is, all of our validations have been placed in their own functions and are being called from the `Create` method. The end result is a much cleaner and easier to read piece of code; we can easily see which validations get run without having to worry ourselves with the implementation details of each validation.

Source code

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

Completed - book.usegolang.com/12.5.2

Changes - book.usegolang.com/12.5.2-diff

12.5.3 Ensuring a valid ID on delete

When we call the Delete method we need to make sure any IDs provided are greater than 0. If we accidentally pass in an ID with the value 0, it will result in the entire database table being deleted which likely isn't what we want.

To implement this validation function we are going to create what is known as a closure. While a closure isn't absolutely necessary for this, it is a good opportunity to demonstrate how you can create more dynamic validation functions that might be used with different parameters depending on the situation.

Box 12.8. What is a closure?

Putting it briefly, a closure is function that is created dynamically and makes use of variables defined outside the scope of the function.

If you would like to read more about closures and anonymous functions, I recommend the following article:

<https://www.calhoun.io/what-is-a-closure/>

We will go ahead and jump right into the code, then explain what is happening after seeing the code.

```
func (uv *userValidator) idGreaterThanOrEqual(n uint) userValFn {
    return userValFn(func(user *User) error {
        if user.ID <= n {
            return ErrInvalidID
        }
        return nil
    })
}
```

```
    })  
}
```

In the code above we first start off by defining a function that accepts an unsigned integer (**uint**), and then returns a **userValFn**. Notice that this function doesn't look like our other validation functions, but instead is used to create one.

That means that inside of our code we need to return a function that matches the **userValFn** definition; that is, it must accept a pointer to a user and return an error.

One way to create a function like this would be to do it dynamically and assign it to a variable, just like you would with any other data. That is, we could create a user validation function this way:

```
fn := func(user *User) error {  
    // do stuff  
    return nil  
}
```

The **fn** variable would then store a function that we could call much like we would any other function.

```
var user User  
err := fn(&user)
```

In our **idGreaterThanOrEqual** function we are doing rough the same thing, but rather storing the function in a variable we precede it with the **return** keyword, signifying that we simply want to return that data.

We also wrap our function with **userValFn(...)** to imply that we want to convert this function into the **userValFn** type. This is similar to converting a string into a byte slice, which you have probably seen happen before.

```
str := "Michael Scott"
b := []bytes(str)
```

But this brings up the bigger question of, “Why did we just go through all that effort? What was the point?”

By doing all of this we have the ability to create a function that only accepts a user and returns an error, but inside of that function we are able to access the unsigned integer, `n`, that we provided to our `idGreaterThan` function. That means we can dynamically create a validation function that verifies IDs are greater than any value, not just 0.

For example if we wanted to create a validation that says IDs must be greater than 10, we could do so with the following code.

```
valFn := uv.idGreaterThanOr(10)
var user User
// This runs just like all our other validation functions,
// but it gets the minimum ID value dynamically
err := valFn(&user)
```

While it is unlikely that we will want to validation specific IDs like this in our code, this validation is meant to provide an example for when you do run into use cases that require more dynamic validations.

Note: Of all the code snippets above, you only need to create the `idGreaterThanOr` function. The rest are for illustration purposes.

We now want to use this validation inside of our `UserValidator`'s `Delete` method. Much like the `ByRemember` method, we will need to instantiate a `User` before we can run our validation functions. We also need to specifically call the `idGreaterThanOr` function because it itself isn't a validation function; as we discussed earlier in this section it instead returns a validation function when you call it with an unsigned integer value.

```
func (uv *userValidator) Delete(id uint) error {
    var user User
    user.ID = id
    err := runUserValFns(&user, uv.idGreaterThan(0))
    if err != nil {
        return err
    }
    return uv.UserDB.Delete(id)
}
```

Yes, our Delete method actually got longer doing our validations this way, but later if we start to add more validations to the Delete method this will end up saving us time, and there are real maintenance benefits to having your code in a uniform manner.

Source code

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

Completed - book.usegolang.com/12.5.3

Changes - book.usegolang.com/12.5.3-diff

12.6 Validating and normalizing email addresses

In this section we are going to focus exclusively on adding validation and normalization methods used to clean and verify a user's email address. That might sound pretty simple at first, but the email address will probably have more validators than any other field stored in our User type.

Specifically, we are going to need to:

- Convert our email addresses to lowercase so we can search for duplicates easily
- Trim any leading or trailing space from our email addresses (" `jon@calhoun.io`" becomes "`jon@calhoun.io`")
- Validate that an email address is present
- Verify that the email address looks reasonably like a real email address
- Check to see if a user exists with the email address we are using, and if so verify that we are updating that same user account before interacting with the database

Let's start by looking at how to normalize the email address so that the rest of our validations are easier to write.

12.6.1 Converting emails to lowercase and trimming whitespace

Whenever we interact with email addresses we need to ensure that they are always in the same format. Searching for `JON@abc.com` in our database when the email address is stored as `jon@abc.com` isn't likely to be very helpful because Postgres is case sensitive when it comes to searching.

Note: Some SQL variants ARE NOT case sensitive, but it is still a good idea to normalize your data.

The best way to handle this is to always normalize our email address by converting it to its lowercase equivalent before storing it in our database and also before performing any queries. While we are at it, it is also a good idea to trim any leading or trailing whitespace. Chances are if a user signs up with " `jon@calhoun.io`" as their email address the extra spaces at the front weren't intentional.

Up until now we have performed a single validation or normalization in each method we create, but in this case we are going to do both of these in a single function. The reasoning for this is that there will never be a situation where we want to perform one of the two validations, but not the other.

Jumping into our code, the first thing we need to do is import the **strings** package. We are going to be importing this so that we can use two functions from it:

- **ToLower**² - This accepts a string as its argument and returns the lower-case equivalent of a string as its return value
- **TrimSpace**³ - This accepts a string as its argument and returns the same string with all leading and trailing whitespace (tabs, spaces, etc) removed

*The **strings** package provides several other helpful methods used to manipulate strings, so if you ever find yourself looking to alter a string this is a good place to start.*

Importing the package is as simple as adding it inside our import block.

```
import (
    "strings"
    // ... the rest of these stay the same
)
```

Next we need to add our normalization function that uses the strings package to manipulate a user's email address. We can't really run into errors here, so the code ends up being pretty short.

²<https://golang.org/pkg/strings/#ToLower>

³<https://golang.org/pkg/strings/#TrimSpace>

```
func (uv *userValidator) normalizeEmail(user *User) error {
    user.Email = strings.ToLower(user.Email)
    user.Email = strings.TrimSpace(user.Email)
    return nil
}
```

Our last step is to find any functions that need this validation and add it. The easiest way to do this is to walk over all the functions in our UserDB and ask ourselves if each one needs this validation. Looking over those methods, it looks like the only three that will need this are ByEmail, Create, and Update.

ByEmail needs this normalization so that we don't accidentally search for email addresses that haven't been normalized. While we know that all of the email addresses stored in our database is going to be lowercase, there is no guarantee that users will use lowercase when logging in to their account.

Create and Update both need this normalization because each of these could be used to change a user's email address, and we need to verify it is normalized before we store it in our database.

Let's start with the ByEmail method. The userValidator doesn't have this method yet, so we need to add it. We also need to create a User object and set the email address so our normalizer has a user to operate on.

```
// ByEmail will normalize an email address before passing
// it on to the database layer to perform the query.
func (uv *userValidator) ByEmail(email string) (*User, error) {
    user := User{
        Email: email,
    }
    err := runUserValFns(&user, uv.normalizeEmail)
    if err != nil {
        return nil, err
    }
    return uv.UserDB.ByEmail(user.Email)
}
```

Next up are the Create and Update methods. These are both as simple as adding the function to our runUserValFns call, so I'll just show the code for both below.

```
func (uv *userValidator) Create(user *User) error {
    err := runUserValFns(user,
        uv.bcryptPassword,
        uv.setRememberIfUnset,
        uv.hmacRemember,
        uv.normalizeEmail)
    // ... this is unchanged
}
```

```
func (uv *userValidator) Update(user *User) error {
    err := runUserValFns(user,
        uv.bcryptPassword,
        uv.hmacRemember,
        uv.normalizeEmail)
    // ... this is unchanged
}
```

If you happened to look at the UserService, you might wonder why we don't need this in the Authenticate method as well. The reason we don't need it there is because that method will ultimately use the ByEmail method to query for a user, so as long as we handle normalization there we don't need to do anything special inside the Authenticate method.

Source code

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

Completed - book.usegolang.com/12.6.1

Changes - book.usegolang.com/12.6.1-diff

12.6.2 Requiring email addresses

The next validation we are going to write is a simpler one - we need to ensure that an email address is present every time we create or update a user.

We could instead just check this when we verify that the email address looks like a real email address, but I find it helpful to distinguish between the two errors. Often times a missing email address can be indicative of other bugs in your application, and it can be useful to inform end users that the email address field is indeed required.

To start we will define an error to return whenever an email address isn't provided. We will be exporting this error so that other code can check for this error specifically, and later in the course we will look at how to turn this error into a useful error message that we can show end users.

```
var (
    // ... There are other errors in our source file. Just add
    // the ErrEmailRequired error below them.

    // ErrEmailRequired is returned when an email address is
    // not provided when creating a user
    ErrEmailRequired = errors.New("models: email address is required")
)
```

After that we write the validation function. This one is pretty simple - check to see if the email address is the empty string and if it is return an error stating that an email address is required. This will run after our normalization code we just wrote, so extra whitespace will be removed turning something like " " (whitespace) into an empty string.

```
func (uv *userValidator) requireEmail(user *User) error {
    if user.Email == "" {
        return ErrEmailRequired
    }
    return nil
}
```

Finally we need to add this to our Create and Update methods on the userValidator. Again, this process only involves adding a single line to each method so I'll just show both code snippets together.

```
func (uv *userValidator) Create(user *User) error {
    err := runUserValFns(user,
        uv.bcryptPassword,
        uv.setRememberIfUnset,
        uv.hmacRemember,
        uv.normalizeEmail,
        uv.requireEmail)
    // ... this is unchanged
}
```

```
func (uv *userValidator) Update(user *User) error {
    err := runUserValFns(user,
        uv.bcryptPassword,
        uv.hmacRemember,
        uv.normalizeEmail,
        uv.requireEmail)
    // ... this is unchanged
}
```

With that we have successfully added a validation that requires email addresses.

Source code

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

Completed - book.usegolang.com/12.6.2

Changes - book.usegolang.com/12.6.2-diff

12.6.3 Verifying emails match a pattern

Our next validator is likely going to be the the most complicated one we write in this section. It isn't complicated because the code is lengthy, but rather because we are going to be introducing regular expressions.

We want to make sure any email addresses we save in our database roughly match a pattern that looks like a real email address. To do this we can make a few relatively simply rules. Things like, “An email address needs to have an @ sign in it.” The easiest way to do this is in code is often with a regular expression (often shortened to regexp or regex).

Box 12.9. What are regular expressions?

Regular expressions are special strings used to describe a pattern that we want to search for. For example, in this section they will give us a way to describe a basic pattern that we expect email addresses to match and verify that all email addresses meet that expectation.

Regular expressions are NOT the same as glob patterns. They are typically more complicated, but are also more specific allowing developers to define exactly what they want to match. A common way of describing them is, “wildcards on steroids”.

In this section I will briefly describe what our regular expression is searching for, but if you decide you want to read more about them I suggest checking out the following website:

<http://www.regular-expressions.info/>

We are going to define a somewhat simple regular expression that we will use to check our email addresses. This regular expression won't match perfectly against all valid email addresses, but it should work for most real email ad-

dresses.

We won't be using a regex that is 100% accurate because this is very hard to do, and often entails using a regular expression that is very lengthy and hard to understand. Instead, we will be using a regex that is simple enough that you can start to learn how it works and expand upon it yourself.

Before we get into the actual regular express pattern, let's first look at how you compile a regular expression in Go. Doing this involves using the `regexp` package⁴ to create a `Regexp`⁵.

To do this, we will want to use one of two methods - `Compile`⁶ or `MustCompile`⁷. The former of these two methods will attempt to create a `Regexp` using a pattern and will return either that `Regexp`, or an error if there is one with your pattern. The latter, `MustCompile`, is a helper method that will call `Compile` and panic if there is an error. Otherwise it returns the `Regexp` returned by `Compile`.

`MustCompile` is often used when a program is expected to terminate if the regular expression doesn't compile, which will be the case in our program. There isn't much point in starting our application up if our regular express isn't valid - we need to take a minute to fix it first.

In code using the `MustCompile` function looks roughly like the following code:

```
r := regexp.MustCompile(`pattern here...`)
```

Box 12.10. What are those backticks?

When defining strings in Go, you will most often see people using double quotes

⁴<https://golang.org/pkg/regexp>

⁵<https://golang.org/pkg/regexp/#Regexp>

⁶<https://golang.org/pkg/regexp/#Compile>

⁷<https://golang.org/pkg/regexp/#MustCompile>

(`"`). In addition to double quotes, you can also create strings using the backtick character (```).

The primary advantages to using the backtick is that you don't need to escape special characters, and your strings can have newlines in them that will be present in the string when you print it out without manually adding them.

They are often used when creating regular expressions because they use several special characters, and having to escape them can make the regular expression harder to read for developers.

The pattern we will be using when constructing our regular expression is shown below. Take a look at it, then we will take a few moments to explain broadly what it is doing.

```
^[a-zA-Z0-9._%+\-]+@[a-zA-Z0-9.\-]+\.[a-zA-Z]{2,16}$
```

This is easiest to parse in pieces, so let's start with the first portion.

```
^[a-zA-Z0-9._%+\-]+
```

This portion of the pattern can be broken into two parts. The first is the `^` character, which states that this pattern must match the start of a string and can't match somewhere in the middle or end of a string.

After the `^` character we have the `[]` brackets filled with some characters, and then followed by a plus sign (`+`). The letters in the brackets are a character set, which is a fancy way of saying we are going to define several different characters and our string can match any of them.

For example, if we were to use the regex pattern **[a-z]** we are saying this can match any single character between **a** and **z**, so any of the 26 lowercase letters in the English alphabet would match.

Our character set includes alphabet letters, digits (**0-9**), and a few special symbols (**.**, **_**, **%**, **+**, and **-**). The last special symbol still needs to be escaped (the **** before **-** is escaping it) because the dash is a special symbol inside of character sets that allows us to do ranges like **a-z**, but by escaping it we are telling our regular expression parser that we want to match the actual **-** character.

Whenever we create character sets, by default they will only match once. By adding the **+** sign after our character set we are stating that we can match any of those characters one or more times. That is, we are stating that it can repeat, but has to match at least once. That means we could match any of the following strings: “**jon**”, “**cat123**”, “**d+_o.g**”

In short, this part of the regular expression is used to match all the letters that go before the at sign. If you ever needed to permit a new character in that part of an email address, you would update this portion of the regular expression.

Next is the **@** part of the original regular expression.

```
^[a-zA-Z0-9._%+\-]+@[a-zA-Z0-9.\-]+\.[a-zA-Z]{2,16}$  
      ^  
This part
```

Because this isn’t wrapped in anything special, this simply states that we need to match against the **@** character exactly.

Everything we have so far states that our pattern must start at the beginning of a string, match 1 or more of the characters in our character set, and then be followed by a single **@** character. So we would be matching things like “**jon@**”.

After that we introduce another character set followed by a plus sign.

```
[a-zA-Z0-9.\-]+
```

This character set is similar to the one we defined earlier, but is simpler. It only accepts letters, digits, dashes, and periods, and the plus at the end signifies that we need to match this character set one or more times.

Note: We permit periods here so that subdomains work in email addresses.

After that we have `\.`, which is an escaped period. When outside of a character set (`[]`), the period has a special meaning and will match against *any* character, so if we want to match the period exactly we need to escape it with the backslash (`\`) character.

This line is like our `@` sign we saw earlier - it states that we need to match against exactly one period.

Then finally we end our regular expression with the following:

```
[a-zA-Z]{2,16}$
```

This states that we need a lowercase letter between `a` and `z`, and that we need between two and sixteen matches. After that the `$` implies that this must match the end of the string, preventing us from matching on a string prefix.

We can be more specific in this part of our regular expression because every valid top-level domain (TLD)⁸ is between two and sixteen characters long, and is composed of only alpha characters. While you might think that TLDs like `.co.uk` will fail this check, we don't need to worry about the `.co.` portion because it will be matched earlier in our regular expression.

If this is all a little fuzzy still, don't worry. Regular expressions take a little while to get used to. What is important now is that you recognize that this will be used

⁸https://en.wikipedia.org/wiki/Top-level_domain

to match a pattern. Understanding how the regex was built isn't as important right now.

Again, this regular expressions is not perfect, but it does cover our basic needs for now, and prepares us to start writing our validation function.

We are going to start by importing the regexp package.

```
import (
    "regexp"
    // ... Just add the import above
)
```

We are going to need to access our regular expression from userValidator methods, so rather than creating a global variable we will use a field on that type.

```
type userValidator struct {
    UserDB
    hmac      hash.HMAC
    emailRegex *regexp.Regexp
}
```

Before we can use the regular expression we need to compile it and assign it to the field. We currently construct our userValidator inside the NewUserService function, but as our complexity starts to increase I like to break this into its own function. We can do that by creating a newUserValidator function, then calling it from the NewUserService function much like we do with the userGorm.

```
func newUserValidator(ldb UserDB,
    hmac hash.HMAC) *userValidator {
    return &userValidator{
        UserDB: ldb,
        hmac:   hmac,
        emailRegex: regexp.MustCompile(
            `^[\w\.-]+@[a-zA-Z\.\-]+\.[a-zA-Z]{2,16}$`),
    }
}
```

```
func NewUserService(connectionInfo string) (UserService, error) {
    ug, err := newUserGorm(connectionInfo)
    if err != nil {
        return nil, err
    }
    hmac := hash.NewHMAC(hmacSecretKey)
    uv := newUserValidator(ug, hmac)
    return &userService{
        UserDB: uv,
    }, nil
}
```

With all that prep work done, we are ready to start writing our validation function. We will start by defining an error we return when an email address doesn't appear to be valid. We can add this to the other errors we have in our variable block.

```
var (
    // ErrEmailInvalid is returned when an email address provided
    // does not match any of our requirements
    ErrEmailInvalid = errors.New("models: email address is not valid")
)
```

After creating the error we can start writing the validation function. Since the regular expression is already created, all we need to do is use the `MatchString`⁹ method which will return true if a string matches our pattern.

```
func (uv *userValidator) emailFormat(user *User) error {
    if user.Email == "" {
        return nil
    }
    if !uv.emailRegex.MatchString(user.Email) {
        return ErrEmailInvalid
    }
    return nil
}
```

⁹<https://golang.org/pkg/regexp/#Regexp.MatchString>

The one “weird” part to this validation is the first if block. If the email address is an empty string we skip this validation. Technically we don’t need to do that, but I prefer to add this so that my validation can be used in situations where an email address isn’t required, and also because I know I have a validation that requires an email address which will catch this case and return a more specific and helpful error.

Our last step is to add this validation to our Create and Update methods. You could also add it to the ByEmail method if you want, but the worst thing that will happen if we don’t is an unnecessary query against the database that doesn’t return any results, and I find that ErrNotFound error returned in that case to be more useful for searching than the ErrEmailInvalid error.

```
func (uv *userValidator) Create(user *User) error {
    err := runUserValFns(user,
        uv.bcryptPassword,
        uv.setRememberIfUnset,
        uv.hmacRemember,
        uv.normalizeEmail,
        uv.requireEmail,
        uv.emailFormat)
    // ... this is unchanged
}
```

```
func (uv *userValidator) Update(user *User) error {
    err := runUserValFns(user,
        uv.bcryptPassword,
        uv.hmacRemember,
        uv.normalizeEmail,
        uv.requireEmail,
        uv.emailFormat)
    // ... this is unchanged
}
```

To test this out, try to sign up for an account using an email address like `j@j` - you should see the message attached to the ErrEmailInvalid variable.

```
models: email address is not valid
```

Box 12.11. Client-side validations

Many developers often think that validating the email address on the client-side (aka the “front end”) is enough, and don’t bother writing a validation like this on the web server. The problem with this approach is that you can’t always guarantee that all of your web requests will be coming from a client that you wrote, and as a result you can’t ensure that those validations will always be present.

For example, someone might write custom code to interact with your web application so they can send requests in a more automated fashion. When this happens, you can’t count on any of your client-side validations to be present and you must check the data on your server.

Source code

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

Completed - book.usegolang.com/12.6.3

Changes - book.usegolang.com/12.6.3-diff

12.6.4 Verifying an email address isn’t taken

Lastly we need to verify that email addresses are not taken by another user whenever we create or update a user. At first this sounds simple, but there is actually one tricky use case we need to consider - updating a user.

When we are creating a new user, or a user is changing their email address, our check is easy. We just need to see if the email address matches any user in our database.

Updating a user when they aren't changing their email address is trickier, because if we search for a user by the email address we will find a user in our database that has that email address - the user we are updating!

Rather than trying to cover all of these use cases individually, I have found the best way to handle this is to follow these rules:

1. Search for a user with the email address. If we DO NOT find one, our validation was successful and we can exit. Otherwise we proceed to the next step.
2. Because we found a user with the email address, we need to check the ID of that user along with the ID of the user we are validating. If the two IDs are the same, we can safely assume that the user DID NOT update their email address. If the two IDs are NOT the same, our validation failed and the user is trying to create a new account or update their email address using an email address that is in use by another account.

The check in step (2) works because new user accounts will always have the uint zero value assigned to their ID (**0**), which will never be used in our database as an ID. That means that anytime we try to create a user with an email address being used by another user we will end up comparing a real id (> 0) with zero, which will always be false.

For existing account this works because we only want to pass the validation when both accounts with the email address are actually the same resource, and the ID is how we verify that they are the same user.

Now that we know how we are going to proceed in writing our code, let's start once again by defining an error and adding it to the errors in our source file.

```
var (
    // ErrEmailTaken is returned when an update or create is attempted
    // with an email address that is already in use.
    ErrEmailTaken = errors.New("models: email address is already taken")
)
```

Next up is the validation function which runs the logic we discussed.

```
func (uv *userValidator) emailIsAvail(user *User) error {
    existing, err := uv.ByEmail(user.Email)
    if err == ErrNotFound {
        // Email address is available if we don't find
        // a user with that email address.
        return nil
    }
    // We can't continue our validation without a successful
    // query, so if we get any error other than ErrNotFound we
    // should return it.
    if err != nil {
        return err
    }

    // If we get here that means we found a user w/ this email
    // address, so we need to see if this is the same user we
    // are updating, or if we have a conflict.
    if user.ID != existing.ID {
        return ErrEmailTaken
    }
    return nil
}
```

Box 12.12. Avoiding cyclical method calls

In the emailIsAvail validation we call another method on the userValidator - the ByEmail method. Doing this is fine, but be careful that you don't create a cycle where method A calls B, then method B calls A again. Doing so could cause your code to stop working as intended as it continuously cycles through the loop of method calls.

This is often easy to miss, especially when you have a longer chain. Eg:

```
A -> B -> C -> D -> A
```

There are no hard rules to prevent this; instead you will just need to be vigilant in reviewing your code for cycles. If they do occur, chances are you will see it when testing our your new code as it will feel like your server has frozen until it eventually errors.

Finally, we update our Create and Update methods adding this validation.

```
func (uv *userValidator) Create(user *User) error {
    err := runUserValFns(user,
        uv.bcryptPassword,
        uv.setRememberIfUnset,
        uv.hmacRemember,
        uv.normalizeEmail,
        uv.requireEmail,
        uv.emailFormat,
        uv.emailIsAvail)
    // ... this is unchanged
}
```

```
func (uv *userValidator) Update(user *User) error {
    err := runUserValFns(user,
        uv.bcryptPassword,
        uv.hmacRemember,
        uv.normalizeEmail,
        uv.requireEmail,
        uv.emailFormat,
        uv.emailIsAvail)
    // ... this is unchanged
}
```

Source code

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

Completed - book.usegolang.com/12.6.4

Changes - book.usegolang.com/12.6.4-diff

12.7 Cleaning up our error naming

Before moving on to write more validations, we are going to take a moment to clean up some of our error naming. What we have works, but it isn't consistent. That is my fault, but it is something that happens in a real code base so I wanted to illustrate how to handle situations like this when they do arrise.

If you look at our errors, we have errors like "ErrInvalidID" as well as errors like "ErrEmailTaken". I consider these inconsistent because the latter has "Err" followed by the field name, and then followed by a brief description, whereas the first error has "Err" followed by the brief description and then finally the field name.

This might not seem important right now, when you are using an editor with autocomplete it can be incredibly helpful to have all of the email related errors show up when you type **ErrEmail** and then pause. We are going to update our error names to address this inconsistency, which means we need to renamed the ErrInvalidID and ErrInvalidPassword errors.

While we are at it, we are going to rename the ErrInvalidPassword error to instead be ErrPasswordIncorrect, changing from the term invalid to incorrect. We are doing this because we typically use the term invalid in our application to mean that something isn't valid, but we are using this specific error only when a user provides an incorrect password when logging in.

Later when we add rules for our passwords we might decide that an invalid password error is useful there, but for now we only need the incorrect password error.

You can perform this refactor however you want, but if you want to learn how to use the **gorename** tool I suggest checking out Appendix 18.2, which outlines how to do this entire refactor using the tool.

After renaming your code your variable block should look roughly like the one below.

```
var (
    // Most errors here remain unchanged, but we are changing
    // ErrInvalidID and ErrInvalidPassword to match the new
    // errors below.

    // ErrIDInvalid is returned when an invalid ID is provided
    // to a method like Delete.
    ErrIDInvalid = errors.New("models: ID provided was invalid")

    // ErrPasswordIncorrect is returned when an incorrect
    // password is used when authenticating a user.
    ErrPasswordIncorrect = errors.New(
        "models: incorrect password provided")
)
```

You also need to update the controllers package. If you used gorename, it should have done this for you. Otherwise open up the source file.

```
$ atom controllers/users.go
```

Look for the **Login** method. In it we have a case that uses the old ErrInvalid-Password error. Change it to use the updated variable name.

```
if err != nil {
    switch err {
    case models.ErrNotFound:
```

```

    fmt.Fprintln(w, "Invalid email address.")
  case models.ErrPasswordIncorrect:
    fmt.Fprintln(w, "Invalid password provided.")
  default:
    http.Error(w, err.Error(), http.StatusInternalServerError)
}
return
}

```

After renaming your variables, even if you used gorenname, there may still be a few comments that need updated. Gorenname updates comments right above a variable or type, but typically does not address other comments in source code because it can't be certain they reference the same variable.

Find the UserService interface and update the comment above the Authenticate method.

```

type UserService interface {
    // Authenticate will verify the provided email address and
    // password are correct. If they are correct, the user
    // corresponding to that email will be returned. Otherwise
    // You will receive either:
    // ErrNotFound, ErrPasswordIncorrect, or another error if
    // something goes wrong.
    Authenticate(email, password string) (*User, error)
    UserDB
}

```

We also need to check the implementation of this method, as we had a comment mentioning the password error there as well.

```

// Authenticate can be used to authenticate a user with the
// provided email address and password.
// If the email address provided is invalid, this will return
// nil, ErrNotFound
// If the password provided is invalid, this will return
// nil, ErrPasswordIncorrect
// If the email and password are both valid, this will return
// user, nil
// Otherwise if another error is encountered this will return

```

```
// nil, error
func (us *userService) Authenticate(email,
    password string) (*User, error) {
    // ... don't change this, just the comment above
}
```

With that we should be done refactoring our errors and we can move on to writing our password validations.

Source code

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

Completed - book.usegolang.com/12.7

Changes - book.usegolang.com/12.7-diff

12.8 Validating and normalizing passwords

In this section we are going to focus on writing validations for our password field. We already have some of these written, such as the normalizer to hash our password; all that is left to do is verify the passwords minimum length and when a user is created or updated, and ensure that a password is provided when creating a user.

12.8.1 Verifying passwords have a minimum length

Our first validation is one that verifies that our password has at least eight characters in it. This might get run when a user hasn't updated their password, so we need to be sure to only run this whenever a new password has been provided.

Like many validations, we will start with an error.

```
var (
    // ErrPasswordTooShort is returned when a user tries to set
    // a password that is less than 8 characters long.
    ErrPasswordTooShort = errors.New("models: password must " +
        "be at least 8 characters long")
)
```

With our error created we can get to work on the validation. If the password hasn't changed it should be the empty string. Otherwise we need to validate the password length. We won't worry about passwords being required on user creation just yet because we will write that validation next.

```
func (uv *userValidator) passwordMinLength(user *User) error {
    if user.Password == "" {
        return nil
    }
    if len(user.Password) < 8 {
        return ErrPasswordTooShort
    }
    return nil
}
```

We need to add this to both the Update and Create methods so that it gets used, but this is one of our validations where the order it gets run in matters. We *must* run this validation before hashing our password, otherwise the password will be set to the empty string and we can't figure out the original length of a password when looking at the hash.

```
func (uv *userValidator) Create(user *User) error {
    err := runUserValFns(user,
        uv.passwordMinLength,
        uv.bcryptPassword,
        uv.setRememberIfUnset,
        uv.hmacRemember,
        uv.normalizeEmail,
        uv.requireEmail,
        uv.emailFormat,
```

```
    uv.emailIsAvail)
    // ... this is unchanged
}
```

```
func (uv *userValidator) Update(user *User) error {
    err := runUserValFns(user,
        uv.passwordMinLength,
        uv.bcryptPassword,
        uv.hmacRemember,
        uv.normalizeEmail,
        uv.requireEmail,
        uv.emailFormat,
        uv.emailIsAvail)
    // ... this is unchanged
}
```

Source code

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

Completed - book.usegolang.com/12.8.1

Changes - book.usegolang.com/12.8.1-diff

12.8.2 Requiring a password

The last password validation we need to write is one that verifies that the password is provided whenever we create a new user. While we are at it, we are also going to write a validation to ensure that our PasswordHash is always given a value.

In both cases we will return the same error. This could lead to confusion in some cases, but in practice it tends to work well for both of these validations.

```
var (
    // ErrPasswordRequired is returned when a create is attempted
    // without a user password provided.
    ErrPasswordRequired = errors.New("models: password is required")
)
```

We will start with the validation to make sure a password is provided whenever creating a user.

```
func (uv *userValidator) passwordRequired(user *User) error {
    if user.Password == "" {
        return ErrPasswordRequired
    }
    return nil
}
```

We only need to add this one to the create method, as updates don't *require* a password. We'll wait to add that code until after we write our second validation - the one that ensures we always have a value set on the PasswordHash field.

```
func (uv *userValidator) passwordHashRequired(user *User) error {
    if user.PasswordHash == "" {
        return ErrPasswordRequired
    }
    return nil
}
```

We need to add both of these errors to the Create method, and the ordering matters. We need to require the password before doing any other password validations, and then require the password hash after hashing our password because it might not have a value set prior to then.

```
func (uv *userValidator) Create(user *User) error {
    err := runUserValFns(user,
        uv.passwordRequired,
        uv.passwordMinLength,
```

```
uv.bcryptPassword,
uv.passwordHashRequired,
uv.setRememberIfUnset,
uv.hmacRemember,
uv.normalizeEmail,
uv.requireEmail,
uv.emailFormat,
uv.emailIsAvail)
// ... this is unchanged
}
```

Update only needs the password hash validation, and once again it needs to be placed after the bcryptPassword function.

```
func (uv *userValidator) Update(user *User) error {
err := runUserValFns(user,
uv.passwordMinLength,
uv.bcryptPassword,
uv.passwordHashRequired,
uv.hmacRemember,
uv.normalizeEmail,
uv.requireEmail,
uv.emailFormat,
uv.emailIsAvail)
// ... this is unchanged
}
```

To test this, start your server up and try to create an account without providing a password. You should see the “password is required” error message.

Source code

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

Completed - book.usegolang.com/12.8.2

Changes - book.usegolang.com/12.8.2-diff

12.9 Validating and normalizing remember tokens

In the final section of this chapter we are going to be writing validations for our remember tokens. We will be adding one validation to ensure that our remember tokens are at least 32 bytes when changed, and another to ensure that the hashed remember token is present on every user.

We can't determine if a remember token is at least 32 bytes without a way to check the number of bytes in our string, and our models package doesn't have a lot of info about how the remember token is created. That logic is handled in the `rand` package we created.

The simplest solution to this problem is to add an `NBytes` function to our `rand` package that can be used to calculate the number of bytes used in any hash string generated by our package's `String` or `RememberToken` functions.

```
$ atom rand/strings.go
```

Our function will essentially be the opposite of the `String` function. It will first base 64 decode the string, and then we will calculate how many bytes are in the resulting byte slice. It is possible to get an error if the data provided isn't truly base 64 encoded, so we also need to return that error if it pops up.

```
// NBytes returns the number of bytes used in any string
// generated by the String or RememberToken functions in
// this package.
func NBytes(base64String string) (int, error) {
    b, err := base64.URLEncoding.DecodeString(base64String)
    if err != nil {
        return -1, err
    }
    return len(b), nil
}
```

Now we need to start utilizing this in our users model in a validation function, but we are going to need a few more errors first. We will need an error for

when our remember token doesn't have enough bytes used in it, and we will also need an error for when a remember token or remember token hash isn't set and is required. We will be adding these to our users model.

```
$ atom models/users.go
```

```
var (
    // ErrRememberRequired is returned when a create or update
    // is attempted without a user remember token hash
    ErrRememberRequired = errors.New("models: remember token "+
        "is required")

    // ErrRememberTooShort is returned when a remember token is
    // not at least 32 bytes
    ErrRememberTooShort = errors.New("models: remember token "+
        "must be at least 32 bytes")
)
```

With our error variables and the NBytes functions ready, we can move on to creating validation functions. We can start with the one that checks the number of bytes used to make a remember token.

Because remember tokens aren't always updated, we first need to see if a remember token is provided. If not we can assume the user is sticking with an existing remember token and trust that other validations will catch it if it truly is an error.

Assuming a remember token is being changed, we will check the number of bytes and return an error if it is less than 32; otherwise we return nil.

```
func (uv *userValidator) rememberMinBytes(user *User) error {
    if user.Remember == "" {
        return nil
    }
    n, err := rand.NBytes(user.Remember)
    if err != nil {
        return err
    }
    if n < 32 {
```

```

    return ErrRememberTooShort
}
return nil
}

```

Note: We could write this similar to our idGreater Than validator, but we aren't likely to change this value frequently so I find hard coding it works well enough.

We will add this to our Update and Create methods in a moment, but first let's also write a function to ensure that our remember hash is always set when we update and create users.

```

func (uv *userValidator) rememberHashRequired(user *User) error {
    if user.RememberHash == "" {
        return ErrRememberRequired
    }
    return nil
}

```

Now we can update the Create and Update methods to use our two new validators. Be sure to call the rememberMinBytes function before we hash the remember token, and then call the rememberHashRequired function after the function that hashes the remember token.

```

func (uv *userValidator) Create(user *User) error {
    err := runUserValFns(user,
        uv.passwordRequired,
        uv.passwordMinLength,
        uv.bcryptPassword,
        uv.passwordHashRequired,
        uv.setRememberIfUnset,
        uv.rememberMinBytes,
        uv.hmacRemember,
        uv.rememberHashRequired,
        uv.normalizeEmail,
        uv.requireEmail,
        uv.emailFormat,
        uv.emailIsAvail)
    // ... this is unchanged
}

```

```
func (uv *userValidator) Update(user *User) error {
    err := runUserValFns(user,
        uv.passwordMinLength,
        uv.bcryptPassword,
        uv.passwordHashRequired,
        uv.rememberMinBytes,
        uv.hmacRemember,
        uv.rememberHashRequired,
        uv.normalizeEmail,
        uv.requireEmail,
        uv.emailFormat,
        uv.emailIsAvail)
    // ... this is unchanged
}
```

With that we are done writing validations for our user resource. We might find later that we need to create more, but this should be good enough for now.

At this point it is advisable to drop your database using the `DestructiveReset` function within `main.go` like we have in the past, and then to restart your server. It is very likely that you have invalid data stored in your data that was created before we made these validations, and this could potentially cause issues. Starting from scratch is a much safer option while we are in development.

After that head on to the next chapter where we will learn how to render nice error messages in our UI. This will give our users a much better experience than seeing a white page with a single error message when they try to sign up with an invalid email address.

Source code

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

Completed - book.usegolang.com/12.9

Changes - book.usegolang.com/12.9-diff

Chapter 13

Displaying errors to the end user

Our application has been steadily improving, but there is still one area we are falling short on; we haven't taken the time to write any reasonable error handling in our application. Instead, whenever we run into an error we often panic or just display the error message on the screen on a plain white page. This works well for getting up and running quickly, but we don't want to deploy our application in that state.

In this chapter we are going to focus on improving our error handling. We will start by learning about Bootstrap Alerts¹, which are a design element of bootstrap that are useful for showing error messages and other notices. After that we will look at how to update our code so that we can display error messages in those alerts while also re-rendering whatever form or page the user was on.

¹<http://getbootstrap.com/docs/3.3/components/#alerts>

13.1 Rendering alerts in the UI

Before jumping into specific errors, the first thing we are going to look at is the HTML we are going to user to render alert messages. We will be using Bootstrap, our HTML framework, to do this. Specifically, we will be using the dismissible bootstrap alerts² component to display our error messages.

Box 13.1. Check out the Bootstrap docs

When you have a spare moment, take some time to check out the Bootstrap docs. Look at the various components available, different options for each component, and try to get a general feel for what all is there.

You don't need to memorize the docs, but it is often nice to have a general idea of what pieces there are to work from. As you start to build web applications, especially early versions and prototypes, the most efficient path to launching is often in using existing bootstrap components until you get the major components of your web application working. After that you can look into using custom components, themes, and giving your application a unique look and feel, but it is often a bad idea to focus on this until you settle on a user experience.

There are four types of alert: success, info, warning, and danger. The only difference between these in the UI is going to be what color they are rendered as. With a default Bootstrap setup success will be green, info will be blue, warning will be yellow, and danger will be red, but these colors can all be customized with bootstrap and you can see a bit of this with the themes on Bootswatch³.

In this course we will mostly be using the **danger** class for our alerts, which is typically used for error messages. That is why this alert type is displayed in

²<https://getbootstrap.com/docs/3.3/components/#alerts-dismissible>

³<https://bootswatch.com/>

red.

Let's go ahead and create an HTML template for our alerts. This should feel somewhat familiar, as we have added other templates in the past. We will be adding this code to a new file in our layouts directory, so we can start by creating the source file.

```
$ atom views/layouts/alert.gohtml
```

Once we have our source file open, our next step is to define the **alert** template. That way we can call this from our other views wherever we need to display an alert message.

```
{{define "alert"}}
<div class="alert alert-danger alert-dismissible"
  role="alert">
  <button type="button" class="close" data-dismiss="alert"
    aria-label="Close">
    <span aria-hidden="true">&times;</span>
  </button>
  <strong>Danger!</strong> Something went wrong! Oh no!
</div>
{{end}}
```

Our **alert.gohtml** source file is only defining a template at this point, so if we want to render it we need to call that template. The simplest way to do this for now is to add it to our layout file.

```
$ atom views/layouts/bootstrap.gohtml
```

We won't do any error checking for now, and will instead always render the alert. This isn't going to work long term, but will give us a sense of what the alert looks like in our app and verify our code is working.

Look for the container-fluid div in your bootstrap template's HTML and update it with the code below.

```
<div class="container-fluid">
  {{template "alert"}}
  {{template "yield" .}}
  {{template "footer"}}
</div>
```

Restart the server, and visit a page in the application. We should now be seeing a red alert at the top of our page similar to Figure 13.1.

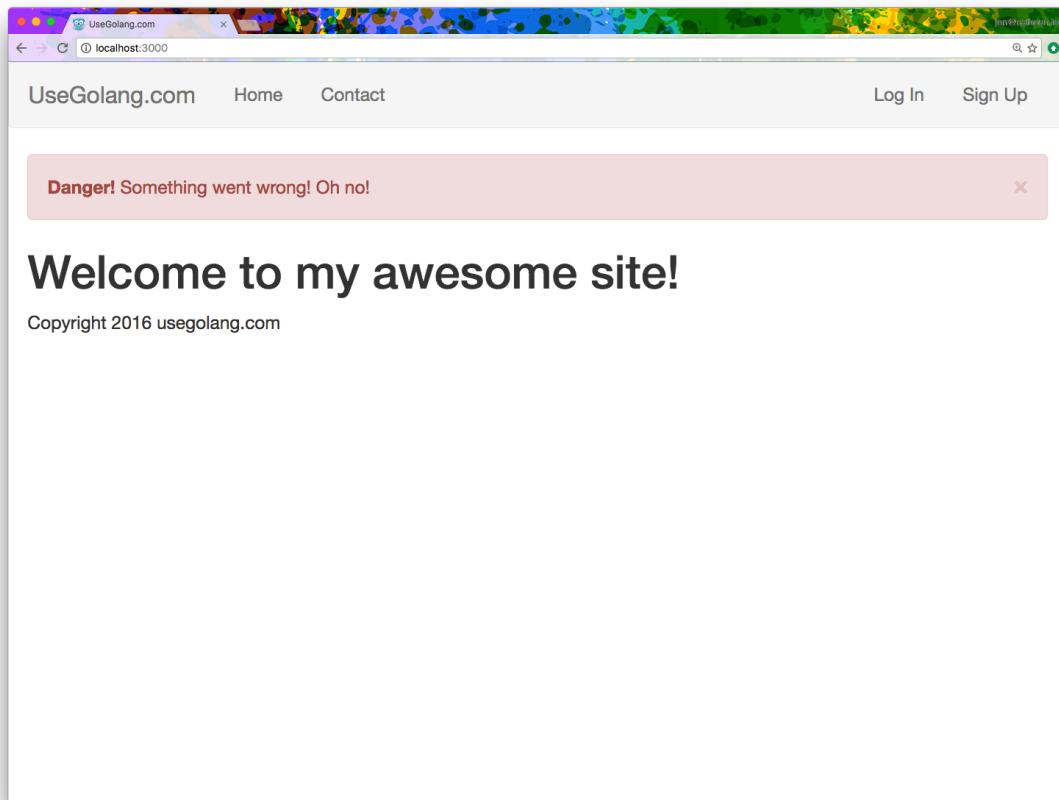


Figure 13.1: Our website with the Bootstrap alert displayed across the top

Source code

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

Completed - book.usegolang.com/13.1

Changes - book.usegolang.com/13.1-diff

13.2 Rendering dynamic alerts

Alerts are pretty useless if they only render static data. We need to be able to insert our specific errors into our alerts to make them useful.

In this section we are going to update our alert template to use dynamic data. We will start be updating the alert template itself, so we can start by opening that source file.

```
$ atom views/layouts/alert.gohtml
```

Once we have our alert open we are going to look it over and decide which pieces of information should be set dynamically. The first piece that sticks out is the “alert-danger” class inside of the outer **div**. This class is what dictates whether our alert is going to be a danger, warning, info, or success alert, and there is a good chance we will want to show success or info messages at some point in our application so we should set this dynamically.

Given that this class dictates the severity level of the error, I tend to call this variable **Level**. We also don’t have to set the entire HTML class dynamically; we can insert our variable mid-string and it will end up creating a single HTML class. The end result is the following piece of code for our opening div.

```
<div class="alert alert-{{.Level}} alert-dismissible" role="alert">
```

The next few lines define an HTML button which is used to render the little **x** at the top right of the alert allowing users to dismiss it. None of that needs to be dynamic, so we can move past it.

The last bit of HTML inside the alert div is the message we render inside of our alert. That is definitely something we want to render dynamically, so we are going to replace that entire line with the **Message** variable. The resulting alert template is shown below.

```
 {{define "alert"}}
<div class="alert alert-{{.Level}} alert-dismissible" role="alert">
  <button type="button" class="close" data-dismiss="alert"
    aria-label="Close">
    <span aria-hidden="true">&times;</span>
  </button>
  {{.Message}}
</div>
{{end}}
```

In order to render this alert properly we are going to need to pass in a piece of data with both a Level and a Message field. We will eventually be building out some data types to help us when rendering views, but for now we are going to create a temporary type that we can use to render our alert. We need to do this from within a controller, so we will be temporarily altering the users controller.

```
$ atom controllers/users.go
```

Once your source file is open, find the **New** method. We are going to be editing this method so that we can test our alerts on the signup page.

Inside the New method, we will start by creating a data type with both a Level and Message field.

```
type Alert struct {
    Level  string
    Message string
}
```

Next we want to create an instance of this that we can use to render our alert. We will test this out with a success alert rather than a danger one.

```
alert := Alert{
    Level:  "success",
    Message: "Successfully rendered a dynamic alert!",
}
```

And finally we need to pass this alert as data into the Render method when we go to render a view. Putting that all together, your New method should look like the code below.

```
func (u *Users) New(w http.ResponseWriter, r *http.Request) {
    type Alert struct {
        Level  string
        Message string
    }
    alert := Alert{
        Level:  "success",
        Message: "Successfully rendered a dynamic alert!",
    }
    if err := u.NewView.Render(w, alert); err != nil {
        panic(err)
    }
}
```

We are just about ready to test out our alert, but we need to update our template call made in our bootstrap template.

```
$ atom views/layouts/bootstrap.gohtml
```

When we originally called the alert template, we did so without passing in any data to it. That is why the original call doesn't have any arguments listed after the template name.

```
 {{template "alert"}}
```

We want to pass in this alert data, so we are going to update that line of code to include a period at the end. The period signifies that we are passing in all the data provided to our bootstrap template, which in this case happens to only be the Alert we just created.

```
<div class="container-fluid">
  <!-- Add the period after "alert" below -->
  {{template "alert" .}}
  {{template "yield" .}}
  {{template "footer" .}}
</div>
```

To see our code in action, restart the server and head to the `/signup` path. That is the page rendered by our `New` method on the users controller.

If everything worked as planned, we should be seeing a success alert message with the message we created earlier.

Source code

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

Completed - book.usegolang.com/13.2

Changes - book.usegolang.com/13.2-diff

13.3 Only display alerts when we need them

Now that we have our alert rendering dynamic data we have another issue. We clearly don't want want to generate this alert HTML all the time, but instead only want to render it when we actually have an alert message to display.

In order to do this, we are going to take advantage of Go's template actions⁴, which are part of both the text/template and html/template package.

Note: The link for template actions directs you to the html/template package because this is the only location that the docs are shown, but the actions there are all available in the html/template package we are using to render our HTML responses.

The action we are going to be using is the the **if** action, which allows us to check if a argument is empty or not. If it is empty, the portion inside of the if block will NOT be rendered. If the argument passed to the if statement is NOT empty, then the entire section inside the if block will be rendered.

Box 13.2. What are empty values?

When dealing with templates, empty values are defined as:

The empty values are false, 0, any nil pointer or interface value, and any array, slice, map, or string of length zero.

This is taken directly from the text/template documentation, and it is important to understand because if statements inside of templates are fairly different from if statements inside of Go code. In Go code we expect the arguments to an if statement to evaluate to true or false, but we can pass things like strings into the **if** action for templates.

⁴<https://golang.org/pkg/text/template/#hdr-Actions>

That means we could update our bootstrap template with code similar to the code below.

```
<div class="container-fluid">
  {{if .}}
    {{template "alert" .}}
  {{end}}
  {{template "yield" .}}
  {{template "footer"}}
</div>
```

Note: We won't be using this code; it is meant for illustration purposes only.

In the code above we will check to see if *any* non-empty data was passed into our template, and if it was we would try to render the alert template using that data. We would then continue to pass that same data into the yield template as well.

That list bit sounds problematic. We don't want to render the alert if there is any data in our template, but instead want to render it when there specifically is an alert message. We also don't want to pass the alert message into our yield template, but would rather pass data meant to be rendered in there.

Let's head back to our users controller and look at one way to solve this problem.

```
$ atom controllers/users.go
```

We know we want to separate our alert data from the data used to render our yield template, so why don't we simply create a type that has two fields - Alert and Yield. Then we will know for sure which piece of data is intended for each template.

Note: In case you forgot, the yield template is used to render each specific page. For example, we use it to render the sign up form, the home page, and any non-layout information.

We are going to add this code inside of our **New** method again, and once we create our new type we will create an instance, assign some data to try it out with, and then pass it into our Render method.

```
func (u *Users) New(w http.ResponseWriter, r *http.Request) {
    type Alert struct {
        Level string
        Message string
    }
    type Data struct {
        Alert *Alert
        Yield interface{}
    }
    alert := Alert{
        Level: "success",
        Message: "Successfully rendered a dynamic alert!",
    }
    data := Data{
        Alert: &alert,
        Yield: "this can be any data b/c its type is interface",
    }
    if err := u.NewView.Render(w, data); err != nil {
        panic(err)
    }
}
```

Now that we have these two pieces of data separated, let's head back to our bootstrap layout and update it to reflect this change.

```
$ atom views/layouts/bootstrap.gohtml
```

We only want to render the alert message if we have an Alert argument that isn't empty in our template, so we can start there. If it is indeed non-empty, we are going to render the alert template and pass in that piece of data.

```
{{if .Alert}}
{{template "alert" .Alert}}
{{end}}
```

The Yield field is a little different. Even if it is empty, we still want to render the yield template as this might have a page with static data, such as the home page. In that case we just need to call the template with the correct data.

```
<div class="container-fluid">
  {{if .Alert}}
    {{template "alert" .Alert}}
  {{end}}
  {{template "yield" .Yield}}
  {{template "footer"}}
</div>
```

With the updated code we are telling our template only to pass specific data to each template. Once inside of that template, all of the data passed in will be accessible with the dot (eg `{{.}}`) much like if we passed in a single field to another function as an argument. You can see this by updating the new users template.

```
$ atom views/users/new.gohtml
```

Add the following h1 tag directly after the template is defined.

```
{{define "yield"}}
<h1>{{.}}</h1>

{{end}}
```

Now restart your server and visit the sign up page at `/signup`. Not only will you see a success alert, but you will also see the string “this can be any data b/c its type is interface” that we set to the Yield field because we are accessing it via the dot `(.)` in our new.gohtml template.

In the next section we are going to start moving towards a more permanent data structure for our views data, but in the meantime this section has hopefully given you an introductory look into how we can start to add and interact with data in our templates to affect how our pages are rendered.

Source code

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

Completed - book.usegolang.com/13.3

Changes - book.usegolang.com/13.3-diff

13.4 A more permanent data type for views

In the last few sections we explored how having a predefined data type can be useful when rendering views. With it we can still place more dynamic page-specific data into a **yield** field, while still having more common fields like **Alert** available when we want to show the user an alert message.

In this section we are going to define a more permanent data type for rendering views. We will name this struct type **Data**, and will be placing it inside the **views** package because it is used in rendering our views. While doing this we will also create an **Alert** data type in the views package to be used when rendering our alerts.

Let's start off by adding the new data types in the views package. We will create a new source file for this code.

```
$ atom views/data.go
```

Inside of this file we need to first introduce the Data and Alert types. For now this can be identical to the one we created earlier inside of our controllers.

```

package views

// Data is the top level structure that views expect data
// to come in.
type Data struct {
    Alert *Alert
    Yield interface{}
}

// Alert is used to render Bootstrap Alert messages in templates
type Alert struct {
    Level string
    Message string
}

```

Up until now we have let data flow into our views in almost any format, but moving forward we are going to start enforcing a little more rigidity. We are going to start expecting all views to receive a Data object if they pass in any data at all.

While we are creating our Alert type, it is probably also a good idea to introduce some constants for each of our alert levels. That way developers can easily choose from predefined alert levels that our UI supports and not have to worry about typos in their strings.

```

const (
    AlertLvlError    = "danger"
    AlertLvlWarning  = "warning"
    AlertLvlInfo     = "info"
    AlertLvlSuccess  = "success"
)

```

Now let's update our New method inside the users controller to use our new data type.

```
$ atom controllers/users.go
```

We no longer need to define the Data and Alert types, so we can get rid of all that code, and we don't really need to add any new code. All we really need to do is update our code that creates the Alert and Data objects to use the ones defined in the views package. We can also update our code to use one of the constant alert levels.

```
func (u *Users) New(w http.ResponseWriter, r *http.Request) {
    alert := views.Alert{
        Level:  views.AlertLvlSuccess,
        Message: "Successfully rendered a dynamic alert!",
    }
    data := views.Data{
        Alert: &alert,
        Yield: "this can be any data b/c its type is interface",
    }
    if err := u.NewView.Render(w, data); err != nil {
        panic(err)
    }
}
```

At this point we can restart our server and our code will be working again, but there is one final thing we are going to do before moving on. We are going to update our **Render** method inside the views package to make sure all data passed into a view is wrapped inside of the Data type we just created.

```
$ atom views/view.go
```

Look for the Render method and update it to reflect the code below. We will cover what is going on in the code after making the changes.

```
func (v *View) Render(w http.ResponseWriter, data interface{}) error {
    w.Header().Set("Content-Type", "text/html")
    switch data.(type) {
    case Data:
        // do nothing
    default:
        data = Data{
            Yield: data,
```

```

    }
}

return v.Template.ExecuteTemplate(w, v.Layout, data)
}

```

In our updated code we added a switch case that checks the underlying type of any data passed into our Render method using a type switch⁵. With this we can safely determine what type our **data** argument has and act accordingly.

The only case we really care about is whether the data is of the new Data type we just created or not. If it is of the Data type we don't need to do anything. That is what our views expect, so we can leave the data along.

If the data is any other type we know that it isn't in the format our views expect. One way to handle this might be to return an error, but in our case we are instead going to wrap it inside of a new Data object, setting this data to the Yield field.

We are doing this because we are assuming that any data provided to the Render method that isn't of the Data type is only meant to be used inside the specific page we are rendering, and we store that data inside the Yield field of our Data objects. In short, this is more of a convenience that will allow us to render views with code like:

```
someView.Render(w, "this is page-specific data!")
```

And our Render method will handle making that happen for us.

Source code

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

⁵<https://tour.golang.org/methods/16>

Completed - book.usegolang.com/13.4

Changes - book.usegolang.com/13.4-diff

13.5 Handling errors in the sign up form

We are now ready to start implementing proper error handling into our application, but first we need to remove some of the code we were using to demonstrate how our alerts and view data works.

Open up the new users template and remove the **h1** tag we added in Section 13.3.

```
$ atom views/users/new.gohtml
```

```
{{define "yield"}}
<!-- Remove the h1 tag below -->
<!-- <h1>{{.}}</h1> -->

<!-- ... the rest of the code stays the same -->
{{end}}
```

After that open up the users.go source file inside the controllers package. Inside of it we are going to remove the code used to create an alert inside of our New method.

```
$ atom controllers/users.go
```

After removing the code, your New method should be back to the original bare version.

```
func (u *Users) New(w http.ResponseWriter, r *http.Request) {
    if err := u.NewView.Render(w, nil); err != nil {
        panic(err)
    }
}
```

Now find the Create method inside of the users controller. We are going to be working on updating this method to add better error handling whenever there is an issue processing a sign up form.

We know that we when we want to render an alert we need to set the Alert field of the `views.Data` type, so we can start off by adding a variable of this type. If we never have an error we can pass this into our view with no changes, but if there is an error we then only need to set it on the variable and it will be rendered in our view.

```
func (u *Users) Create(w http.ResponseWriter, r *http.Request) {
    var vd views.Data

    // ... nothing else changes
}
```

After that we look through our source code looking for any situations that could result in an error. Up until now we have been handling these with either a panic or by calling a method like `http.Error`, so we can just look for those.

Inside the Create method the first panic shows up right after attempting to parse the form. Errors here aren't likely to occur, but when they do we don't really know what error messages to expect. Our `parseForm` function returns errors generated by both the `net/http` package, as well as errors created by the `gorilla/schema` package.

Rather than trying to handle all of those errors individually, we are going to create an alert displaying a generic error message letting the user that something went wrong. We will then render the sign up form again with the alert and let the user resubmit the form again if they want to retry.

Box 13.3. Displaying unfiltered errors

Generally speaking, displaying errors to an end user without knowing exactly what the message says is a bad idea. When we create the error on our own it is often okay to render the message because we know what it says, but with errors we do not create on our own it can often be hard to tell if that error message has any sensitive information in it.

Rather than displaying unknown errors, the appropriate behavior is to either create errors of your own based on what caused the error, or to render a generic error message to the end user. If you believe you might need additional information from the error message you should use a logging or alerting system to notify developers rather than rendering it to the end user.

Rendering a generic error could be done using code roughly like the code below, but anytime we have hard coded strings in our application we should take a moment to ask ourselves if these would be suited to instead be a constant provided by one of our packages.

```
vd.Alert = &views.Alert{  
    Level:  views.AlertLvlError,  
    Message: "Something went wrong!",  
}  
u.NewView.Render(w, vd)
```

In this case the error message definitely is something that should be a constant. It is incredibly likely that we will be displaying a generic error message in many different cases, and we won't end up with many different generic error messages in our application.

Open up the source file where we declared our Data type for views.

```
$ atom views/data.go
```

Inside of this source file we are going to add the constant AlertMsgGeneric that can be used in any situation where we need a generic error message.

```
const (
    AlertLvlError    = "danger"
    AlertLvlWarning  = "warning"
    AlertLvlInfo     = "info"
    AlertLvlSuccess  = "success"

    // AlertMsgGeneric is displayed when any random error
    // is encountered by our backend.
    AlertMsgGeneric = "Something went wrong. Please try " +
        "again, and contact us if the problem persists."
)
```

After that save your code and head back to the users controller source code we were just editing. Find the Create method and then find the if statement where we parse our form and panic if there is an error. Inside of this if block we are going to add the code to create an Alert and render the NewView with that alert whenever an error occurs. While we are at it, we will also log out the error so we have access to the real error message in case we need to debug the issue.

```
import (
    // Add the log package to our imports
    // It works similar to the fmt package, and has a Println
    // function that will print out data to the terminal
    "log"
)

func (u *Users) Create(w http.ResponseWriter, r *http.Request) {
    var vd views.Data
    var form SignupForm
    if err := parseForm(r, &form); err != nil {
        log.Println(err)
        vd.Alert = &views.Alert{
            Level:  views.AlertLvlError,
            Message: views.AlertMsgGeneric,
        }
    }
}
```

```
    u.NewView.Render(w, vd)
    return
}

// ... The rest of the method remains unchanged for now
}
```

As we progress through this chapter we are going to continue writing code like this; code where we log an error, create an alert, and then render a view with that alert. The important thing to takeaway here is that none of this is going to be automated right now, and we need to explicitly state that we want to render an error when something goes wrong.

This is part of the reason why Go's error handling is very different from what you will see in other languages. In languages where you can "throw" or "raise" exception, it is often unclear what code can and can't return an error, so it is easy to miss an error. As a result, testing all your error paths can be hard and it can also be challenging to capture them all and render them correctly.

Go is at the other end of this spectrum; errors are in the forefront of your application and are returned just like any other data, making it crystal clear whenever an error can be encountered. Not only does this make it easier to test your code, but it also makes it easier to write code that properly renders each error based on what it might be.

Enough of the lecturing; let's continue looking at our Create method for other cases where we are handling errors. The next one occurs when we attempt to create a user with the user service.

```
if err := u.us.Create(&user); err != nil {
    http.Error(w, err.Error(), http.StatusInternalServerError)
    return
}
```

Rather than rendering an error with the `http.Error` function, we are going to create an alert and render our NewView again.

```

if err := u.us.Create(&user); err != nil {
    vd.Alert = &views.Alert{
        Level:  views.AlertLvlError,
        Message: err.Error(),
    }
    u.NewView.Render(w, vd)
    return
}

```

In this case we are doing two things different from before. We aren't logging the error message, and we are instead using the error message directly inside of our alert. But didn't we just get done talking about how it is a bad idea to display error messages directly to users?

Long term what we are doing here is a bad idea and we will be updating our code in the next section to fix it, but for now we are going to display the error message directly in our alert so that we can see error messages like, "email address is already taken" in the alert.

The final error case we need to handle in the Create method occurs whenever our **signIn** method fails for any reason. At this point we know the user resource has been properly created, but for some reason we weren't able to log the user in.

Errors that occur at this point are typically caused by database outages or some other similar issue, so it is often simpler to just redirect the user to the login page and allow them to log into their new account. While this user experience isn't perfect, keep in mind that this is very unlikely to happen and is the worst-case scenario.

```

err := u.signIn(w, &user)
if err != nil {
    http.Redirect(w, r, "/login", http.StatusFound)
    return
}

```

With that last bit of code we have successfully implemented error handling for

our sign up form. Whenever a user attempts to create a new user account and experiences an error they should see a “danger” alert displaying an error message explaining what went wrong, or in the case of a generic error our application will log the error.

Restart your server and experiment with the sign up form. Try creating an account with an email address that is already taken, no password, or a really short password. In each case you should see a different error message, but each message is prefixed with the “models:” part of the error message.

In the next section we will look at how to start white-listing error messages and rendering them with an error message designed to be read by end users, not developers.

Source code

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

Completed - book.usegolang.com/13.5

Changes - book.usegolang.com/13.5-diff

13.6 White-listing error messages

In [Section 13.5](#) we saw that there are going to be times when we want to render our errors to end users, giving them a chance to understand why we couldn’t accept their input and to fix the issue. We also discussed why it is a bad idea to expose every error message to end users, as this could lead to data leaks.

In this section we are going to discuss how to white-list certain error messages while rendering a generic error message for others. In doing this we will make rendering errors much easier moving forward.

While we are making these changes we will also be updating our errors to make them constants instead of variables using a technique introduced by Dave Cheney in his “Constant errors”⁶ blog post. This will allow us to define our errors as constants rather than variables that could be changed while our program is running.

We are going to be starting out in the views package.

```
$ atom views/data.go
```

In this source file we are going to add a method that allows us to easily set an alert on our Data type using an error. The simple version of this method is shown below, and we will be tweaking the code a bit as we progress.

```
func (d *Data) SetAlert(err error) {
    d.Alert = &Alert{
        Level: AlertLvlError,
        Message: err.Error(),
    }
}
```

In our new SetAlert method we are taking an error and then creating an alert using the error message, but as we discussed earlier we don’t want to do this for all errors. Instead we want a way to say in our code, “Yes, this error message can be exposed to end users and this is what the public message should say.” If we could do that, then we would be able to update our code to roughly be:

```
// Note: This code WILL NOT compile
func (d *Data) SetAlert(err error) {
    var msg string
    if err is public {
        // Public() would return the public error message
        msg = err.Public()
    } else {
```

⁶<https://dave.cheney.net/2016/04/07/constant-errors>

```
    msg = AlertMsgGeneric
}
d.Alert = &Alert{
    Level:  AlertLvlError,
    Message: msg,
}
}
```

In this case the code we just looked at won't compile, but writing out what we wish our code could do is helpful because it helps us determine how we could actually implement this.

If you look closely at the snippet above, the only thing that really differentiates a public error from a non-public error is that public errors need to have a `Public` method that is used to generate a public error message. As long as our error has that method, we can use it to generate the alert message.

If we want to check for this in our Go code we need to first define an interface, so let's start there.

```
type PublicError interface {
    error
    Public() string
}
```

The `error` type is actually an interface in Go, defining any type with an `Error` method that returns a string. That means we can embed that interface into our `PublicError` interface to ensure our public errors also work as regular errors.

The second line in the interface is what makes it different from the built-in error type. We also want anything that implements the `PublicError` interface to have a `Public` method that can be used to return an error message intended to be displayed to our end users.

With our interface defined, we can update our `SetAlert` method to take advantage of it and check if incoming errors are public or not. To do this we will

need to use a type assertion⁷, which is a way of letting us check if a piece of data implements an interface at run-time.

Box 13.4. What are type assertions?

If you are unfamiliar with type assertions I suggest you check out the Effective Go link in the footnotes, but for now the most important thing to understand is the arguments returned by a type assertion.

When you write a type assertion the code will typically look something like below.

```
publicError, ok := err.(PublicError)
```

The latter portion - **err.** (**PublicError**) - is the type assertion. We are checking to see if the **err** variable could be converted into the PublicError interface.

The type assertion has two return arguments; the first is the original variable, in this case **err**, after it is converted into the new type. The second argument is a boolean representing whether or not the conversion was successful.

It is possible for a type conversion to fail, so you need to check the boolean argument before using the first argument to ensure that it was converted successfully. For example, if our error does NOT implement the PublicError interface, **ok** would be false and we wouldn't be able to use the **publicError** variable.

We will start off by doing a type assertion in our code and checking to see if it was successful. If it was, we will use the **Public** method of the converted variable to set the message. Otherwise we will use a generic message because this error isn't meant to be public. We can also choose to log the error message at this point if we wanted.

⁷https://golang.org/doc/effective_go.html#blank_implements

```

import "log"

func (d *Data) SetAlert(err error) {
    var msg string
    if pErr, ok := err.(PublicError); ok {
        msg = pErr.Public()
    } else {
        log.Println(err)
        msg = AlertMsgGeneric
    }
    d.Alert = &Alert{
        Level: AlertLvlError,
        Message: msg,
    }
}

```

With that last change our SetAlert method is ready to differentiate between public errors (aka white-listed errors) and non-public errors, but we have one last problem to address; we haven't defined any public errors in our code yet!

Defining public errors requires us to define a new type that implements the PublicError interface. We will be doing that inside of our users model, so let's open up that source code.

```
$ atom models/users.go
```

We are going to start by declaring the new error type. The only real data we need to embed into our public errors is the message, which is a string, so we can just use **string** as the underlying data type. We then need to define the Error and Public methods so that our errors implement the PublicError type.

```

type modelError string

func (e modelError) Error() string {
    return string(e)
}

func (e modelError) Public() string {
    return string(e)
}

```

Before moving on, that Public method could be improved so let's take a stab at improving it. We know that all of our modelErrors start off with the string “models: “, and we likely don't want to display this to end users, so let's use the strings package's Replace function⁸ to replace that part of the string with the empty string.

```
s := strings.Replace(string(e), "models: ", "", 1)
```

*Note: The **1** at the end states that we only want this replacement to happen one time.*

The next thing we are going to do is capitalize the first letter of the error message so that rather than seeing “email is required” the user instead sees “Email is required”. The strings package offers a Title function⁹ that partially does what we need, but it attempts to capitalize every word in the string. We only want to capitalize the first word.

To address this we are going to first split our string into a slice of strings everywhere there is a space using the Split function¹⁰. For example, if we started with the string, “email is required” we would split it into a slice like:

```
[]string{"email", "is", "required"}
```

Doing this will allow us to access only the first word of the message and pass it into the Title function to capitalize it. We can then update its value in the slice of strings then use the Join function¹¹ to merge all of these strings back together reinserting the original spaces.

All together the code ends up being:

⁸<https://golang.org/pkg/strings/#Replace>

⁹<https://golang.org/pkg/strings/#Title>

¹⁰<https://golang.org/pkg/strings/#Split>

¹¹<https://golang.org/pkg/strings/#Join>

```
import "strings"

func (e modelError) Public() string {
    s := strings.Replace(string(e), "models: ", "", 1)
    split := strings.Split(s, " ")
    split[0] = strings.Title(split[0])
    return strings.Join(split, " ")
}
```

Now when we write an error message like “models: email is taken” it will be turned into “Email is taken” via the Public method.

We need to update all of our errors that are intended to be public, so we are going to replace all of our existing errors with ones of the new modelError type, and because we made the underlying type of our modelError a string we can define these as constants instead of variables.

```
const (
    // The comments for each error should remain unchanged.
    // They are excluded here for brevity.

    ErrNotFound modelError = "models: resource not found"

    ErrIDInvalid modelError = "models: ID provided was invalid"

    ErrPasswordIncorrect modelError = "models: incorrect " +
        "password provided"

    ErrPasswordTooShort modelError = "models: password must " +
        "be at least 8 characters long"

    ErrPasswordRequired modelError = "models: password is required"

    ErrEmailRequired modelError = "models: email address is " +
        "required"

    ErrEmailInvalid modelError = "models: email address is " +
        "not valid"

    ErrEmailTaken modelError = "models: email address is " +
        "already taken"

    ErrRememberRequired modelError = "models: remember token " +
        "is required"
```

```
ErrRememberTooShort modelError = "models: remember token " +
    "must be at least 32 bytes"
)
```

Now we are finally ready to update our users controller to use the new SetAlert method.

```
$ atom controllers/users.go
```

Find the two places where we create an alert and replace them with a call to the SetAlert method. We can also get rid of the code that logs the error message as our SetAlert method handles that now.

```
func (u *Users) Create(w http.ResponseWriter, r *http.Request) {
    var vd views.Data
    var form SignupForm
    if err := parseForm(r, &form); err != nil {
        vd.SetAlert(err)
        u.NewView.Render(w, vd)
        return
    }
    user := models.User{
        Name:      form.Name,
        Email:     form.Email,
        Password: form.Password,
    }
    if err := u.us.Create(&user); err != nil {
        vd.SetAlert(err)
        u.NewView.Render(w, vd)
        return
    }
    err := u.signIn(w, &user)
    if err != nil {
        http.Redirect(w, r, "/login", http.StatusFound)
        return
    }
    http.Redirect(w, r, "/cookietest", http.StatusFound)
}
```

Restart your server and verify that your new errors are working. You should be seeing specific error messages when your password is too short, an email

address is taken, or any other time a validation fails, and each of those should have a capitalized first letter and the “models: “ prefix should be removed.

Box 13.5. Interfaces can be defined anywhere

One thing worth noting with our PublicError interface is that our modelError type inside the models package never directly sees or interacts with the type. It is possible to implement an interface in Go without ever importing or using the interface directly. This is a very big difference from other languages like Java where you have to explicitly state “this class implements interface X”, but it is also a very powerful feature of Go.

Source code

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

Completed - book.usegolang.com/13.6

Changes - book.usegolang.com/13.6-diff

13.7 Handling login errors

Next on our agenda is to address how we are handling errors that occur when logging users in via the Login action inside our users controller. Open up your users controller source file and find the **Login** method.

```
$ atom controllers/users.go
```

The first error we need to address is one we saw in the Create method; whenever we fail to parse a form we need to render the login form again along with an alert explaining that something went wrong.

```
func (u *Users) Login(w http.ResponseWriter, r *http.Request) {
    var vd views.Data
    var form LoginForm
    if err := parseForm(r, &form); err != nil {
        vd.SetAlert(err)
        u.LoginView.Render(w, vd)
        return
    }

    // ... the rest of the code stays the same
}
```

After that we need to look at any errors returned when we attempt to Authenticate the user's email address and password. While it is tempting to just call the SetAlert method with any errors, we are still going to write a custom message for the ErrNotFound error, as this error isn't as clear as telling the user the email address they provided didn't match any users.

We can get rid of the ErrPasswordIncorrect error so our switch statement will get a bit smaller, and regardless of which error we see we want to render the LoginView again before returning.

```
user, err := u.us.Authenticate(form.Email, form.Password)
if err != nil {
    switch err {
    case models.ErrNotFound:
        vd.Alert = &views.Alert{
            Level: views.AlertLvlError,
            Message: "No user exists with that email address",
        }
    default:
        vd.SetAlert(err)
    }
}
```

```
    }
    u.LoginView.Render(w, vd)
    return
}
```

The last error we need to handle is the one returned by the signIn method. When we get an error here we want to render the login page again so the user can try logging in again.

```
err = u.signIn(w, user)
if err != nil {
    vd.SetAlert(err)
    u.LoginView.Render(w, vd)
    return
}
http.Redirect(w, r, "/cookietest", http.StatusFound)
```

With that we are done handling errors in our Login method. The last bit of code we are going to write is entirely optional, but is intended to make it easier to generate Alerts moving forward.

We saw in our switch statement earlier that when we want to write a custom error message we can't use the SetAlert method, and instead need to manually construct the Alert object. There is nothing wrong with this, but you can also write a method to simplify this if you prefer.

```
$ atom views/data.go
```

```
func (d *Data) AlertError(msg string) {
    d.Alert = &Alert{
        Level:  AlertLvlError,
        Message: msg,
    }
}
```

Now we can update the switch statement inside of our Login method to use the AlertError method instead of building the alert manually.

```
user, err := u.us.Authenticate(form.Email, form.Password)
if err != nil {
    switch err {
    case models.ErrNotFound:
        vd.AlertError("No user exists with that email address")
    default:
        vd.SetAlert(err)
    }
    u.LoginView.Render(w, vd)
    return
}
```

While this actually resulted in more code now, it will eventually end up saving us code later if we find ourselves writing custom alert messages in different controllers.

Source code

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

Completed - book.usegolang.com/13.7

Changes - book.usegolang.com/13.7-diff

13.8 Recovering from view rendering errors

The final class of errors we need to address is one we have been ignoring a bit; whenever we attempt to render our templates it is possible to run into an error. This could occur for any number of reasons, such as attempting to access data

that wasn't provided to our template or calling a custom function that ends up returning an error.

Unfortunately there isn't a great way to handle errors like this. As soon as we start writing to a ResponseWriter object the 200 status code is set and any data that has already been written to the ResponseWriter can't be taken back. That basically leaves us with two options:

1. Try to insert an error message in the middle of the page when our template execution errors.
2. Execute the entire template into a temporary buffer, and then copy it to the ResponseWriter once we know it has executed correctly.

The first option uses less memory and can be faster when shaving every little millisecond off matters, but for most web applications this isn't very important. People can't really perceive a few milliseconds of delay, and the memory footprint is relatively minor.

We will be going with the second option because it won't slow down our application noticeably, and it provides a much better user experience. Rather than ending up with a half-rendered page users will get a clear error page, and we will also have a chance to set the proper status code.

We will be working inside the views package to implement these changes.

```
$ atom views/view.go
```

The first thing we are going to change is the **Render** method's return type. We currently have it returning an error whenever we run into one executing our templates, leaving the error handling up to whoever called the Render method. Moving forward we are going to handle those errors inside the Render method, so we do not have to return an error.

Note: If you want to continue returning errors so you can track when errors do occur you can, but we won't be doing that in this course.

```
func (v *View) Render(w http.ResponseWriter, data interface{}) {
    // ... we will edit the internals of this method shortly
}
```

After that we need to skip down to the return statement where we execute our template. We are going to start off by creating a **Buffer**¹², a type that comes from the **bytes** package.

```
var buf bytes.Buffer
```

Buffer is a type that implements the Read and Write interface from the **io** package. There are many use cases for Buffer, but in this situation we are going to be using the buffer as a temporary location to execute our templates into. Once we have confirmed that there weren't any errors executing the template we will have the entire executed template stored inside of our Buffer, and we can copy it over to the ResponseWriter.

Remember: We are using a buffer because writing any data to ResponseWriter will result in a 200 status code and we can't undo that write. By writing to a buffer first we can confirm that the entire template executes before we start writing any data to the ResponseWriter.

```
var buf bytes.Buffer
err := v.Template.ExecuteTemplate(&buf, v.Layout, data)
if err != nil {
    // We will handle this in a moment
}
// If we get here that means our template executed correctly
// and we can copy the buffer to the ResponseWriter
io.Copy(w, &buf)
```

¹²<https://golang.org/pkg/bytes/#Buffer>

On the last line of our code we use the `Copy`¹³ function provided by the `io` package. This function takes in a Writer and a Reader, and then copies all the data in the Reader over to the Writer. In our code we are copying all of the data from the Buffer over to the ResponseWriter.

The last thing we need to implement is what to do when an error occurs executing our template. In this situation we are going to utilize the `Error` function provided by the `net/http` package, and we will set the message to be a generic “Something went wrong” message, along with a 500 HTTP status code signaling that we had an internal server error.

Putting that all together, our Render method should look like the one shown below.

```
import (
    "bytes"
    "io"
)

func (v *View) Render(w http.ResponseWriter, data interface{}) {
    w.Header().Set("Content-Type", "text/html")
    switch data.(type) {
    case Data:
        // do nothing
    default:
        data = Data{
            Yield: data,
        }
    }
    var buf bytes.Buffer
    err := v.Template.ExecuteTemplate(&buf, v.Layout, data)
    if err != nil {
        http.Error(w, "Something went wrong. If the problem " +
            "persists, please email support@lenslocked.com",
            http.StatusInternalServerError)
        return
    }
    io.Copy(w, &buf)
}
```

After finishing the Render method we need to update any code that expected it

¹³<https://golang.org/pkg/io/#Copy>

to return an error. We will start off by updating the ServeHTTP method.

```
func (v *View) ServeHTTP(w http.ResponseWriter, r *http.Request) {
    v.Render(w, nil)
}
```

Next is the New method in the users controller.

```
$ atom controllers/users.go
```

```
func (u *Users) New(w http.ResponseWriter, r *http.Request) {
    u.NewView.Render(w, nil)
}
```

Rebuild and restart your server to verify that everything is working again. We are officially done rendering errors for now, and we are ready to move on to much more interesting things!

In the next chapter we will learn how to add the Gallery resource, allow users to create new galleries, limit access to each individual gallery based on the currently logged in user, how middleware works, and much more.

Source code

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

Completed - book.usegolang.com/13.8

Changes - book.usegolang.com/13.8-diff

Chapter 14

Creating the gallery resource

In this chapter we are going to take everything we have learned so far and start putting it to use to build out the Gallery resource. We will need to create a new model, a new controller, new actions, and new views, just like we did when creating the user resource.

In our application galleries are going to serve as a container resource. They will have a little bit of data on their own - such as the gallery name - but for the most part they are used to organize images and dictate permissions.

When a user goes to create a gallery, they will first tell us the name of the gallery, and then they will start adding images to the gallery. We will start by only supporting image uploads, but our code could be extended to also support Dropbox syncing and much more.

After a user has added images to a gallery they will be able to share a URL to that gallery which others can visit in their browser and view the gallery. Only the author will be able to edit the gallery, but everyone will be able to view it.

Let's get started by defining the data we are going to store in our database.

14.1 Defining the gallery model

We will be creating our gallery model in a new source file inside the models package.

```
$ atom models/galleries.go
```

We can start off by defining the package.

```
package models
```

Our gallery resource may change over time, but for now it needs the following data:

1. Normal model information, like an ID, when it was created, updated, etc.
2. A way of tracking which user owns the gallery. This will always be set to the user who creates the gallery, so we won't need to let the end user edit this, but we will need to store it on the model.
3. A title that is editable by the owner. This will typically be something like "Beach day with the kids!"
4. We will eventually need a way to link images to our gallery.

The normal model information will be taken care of using GORM's **Model** type, so we don't need to discuss the first piece of data much.

```
import "github.com/jinzhu/gorm"

type Gallery struct {
    gorm.Model
}
```

After that we need a way to track which user owns the gallery. The easiest way to do this is to store the ID of whatever user owns the gallery in the model. That way we can easily lookup the user with their ID, and we can also find all of a specific user's galleries by running a query like, "Select all of the galleries where the user_id is 123."

User IDs are always unsigned integers (`uint`), so we will name that field `UserID` and give it the type `uint`. We want to require this field, and as we discussed earlier we will likely be querying for galleries based on the user ID, so we will add the "not_null" and "index" tags to inform GORM that we want to require this field to have a value and we want to index the field to make our queries by UserID faster.

```
type Gallery struct {
    gorm.Model
    UserID uint `gorm:"not_null;index"`
}
```

The title field can be stored in a string field, and the only special rule we want here is to require the field to have a value. A gallery wouldn't be very useful without at least having a title.

```
type Gallery struct {
    gorm.Model
    UserID uint `gorm:"not_null;index"`
    Title  string `gorm:"not_null"`
}
```

While we will eventually want a way to link images to our gallery, we don't have any image resources yet so we are going to skip that for now. Once our users can create galleries with a Title and share them we will look at adding images to our galleries.

Source code

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

Completed - book.usegolang.com/14.1

Changes - book.usegolang.com/14.1-diff

14.2 Introducing the GalleryService

We need a way to interact with galleries, so our next step is to create a GalleryService. This will be an interface that exposes a set of functions which can be used to interact with our galleries data store.

Open up the galleries source file inside the models package.

```
$ atom models/galleries.go
```

We don't have any specific methods we need to offer in our gallery service just yet, but we know that we will be creating galleries shortly so we can start off with just the Create method and stub it out for the time being.

```
type GalleryService interface {
    GalleryDB
}

type GalleryDB interface {
    Create(gallery *Gallery) error
}

type galleryGorm struct {
    db *gorm.DB
}

func (gg *galleryGorm) Create(gallery *Gallery) error {
```

```
// TODO: Implement this later  
return nil  
}
```

We are going to work a bit more on implementing the GalleryService in a bit, but first we are going to look at how to simplify construction of multiple services in the rest of our code.

Source code

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

Completed - book.usegolang.com/14.2

Changes - book.usegolang.com/14.2-diff

14.3 Constructing many services

Now that we have two services, it is time to look at how we construct our services.

We could continue to do this as we are in our main package; that is, we could create each service individually using functions like NewUserService and NewGalleryService, but this approach introduces a problem. Doing this with our current code would result in a new database connection being created for each service.

We don't want to recreate our database connection for each of our services, but we also don't want to manage database connections outside of our models package. One of the primary benefits to isolating all of our database interactions

inside the models package is that we don't have to think about database logic outside of the package.

To solve this problem we are going to introduce a new function - the NewServices function - that will be used to construct all of our services and return them.

We will add this code to a new source file inside models package.

```
$ atom models/services.go
```

When writing our NewServices function we could use multiple return values and return multiple services that way. For example:

```
func demo() (UserService, GalleryService, error)
```

Note: This is for illustration purposes only. Do not code it.

As we introduce more services this approach would become tiresome, so instead we are going to introduce a single type that contains all of our services, and then we will return a single copy of that type in our NewServices function.

```
package models

func NewServices(connectionInfo string) (*Services, error) {
    // TODO: Implement this...
}

type Services struct {
    Gallery GalleryService
    User    UserService
}
```

With the Services type we can easily construction all of our services and return them in a single object.

Next we need to implement the NewServices function. In it we will open a database connection, check for errors, set our log mode to true, and then use that database connection to construct our individual services.

```
import "github.com/jinzhu/gorm"

func NewServices(connectionInfo string) (*Services, error) {
    db, err := gorm.Open("postgres", connectionInfo)
    if err != nil {
        return nil, err
    }
    db.LogMode(true)
    // And next we need to construct services, but
    // we can't construct the UserService yet.
    return &Services{
        // User: ??? - we will get to this shortly,
        Gallery: &galleryGorm{},
    }, nil
}
```

In the past we have constructed our UserService using the NewUserService function which opens up a new database connection, but moving forward we want to reuse the database connection that our NewServices function created. That means we need to update our NewUserService function to accept a preexisting database connection rather than a string used to open a new connection.

```
$ atom models/users.go
```

```
func NewUserService(db *gorm.DB) UserService {
    ug := &userGorm{db}
    hmac := hash.NewHMAC(hmacSecretKey)
    uv := newUserValidator(ug, hmac)
    return &userService{
        UserDB: uv,
    }
}
```

While changing the NewUserService function we were also able to update the return types, opting to only return a single value now that we don't have to worry about errors opening a new database connection.

We can also remove the newUserGorm function, as we are no longer using it.

```
// Find and delete this function.
func newUserGorm(connectionInfo string) (*userGorm, error) {
    // ...
}
```

After changing our NewUserService function we can go back to our NewServices function and update it to use the updated code.

```
$ atom models/services.go
```

```
func NewServices(connectionInfo string) (*Services, error) {
    db, err := gorm.Open("postgres", connectionInfo)
    if err != nil {
        return nil, err
    }
    db.LogMode(true)
    return &Services{
        User:    NewUserService(db),
        Gallery: &galleryGorm{},
    }, nil
}
```

Our models package is now updated, but we need to update our **main.go** source file to account for the changes we made.

```
$ atom main.go
```

We will be updating the first half of the **main** function.

```
func main() {
    psqlInfo := fmt.Sprintf("host=%s port=%d user=%s "+
        "password=%s dbname=%s sslmode=disable",
        host, port, user, password, dbname)
```

```
services, err := models.NewServices(psqlInfo)
if err != nil {
    panic(err)
}
// TODO: Simplify this
defer services.User.Close()
services.User.AutoMigrate()

staticC := controllers.NewStatic()
usersC := controllers.NewUsers(services.User)

// ... the routing code remains unchanged
}
```

Our code should now compile and run much like it did before, but rather than constructing each service individually we are constructing them all with a single function call. We are still manually auto-migrating and closing the user service, but we will address that in the next section.

To recap: In this section we wrote code that is intended to make it easier to construct all of our services exported by the `models` package. At this point our code won't functionally be much different than it was before, but this will make it easier to use the `models` package in the rest of our code.

Source code

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

Completed - book.usegolang.com/14.3

Changes - book.usegolang.com/14.3-diff

14.4 Closing and migrating all models

When we updated our main function in the last section we ended up writing some code like the code shown below.

```
defer services.User.Close()
services.User.AutoMigrate()
```

With this code we are telling our program to close the user service just before the main function exits, and we are immediately auto-migrating our users table.

As we add more models, we aren't going to want to write this for every resource. That is, we won't want to have:

```
defer services.User.Close()
defer services.Gallery.Close()
defer services.Image.Close()
services.User.AutoMigrate()
services.Gallery.AutoMigrate()
services.Image.AutoMigrate()
```

Wouldn't it be much easier if we could instead do this all with only two lines of code?

```
defer services.Close()
services.AutoMigrate()
```

Not only is this code easier to read, write, and maintain, but it also can help us avoid potential bugs.

With the original code it is unclear what happens if we close a single service but continue using other services. Will that end up causing an error because the database connection is closed as well, or will the Close method leave it open? If the connection is left open, when will it eventually be closed?

We are going to update our code to provide a single function used close all of our services, and another single function used to migrate all of our models. While we are at it, we will also add a third method that can be used to perform a destructive reset on all of our database tables much like our current `DestructiveReset` method offered by the `UserService`.

All of these methods are going to be added to the `Services` type inside the `models` package.

```
$ atom models/services.go
```

We will start with the `Close` method, in which all we need to do is close the database connection. In order to close the database connection we need to have access to it. The easiest way to provide access is to introduce a new field to the `Services` type that stores the `gorm.DB` connection after we open it.

```
func NewServices(connectionInfo string) (*Services, error) {
    db, err := gorm.Open("postgres", connectionInfo)
    if err != nil {
        return nil, err
    }
    db.LogMode(true)
    return &Services{
        User:    NewUserService(db),
        Gallery: &galleryGorm{},
        db:      db,
    }, nil
}

type Services struct {
    Gallery GalleryService
    User    UserService
    db      *gorm.DB
}
```

After that we can access the database connection and close it inside our new `Close` method.

```
// Closes the database connection
func (s *Services) Close() error {
    return s.db.Close()
}
```

Next up is the AutoMigrate method, which will be similar to the AutoMigrate method we wrote as part of the userGorm type, but will attempt to automatically migrate all of our models.

```
// AutoMigrate will attempt to automatically migrate all tables
func (s *Services) AutoMigrate() error {
    return s.db.AutoMigrate(&User{}, &Gallery{}).Error
}
```

Finally we have the DestructiveReset method, which will try to drop every database table that exists and then reconstruct them with the AutoMigrate method.

```
// DestructiveReset drops all tables and rebuilds them
func (s *Services) DestructiveReset() error {
    err := s.db.DropTableIfExists(&User{}, &Gallery{}).Error
    if err != nil {
        return err
    }
    return s.AutoMigrate()
}
```

Now that we have a way to close and migrate all of services at once we can update our `main` function to use the new methods.

```
$ atom main.go
```

```
func main() {
    psqlInfo := fmt.Sprintf("host=%s port=%d user=%s "+
        "password=%s dbname=%s sslmode=disable",
```

```
host, port, user, password, dbname)
services, err := models.NewServices(sqlInfo)
if err != nil {
    panic(err)
}
defer services.Close()
services.AutoMigrate()

// ... The rest remains unchanged
}
```

And finally we can remove any unused code we have left, such as the Close, AutoMigrate, and DestructiveReset methods we wrote as part of the UserService. Those methods are no longer necessary there, and removing them will help simplify the UserService a bit.

```
$ atom models/users.go
```

First we will remove the methods from the UserDB interface.

```
type UserDB interface {
    // Methods for querying for single users
    ByID(id uint) (*User, error)
    ByEmail(email string) (*User, error)
    ByRemember(token string) (*User, error)

    // Methods for altering users
    Create(user *User) error
    Update(user *User) error
    Delete(id uint) error
}
```

After that we need to find the three method implementations associated with the userGorm type and delete them.

```
// Delete these three functions
func (ug *userGorm) Close() error { ... }
func (ug *userGorm) DestructiveReset() error { ... }
func (ug *userGorm) AutoMigrate() error { ... }
```

Restart your server and you should see logs indicating that the galleries table is being created by the call to AutoMigrate.

```
# Your logs will be slightly different, but should have
# messages starting similarly to the ones below.
CREATE TABLE "galleries" (...)

CREATE INDEX idx_galleries_deleted_at ...
CREATE INDEX idx_galleries_user_id ...
```

Source code

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

Completed - book.usegolang.com/14.4

Changes - book.usegolang.com/14.4-diff

14.5 Creating new galleries

Writing the code necessary to create a new gallery involves touching a lot of different areas of our application. We need to write code in our models package to handle writing data to the database, validations for that data, an HTML template used to render the new gallery form, and we will need to add a new controller and actions to render and then parse the form once it is submitted.

Rather than trying to do this all at once, we are going to break it into four steps:

1. The first thing we are going to do is write a working GalleryService implementation that allows us to insert galleries into our database.
2. Once we have a working GalleryService implementation we will focus on creating the galleries controller and a view to render the new gallery form.
3. After that we will write the code used to parse the new gallery form and use that data to create a new gallery.
4. Finally, we will add validations to our gallery service so that we can ensure we are only saving valid data to the database.

14.5.1 Implementing the gallery service

Our layers used in the gallery service are going to match the ones used in the user service. We will have a galleryService that implements the GalleryService interface, a galleryValidator for validations, and a galleryGorm for interacting with our database.

We already have the galleryGorm, so let's start off by adding the galleryService and galleryValidator.

```
$ atom models/galleries.go
```

```
type galleryService struct {
    GalleryDB
}

type galleryValidator struct {
    GalleryDB
}
```

After that we are going to ensure that our galleryGorm always implements our GalleryDB interface.

```
var _ GalleryDB = &galleryGorm{}
```

We haven't implemented the galleryGorm's Create method yet either, so we can add that next. This source code is nearly identical to the code used in userGorm's Create method because GORM uses the resource we are saving to determine which table to insert the record into for us.

```
func (gg *galleryGorm) Create(gallery *Gallery) error {
    return gg.db.Create(gallery).Error
}
```

It is also handy to have a way to construct a GalleryService with each of these layers already in place, so we are going to add a NewGalleryService function as well.

```
func NewGalleryService(db *gorm.DB) GalleryService {
    return &galleryService{
        GalleryDB: &galleryValidator{
            GalleryDB: &galleryGorm{
                db: db,
            },
        },
    }
}
```

We can then update our NewServices function to use the NewGalleryService function.

```
$ atom models/services.go
```

```
func NewServices(connectionInfo string) (*Services, error) {
    db, err := gorm.Open("postgres", connectionInfo)
    if err != nil {
```

```
    return nil, err
}
db.LogMode(true)
return &Services{
    User:    NewUserService(db),
    Gallery: NewGalleryService(db),
    db:      db,
}, nil
}
```

With that we have a working gallery service and can insert new galleries into our database's galleries table with the Create method we implemented. We don't validate the data yet, but we can wait to do that until we implement the controller and actions needed to render and process a new gallery form.

Source code

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

Completed - book.usegolang.com/14.5.1

Changes - book.usegolang.com/14.5.1-diff

14.5.2 The galleries controller

In this section we are going to introduce the galleries controller, along with a **New** view used to render the new gallery form. We will look at the source code to process the form in the next section - for now we just want to render the form and get our controller setup.

We are going to start by creating the template for our view, which means we need to create a directory to store our gallery views and then create a template file there.

```
$ mkdir views/galleries
$ atom views/galleries/new.gohtml
```

Inside the template we are going to define a “yield” template like we normally do, and in it we will render the form much like we did for the sign up page.

```
 {{define "yield"}}
<div class="row">
  <div class="col-md-6 col-md-offset-3">
    <div class="panel panel-primary">
      <div class="panel-heading">
        <h3 class="panel-title">Create a gallery</h3>
      </div>
      <div class="panel-body">
        {{template "galleryForm"}}
      </div>
    </div>
  </div>
{{end}}

 {{define "galleryForm"}}
<form action="/galleries" method="POST">
  <div class="form-group">
    <label for="title">Title</label>
    <input type="text" name="title" class="form-control" id="title" placeholder="What is the title?">
  </div>
  <button type="submit" class="btn btn-primary">Create</button>
</form>
{{end}}
```

Our form needs a single input field for the title of the gallery, and we will use the ‘title’ name for that field. Aside from that the rest of the data about our gallery will be derived from the currently logged in user (the UserID field) or set automatically by GORM and the database (ID, CreatedAt, etc).

When we submit the form we will send it to the `/galleries` path which we will setup in our router in the next section, and we will do this via the POST method.

Next we need a controller to contain this view and help us render it. We will place the galleries controller in its own source file to keep things organized.

```
$ atom controllers/galleries.go
```

We'll start with the Galleries type that will contain all of our views and handler functions associated with the gallery resource. The galleries type is going to eventually need access to the GalleryService to interact with our database, and it will need a view for the new gallery page we just created the template for.

```
package controllers

import (
    "lenslocked.com/models"
    "lenslocked.com/views"
)

type Galleries struct {
    New *views.View
    gs  models.GalleryService
}
```

Next we will write a NewGalleries function to simplify constructing this controller. In this we will require the caller to pass in a GalleryService for us to use, but we will construct the New view on our own using the template we created at the start of this section.

```
func NewGalleries(gs models.GalleryService) *Galleries {
    return &Galleries{
        New: views.NewView("bootstrap", "galleries/new"),
        gs:   gs,
    }
}
```

Finally we need to update our **main** function to create a galleries controller and then to map the “/galleries/new” path to the new galleries view.

```
$ atom main.go
```

```
func main() {
    // ... our existing code for setting up services
    // remains here

    staticC := controllers.NewStatic()
    usersC := controllers.NewUsers(services.User)
    galleriesC := controllers.NewGalleries(services.Gallery)

    r := mux.NewRouter()
    // ... our existing routes are here

    // Gallery routes
    r.Handle("/galleries/new", galleriesC.New).Methods("GET")

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

Restart your web application and navigate to our new page - localhost:3000/galleries/new. You should see a form to create a new gallery with only the title field.

Source code

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

Completed - book.usegolang.com/14.5.2

Changes - book.usegolang.com/14.5.2-diff

14.5.3 Processing the new gallery form

Next on our agenda is processing the new gallery form and using the data to insert a record into our database. To do this we are going to add the Create

[H]

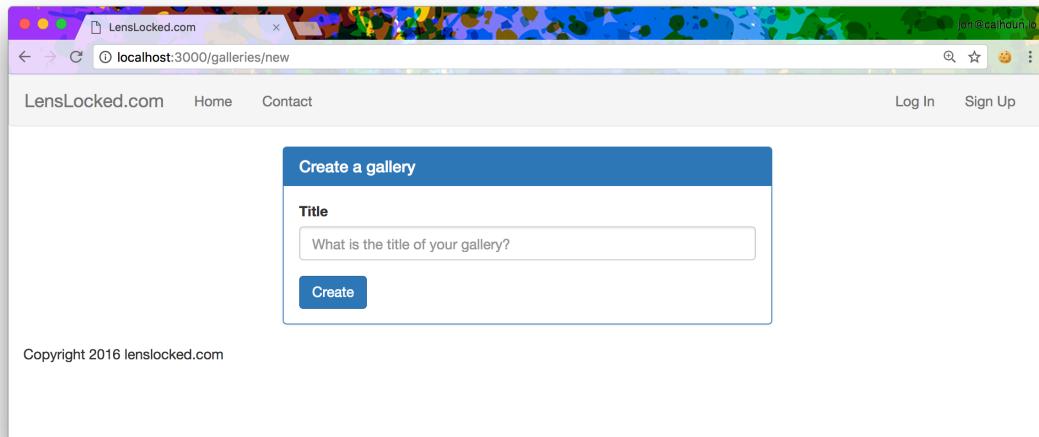


Figure 14.1: Our new gallery form

method to our galleries controller.

```
$ atom controllers/galleries.go
```

We'll start by stubbing out the create method and adding a few imports we will be using.

```
// Add these two imports
import (
    "fmt"
    "net/http"
)

// POST /galleries
func (g *Galleries) Create(w http.ResponseWriter, r *http.Request) {
    // TODO: Implement this
}
```

The next thing we need to do is parse the incoming form data. To do this we are

going to create a new struct type similar to our `SignupForm`, but this one will only parse the Title from the form.

```
type GalleryForm struct {
    Title string `schema:"title"`
}
```

Now we can use the `parseForm` function to parse the data submitted in the web request into a `GalleryForm` instance. If there is an error parsing the form we will use the `views.Data` type we created to set an alert and render the new gallery form and return. Otherwise our code will proceed.

```
func (g *Galleries) Create(w http.ResponseWriter, r *http.Request) {
    var vd views.Data
    var form GalleryForm
    if err := parseForm(r, &form); err != nil {
        vd.SetAlert(err)
        g.New.Render(w, vd)
        return
    }
    // TODO: Finish implementing this
}
```

With our form data available in the `form` variable we can then construct a new Gallery.

```
gallery := models.Gallery{
    Title: form.Title,
}
```

After that we will call the `GalleryService`'s `Create` method passing in the `gallery` and checking for errors. If there is an error we need to render the `New` view again along with an alert created from the error. Otherwise we will print out the gallery that we successfully created.

```
func (g *Galleries) Create(w http.ResponseWriter, r *http.Request) {
    var vd views.Data
    var form GalleryForm
    if err := parseForm(r, &form); err != nil {
        vd.SetAlert(err)
        g.New.Render(w, vd)
        return
    }
    gallery := models.Gallery{
        Title: form.Title,
    }
    if err := g.gs.Create(&gallery); err != nil {
        vd.SetAlert(err)
        g.New.Render(w, vd)
        return
    }
    fmt.Fprintln(w, gallery)
}
```

Note: One piece of code that is missing here is the code necessary to fill in the UserID field of our gallery. We will eventually fill this in with the currently logged in user's ID, but for now we will leave it blank.

The last thing we need to do is update our router. We will process the form at the `/galleries` path when the HTTP method is a POST.

```
$ atom main.go
```

```
func main() {
    // ... most of our code stays the same

    // Gallery routes
    r.Handle("/galleries/new", galleriesC.New).Methods("GET")
    r.HandleFunc("/galleries", galleriesC.Create).Methods("POST")

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

While we don't have any validations in place, we can now create new galleries

using our form. Restart your server and head to the new galleries page to test it out - localhost:3000/galleries/new.

Source code

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

Completed - book.usegolang.com/14.5.3

Changes - book.usegolang.com/14.5.3-diff

14.5.4 Validating galleries

Just like we validated users, we also need to validate our galleries before we store them in our database. Specifically, we want to validate that a UserID is always set so we know which user created each gallery, and we will also want to require a title.

We'll start by defining a type for our gallery validation functions along with a function to run them, just like we did with the user validations.

```
$ atom models/galleries.go
```

```
type galleryValFn func(*Gallery) error

func runGalleryValFns(gallery *Gallery, fns ...galleryValFn) error {
    for _, fn := range fns {
        if err := fn(gallery); err != nil {
            return err
        }
    }
    return nil
}
```

Now we can get started writing validation functions for our galleries, starting with one to ensure a UserID is set.

```
const (
    ErrUserIDRequired modelError = "models: user ID is required"
)

func (gv *galleryValidator) userIDRequired(g *Gallery) error {
    if g.UserID <= 0 {
        return ErrUserIDRequired
    }
    return nil
}
```

Note: Errors like the one we just created likely shouldn't be public as the end user has no control over what UserID is set. If you would like to learn how to create a private error type similar to our modelError type check out Appendix 18.4.

Next up is the validation to ensure the title is provided.

```
const (
    ErrTitleRequired modelError = "models: title is required"
)

func (gv *galleryValidator) titleRequired(g *Gallery) error {
    if g.Title == "" {
        return ErrTitleRequired
    }
    return nil
}
```

And finally we need to implement the Create method on the galleryValidator and have it run the two validations we just wrote.

```
func (gv *galleryValidator) Create(gallery *Gallery) error {
    err := runGalleryValFns(gallery,
        gv.userIDRequired,
        gv.titleRequired)
```

```
if err != nil {
    return err
}
return gv.GalleryDB.Create(gallery)
}
```

If we restart our server and try to create a gallery at this point we will see a “User ID is required” error. Until we implement the code to look up the current user’s ID and set it in the UserID field we won’t be able to create galleries with our form.

Source code

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

Completed - book.usegolang.com/14.5.4

Changes - book.usegolang.com/14.5.4-diff

14.6 Requiring users via middleware

Whenever someone submits the new gallery form we need to ensure that they are logged in so we can use their UserID and give them ownership of the gallery.

While we could take the code we wrote in our CookieTest that looks up the currently logged in user and add it to every action that requires a user to be logged in, this ends up being very tedious to do in practice. Even if we moved that code to a shared function, we still would end up writing the same three or four lines of code in every handler function that requires a user to be logged in.

Instead, it would be much nicer if our application were setup so that our handler functions were only run after validating the user is logged in. If that were the

[H]

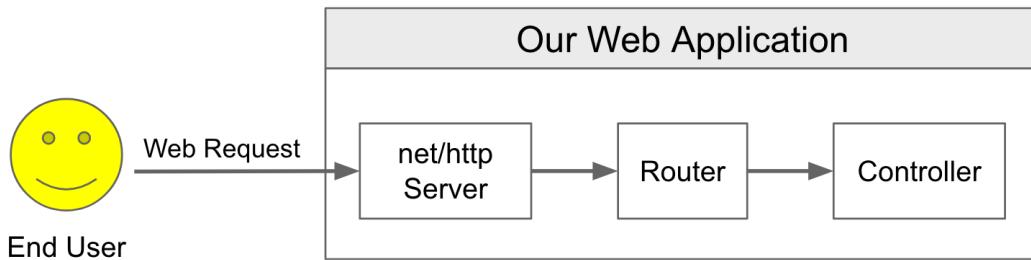


Figure 14.2: A web request without middleware

case we wouldn't need to think about checking if a user is logged in; instead our handlers could just assume that the user was logged in much like our userGorm can assume the data it is working with has already been validated. Middleware, along with the context package, will allow us to set our application up this way.

In this section we will learn to create reusable middleware that can be used to verify a user is logged and redirect them to the login page if they are not. We will then look at how to use the context package to store request-specific information, such as the currently logged in user's ID, and then attach it to the incoming web request so that we can access it later in our handler functions.

14.6.1 Creating our first middleware

Middleware is essentially a fancy name for inserting code in between different parts of our application. Whenever a user makes a request to our current application, the flow of logic looks roughly like [Figure 14.2](#).

The request is first processed by the `net/http` package. After the http package does its work it then calls our router's `ServerHTTP` method which will handle looking at what page the user is trying to request and deciding what code to call

[H]

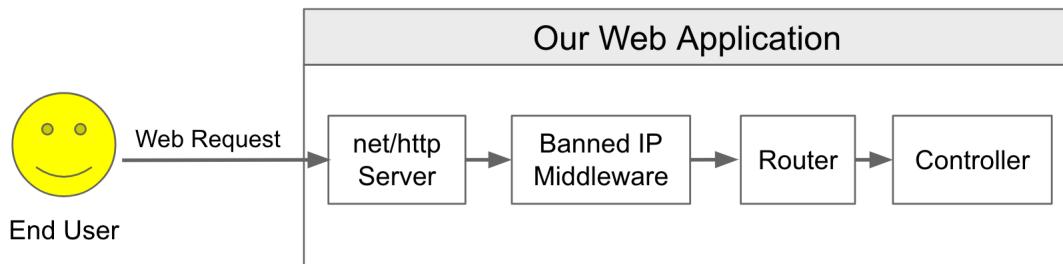


Figure 14.3: Banned IP middleware

next. For example, if the user is visiting the `/contact` path our router will call the Contact method provided by the static controller we created earlier.

```
r.HandleFunc("/contact", staticC.Contact).Methods("GET")
```

Middleware are pieces of code that get inserted in between the net/http package and our router, or between our router and controllers. For example, we might add a middleware before our router is called to block users with a banned IP address.

Our middleware could then determine if a user is banned or not by looking at their IP address. If the user is banned, the middleware would return an error and not bother calling our router code. Otherwise if the user is not banned it would call our router's ServeHTTP method and our code would continue handling the web request just like it does now.

```

if <user is banned> {
    http.Error(...)
    return
} else {
    // nextHandler = router.ServeHTTP - an argument
    // passed into this code that is an http.HandlerFunc
  
```

```
// function that we can call
nextHandler(w, r)
}
```

Using middleware is another example of interface chaining, something we discussed in [Box 12.2](#). That is, we are creating a series of function calls and any of them can decide whether to call the next function in the chain or terminate early.

```
net/http -> bannedIPMiddleware -> router -> controller
^
Our code here could return an error before ever
proceeding to the router.
```

Designing your code this way can have many benefits, but the primary one we are going to focus on now is that it simplifies our code. Imagine we wrote a middleware that would redirect a user to the login page if they were not signed in, otherwise it would call whatever handler function is next in its chain.

```
router -> requireUserMiddleware -> controllerAction
```

Now we wouldn't need to write code individually for each of our controller methods to check to see if a user is logged in. Instead we would apply this middleware to the actions that require a user to be logged in, and our controller code wouldn't change at all.

That is what we are going to be doing in this section. Writing our first middleware that can check to see if a user is logged in and then reacts accordingly.

We will start by creating a new package - the middleware package - and in it we will be adding a `RequireUser` middleware.

```
$ mkdir middleware
$ atom middleware/require_user.go
```

In order to check if a user is logged in we need access to the UserService, so we will start of by creating a struct type that has a UserService field.

```
package middleware

import (
    "lenslocked.com/models"
)

type RequireUser struct {
    models.UserService
}
```

Our next step is to write the code that redirects a user if they aren't logged in, but otherwise renders the next handler function. To do this we are going to once again need to create a closure, which we discussed previously in [Box 12.8](#).

We will be using a closure in order to take in an argument - the next http handler to call if a user is signed in - and then return a function that matches the http.HandlerFunc definition. That is, we want to return a function that accepts a ResponseWriter and Request as its two arguments.

By writing our functions to match this definition we know that they will continue to work with the standard library's net/http package as well as our gorilla/mux's Router type.

We will start by stubbing out our function that creates the closure.

```
// ApplyFn will return an http.HandlerFunc that will
// check to see if a user is logged in and then either
// call next(w, r) if they are, or redirect them to the
// login page if they are not.
func (mw *RequireUser) ApplyFn(next http.HandlerFunc) http.HandlerFunc {
    // We want to return a dynamically created
```

```
// func(http.ResponseWriter, *http.Request)
// but we also need to convert it into an
// http.HandlerFunc
return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
    // TODO: Check if a user is logged in.
    // If so, call next(w, r)
    // If not, http.Redirect to "/login"
})
}
```

As we implement the `ApplyFn` fully, we will mostly be taking code we learned about as we created the `CookieTest` method on the users controller and moving it here. For instance, we will start by getting the `remember_token` cookie from the `Request` object much like we did in the `CookieTest` method.

```
cookie, err := r.Cookie("remember_token")
if err != nil {
    http.Redirect(w, r, "/login", http.StatusFound)
    return
}
```

In this case we will redirect the user to the login page if there is an error, but otherwise this code is the same.

Next we are going to use the `UserService` to look up a user via the remember token stored on the cookie we just retrieved. As long as the user has a valid session this should return a user, otherwise we will get an error indicating that the user needs redirected to the login page.

```
user, err := mw.UserService.ByRemember(cookie.Value)
if err != nil {
    http.Redirect(w, r, "/login", http.StatusFound)
    return
}
fmt.Println("User found: ", user)
```

Finally, we will call the `next` function provided to our `ApplyFn` method once

we have verified that the user is indeed logged in. Putting it all together we get the source code below.

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

    "lenslocked.com/models"
)

func (mw *RequireUser) ApplyFn(next http.HandlerFunc) http.HandlerFunc {
    return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
        cookie, err := r.Cookie("remember_token")
        if err != nil {
            http.Redirect(w, r, "/login", http.StatusFound)
            return
        }
        user, err := mw.UserService.ByRemember(cookie.Value)
        if err != nil {
            http.Redirect(w, r, "/login", http.StatusFound)
            return
        }
        fmt.Println("User found: ", user)
        next(w, r)
    })
}
```

We are also going to want to apply our middleware to http.Handler interfaces, which we can do by passing their ServeHTTP method into our ApplyFn function.

```
func (mw *RequireUser) Apply(next http.Handler) http.HandlerFunc {
    return mw.ApplyFn(next.ServeHTTP)
}
```

Next up is applying our middleware. To do this we need to open up the `main.go` source file construct a RequireUser middleware.

```
$ atom main.go
```

```
// Add the middleware package to our imports
import (
    "lenslocked.com/middleware"
)

func main() {
    // ... our services and controllers setup code remains
    // unchanged here

    requireUserMw := middleware.RequireUser{
        UserService: services.User,
    }

    // ... our router code remains unchanged here
}
```

We want to require a user to be logged in whenever they visit both the new gallery form as well as when they submit it. To do this, we call either the `Apply` or `ApplyFn` method of our middleware, passing in our controller action as the only argument. By applying our middleware we get a new `http.HandlerFunc` back that will only run our original handler if the user is logged in.

```
// galleriesC.New is an http.Handler, so we use Apply
newGallery := requireUserMw.Apply(galleriesC.New)
// galleriesC.Create is an http.HandlerFunc, so we use ApplyFn
createGallery := requireUserMw.ApplyFn(galleriesC.Create)
```

Note: Whether we use `Apply` or `ApplyFn` depends on whether our argument is an `http.Handler` or an `http.HandlerFunc`.

We can then pass the results of the `Apply` (or `ApplyFn`) function into our router where we were previously passed the original.

```
r.Handle("/galleries/new", newGallery).Methods("GET")
r.Handle("/galleries", createGallery).Methods("POST")
```

Note: In the final code we don't store the results of the Apply function in an intermediate variable, but the result is the same.

Now when we visit the new gallery page or submit the form our middleware will ensure that a user is logged in.

If we want to test that our middleware works we would first delete any cookies associated with localhost in our browser settings, and then try to visit the new gallery page. We will instead be redirected to the login page.

Source code

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

Completed - book.usegolang.com/14.6.1

Changes - book.usegolang.com/14.6.1-diff

14.6.2 Storing request-scoped data with context

While our middleware will currently verify that a user is logged in, it doesn't do anything to help us figure out which user is logged in. After our middleware looks up the user via their remember token the results are discarded. It would be nice if we could reuse that data and avoid having to look the same user up from the database multiple times per request. The context package, which was added to the http.Request object in Go 1.7, provides us with a way to do just that.

Box 14.1. Context controversy

Storing data inside a context, especially in middleware or during a web request, can be controversial. There are some potential issues with type safety being lost, and it can make testing your code both easier in some ways, but harder in others.

For example, it can be easier to test your handlers because you no longer have to focus on generating remember tokens and testing all the user lookup code that is handled by our middleware. Instead we could attach a single user to the context and our handlers should work as intended.

It can be harder to test because this data expectation isn't always clear. When we define a method like:

```
func doStuff(user User, years int)
```

We are making it clear what data we expect. By putting data inside the context package we are masking some of those requirements.

In practice I find that using context values this way is almost always a net-win, and there are ways to combat both the type safety loss (which we will see in this section), along with ways to combat the hidden complexity which are discussed at length at the link below.

book.usegolang.com/context

You can also read/watch a bit more about the context package in general at the following two resources:

- blog.golang.org/context - the official Go blog
- [youtube.com/watch?v=LSzR0VEraWw](https://www.youtube.com/watch?v=LSzR0VEraWw) - Francesc Campoy's justforfunc episode on the context package

The context package has existed in Go for a while, but prior to Go 1.7 it wasn't part of the standard library. Instead, it was part of the /x/ packages¹ which are essentially a set of sub-repositories created by the Go core team, but developed under looser compatibility requirements.

In Go 1.7 this package was added to the standard library, and at the same time the `http.Request` object also had a context attached to it making it easier to add extra data to your requests.

The context package is intended to be used for cancellation, deadlines, and storing request-scoped values. We won't be using it for the first two uses cases, but if you would like to read more about them I suggest checking out the links at the bottom of [Box 14.1](#).

What we will be using the context package for is storing request-scoped values. Specifically, we will be using it to store the currently logged in user after looking them up in our `RequireUser` middleware, and then we will have a way to reuse that data later in our handlers without doing another database lookup.

When we want to retrieve a context from an http request, we would do so by writing the following code:

```
func someHandler(w http.ResponseWriter, r *http.Request) {
    // This is how we get a request's context
    ctx := r.Context()
}
```

Once we have a context, we can then interact with it much like we would any other context object. One important thing to note here is that when you interact with a context you create a new context derived from the original rather than altering the original. For example, if we wanted to add a value to our context we would use the `WithValue`² function provided by the context package which takes in an existing context, a key, and a value, and then returns us a new context with that value stored at that key.

¹<https://github.com/golang/go/wiki/SubRepositories>

²<https://golang.org/pkg/context/#WithValue>

```
import "context"

newCtx := context.WithValue(ctx, "my-key", "my-value")
```

After doing this, we could then retrieve our value by using the **Value** method provided by the context object.

```
// Retrieve the value stored at "my-key"
myKey := newCtx.Value("my-key")
```

One downside to this is that the context package returns all of its data as the empty interface type (**interface{}**). That means that before we can use the values returned, we will need to first convert them into the correct data type.

```
myKeyStr, ok := myKey.(string)
if !ok {
    // type conversion failed!
}
// otherwise myKeyStr will have a string representation of
// the value in myKey
```

This lack of type safety is one of the sources of controversy involved with the context package. By storing values in our context we lose some of the compile-time type safety we have come to expect. For example, if we stored an integer in our context instead of a string in the example above, the type assertion trying to turn it into a string would fail and our code would then have to try to recover from this manually, and that often isn't possible because we don't know how to proceed with invalid data.

```
newCtx := context.WithValue(ctx, "my-key", 123)
myKey := newCtx.Value("my-key")
if myKeyStr, ok := myKey.(string); !ok {
    // we have an issue here because myKey is an int, not a
    // string! We now have to try to recover in our code
    // instead of the compiler catching this for us!
    // The only way to proceed here is often with an error
}
```

Luckily, there is a way to combat this which involves creating your own context package that wraps some of the existing context package's functions and uses private keys. This is much easier to understand if we have code to go along with it, so we will be writing our own context package moving forward. We will start by creating the directory for the package and a source file.

```
$ mkdir context  
$ atom context/context.go
```

We will start with our imports and the package declaration.

```
package context

import (
    "context"
    "lenslocked.com/models"
)
```

We are going to import both the context package and the models package because we are going to write a function that accepts an existing context and a user, and then returns a new context with that user set as a value.

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

As long as developers only call the **WithUser** function to assign a value to the “user” key we will never have to worry about invalid data being stored at that key. That is, we don't have to worry about our type assertion failing like we discussed earlier.

We can also take this a step further. Right now anyone could accidentally writing the following code anywhere in our application (including in a third party library) and cause an issue:

```
newCtx := context.WithValue(ctx, "user", "invalid-data")
```

Luckily the keys used for context values have two components to them - the type, and the value. This means that using the string key “user” is not the same as using a different type, even if the underlying type is a string. For example, if you ran the following code you would see two different values stored in the context even though both keys appear to be the user string.

```
ctx := context.TODO()

type privateKey string
var a privateKey = "user"

ctx = context.WithValue(ctx, a, "123")
ctx = context.WithValue(ctx, "user", 456)

fmt.Println(ctx.Value(a))
fmt.Println(ctx.Value("user"))
```

This occurs because our privateKey type, while backed by a string, is not actually the same as a string, and the keys used for the context package take both the type and the value into consideration.

What this means for our code is that we can introduce a privateKey type for our context package, and then as long as we don’t export any of those keys it won’t be possible for code outside of the package to overwrite our context values.

```
type privateKey string

const (
    userKey privateKey = "user"
)

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

Neither privateKey or userKey are exported, so now code outside of the context package we created won't have a way to interact with any data stored at that key. This also means we won't be able to retrieve any values stored with these keys, so we need to provide a function to lookup a user from a given context.

```
func User(ctx context.Context) *models.User {
    if temp := ctx.Value(userKey); temp != nil {
        if user, ok := temp.(*models.User); ok {
            return user
        }
    }
    return nil
}
```

Our type assertion should always be true now, but we still need to check to be safe. We should also verify that a user was previously stored in the context, which is what our first “temp != nil” check is for.

If either of these issues arise, we will return nil. Otherwise we will return the user stored in the context after using a type assertion.

With our context package finished, we are going to update our RequireUser middleware to use this package. Inside our middleware we will retrieve the context from the request, create a new context with the user stored inside of it, and then create a new request with the updated context.

```
$ atom middleware/require_user.go
```

Add our new package to our imports.

```
import (
    // Add this
    "lenslocked.com/context"
)
```

Then head down to the `ApplyFn` function we wrote. Near the end of this function we are going to retrieve the context from the request like we saw earlier in this section. Once we have the context we will pass it into the `WithUser` function we wrote along with the user we want to set, and then set the return value to the `ctx` variable. Finally, we create a new request using the existing one and our new context. Much like we don't alter a context object directly and instead derive a new one, we also need to derive a new request using the existing one and a new context. We will then pass the new request to the next handler.

```
func (mw *RequireUser) ApplyFn(next http.HandlerFunc) http.HandlerFunc {
    return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
        // ... cookie lookup and user verification remain unchanged

        // Get the context from our request
        ctx := r.Context()
        // Create a new context from the existing one that has
        // our user stored in it with the private user key
        ctx = context.WithUser(ctx, user)
        // Create a new request from the existing one with our
        // context attached to it and assign it back to `r`.
        r = r.WithContext(ctx)
        // Call next(w, r) with our updated context.
        next(w, r)
    })
}
```

Finally, we need to update the `Create` method in our `Galleries` controller to retrieve this user and assign the appropriate `UserID` to the gallery we are creating.

```
$ atom controllers/galleries.go
```

Find the `Create` method. We will be updating it just after we parse the form, and we will be looking up the user. We will be retrieving the user from the context, and then using that user's ID as our `UserID` field for the `Gallery` model we construct.

```
func (g *Galleries) Create(w http.ResponseWriter, r *http.Request) {
    // ... parse the form w/ the existing code

    // This is our new code
    user := context.User(r.Context())
    // Then we update how we build the Gallery model
    gallery := models.Gallery{
        Title: form.Title,
        UserID: user.ID,
    }

    // ... then we create the gallery just like we were, using
    // the g.gs.Create(...) method call.
}
```

We are now be able to create galleries using our form, and every gallery we create should automatically be assigned a UserID based on whoever is currently logged in. You can test this by creating new accounts (or logging into existing ones) and creating a gallery with each account. Then check your logs to verify that each gallery was created with the appropriate user ID.

Source code

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

Completed - book.usegolang.com/14.6.2

Changes - book.usegolang.com/14.6.2-diff

14.7 Displaying galleries

The next handler we are going to implement is the one used to render a gallery after it has been created. This will be pretty boring right now because we don't have any images in our gallery, but we can update it later once we implement

image uploading. For now we want to focus on just displaying the title of a gallery and redirecting a user there after they create a new gallery.

We are going to start by creating a view and adding it to our controller. After that we will update our routes and look at how to parse URL parameters using gorilla/mux. Once we have the ID of the gallery we are supposed to render we will need to write some code to retrieve it from the database, and finally we will look at how to generate a URL for an existing gallery using our router, a named route, and the gallery's ID. We will then use the generated URL to redirect a user once they create a new gallery.

14.7.1 Creating the show gallery view

The first piece we need is a template used to render our gallery. The only data we have access to with a gallery is its title right now, so that vastly simplifies our view.

```
$ atom views/galleries/show.gohtml
```

```
{{define "yield"}}
<div class="row">
  <div class="col-md-12">
    <h1>
      {{.Title}}
    </h1>
    <p>Images coming soon...</p>
  </div>
</div>
{{end}}
```

This template will render the title of a gallery along with a little message indicating that we will be adding images soon.

We need to setup the view inside of our galleries controller for this template as well, so we can do that next.

```
$ atom controllers/galleries.go
```

First we update the Galleries type to include a field for our new view. We'll name it ShowView.

```
type Galleries struct {
    New      *views.View
    ShowView *views.View
    gs       models.GalleryService
}
```

After that we need to update the NewGalleries function so that it handles setting up the view.

```
func NewGalleries(gs models.GalleryService) *Galleries {
    return &Galleries{
        New:      views.NewView("bootstrap", "galleries/new"),
        ShowView: views.NewView("bootstrap", "galleries/show"),
        gs:       gs,
    }
}
```

With that we have our view setup, but we can't render it just yet. We need to implement a few other things first, such as a way to parse the ID of the gallery we want to render from the URL.

Source code

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

Completed - book.usegolang.com/14.7.1

Changes - book.usegolang.com/14.7.1-diff

14.7.2 Parsing the gallery ID from the path

When we learned about REST, we discussed how creating a RESTful application will affect the paths we use for our resources. For example, when you want to create a new gallery the path we use is:

```
/galleries/new
```

We also discussed in [Table 7.4](#) how specific galleries can be mapped to paths that include the ID in them. For example, whenever you want to view a gallery with the ID **12**, you might visit the path:

```
/galleries/12
  ^
  ^
resource   ^
        ID
```

In this section we are going to see how to implement this using our gorilla/mux router's path variables feature, which allows us to add parameters into our URL and then parse them out later inside of our controller. Below are a few examples taken from the gorilla/mux docs.

```
r := mux.NewRouter()
r.HandleFunc("/products/{key}", ProductHandler)
r.HandleFunc("/articles/{category}/{id:[0-9]+}", ArticleHandler)
```

In the first example we see a variable named **key** as the last portion of the path. In this case, the router will capture anything after the **/products/** prefix and before a trailing slash (**/**) and place it into the **key** variable.

In the second example we see something similar with the **category** variable, but there is also a second variable named **id** that is different. Rather than accepting anything as an ID, we are using a regular expression pattern to limit

which characters can be mapped to that variable. In this particular case we are saying that `id` can only be composed of numeric characters, or in other words we are saying that the `id` has to be a number.

Below are a few examples of paths and what their corresponding variables would be mapped to with the code above.

```
If you visit: /products/something-awesome
The "key" variable will get the value: something-awesome

If you visit: /articles/dogs/25
The "category" variable will have the value: dogs
And the "id" variable will have the value: 25
```

To access these variables, we would use the gorilla/mux package inside of our handler function with code similar the snippet below.

```
func someHandler(w http.ResponseWriter, r *http.Request) {
    vars := mux.Vars(r)
    category := vars["category"]
}
```

In the first line we call the `Vars` function³ which will return a map of keys and their values.

```
// From the gorilla/mux docs...
func Vars(r *http.Request) map[string]string
```

Once we have our map we can then look up any variables defined in our path using the variable name as the key in our map. One thing that is important to note is that all of these will be strings by default, so if you need to convert something like an ID into an integer you will need to use a conversion function, such

³<http://www.gorill toolkit.org/pkg/mux#Vars>

as **Atoi**⁴ from the **strconv** package⁵, which can be used to convert strings to integers.

Note: Atoi stands for “ASCII to integer”, and is a legacy name that has been used for functions converting strings to integers for a very long time in programming.

Now that we know what we are about to do, let’s start by defining the route for our show gallery page.

```
$ atom main.go
```

We are going to introduce a single new route after our existing gallery routes.

```
func main() {
    // ... most of this remains unchanged

    r := mux.NewRouter()
    // ... we have a few existing routes here

    // Gallery routes
    // ... leave the existing gallery routes alone and add the
    // new route below.
    r.HandleFunc("/galleries/{id:[0-9]+}",
        galleriesC.Show).Methods("GET")

    // ... the remaining code is unchanged
}
```

This code tells our router that we want to map all web requests that start with **/galleries/** and end with a numeric ID to the Show method on our galleries controller. We haven’t defined that method yet, so let’s do that next.

⁴<https://golang.org/pkg/strconv/#Atoi>

⁵<https://golang.org/pkg/strconv/>

```
$ atom controllers/galleries.go
```

Rather than trying to explain this all after the fact, we will look at the completed code below with comments before nearly every line intended to help explain what each line of code is used for.

```
import (
    // Add this import
    "strconv"
)

// GET /galleries/:id
func (g *Galleries) Show(w http.ResponseWriter, r *http.Request) {
    // First we get the vars like we saw earlier. We do this
    // so we can get variables from our path, like the "id"
    // variable.
    vars := mux.Vars(r)
    // Next we need to get the "id" variable from our vars.
    idStr := vars["id"]
    // Our idStr is a string, so we use the Atoi function
    // provided by the strconv package to convert it into an
    // integer. This function can also return an error, so we
    // need to check for errors and render an error.
    id, err := strconv.Atoi(idStr)
    if err != nil {
        // If there is an error we will return the StatusNotFound
        // status code, as the page requested is for an invalid
        // gallery ID, which means the page doesn't exist.
        http.Error(w, "Invalid gallery ID", http.StatusNotFound)
        return
    }
    // This line is only to prevent the compiler from complaining
    // about us not using the id parameter. We will use it later.
    _ = id

    // Finally we need to lookup the gallery with the ID we
    // have, but we haven't written that code yet! For now we
    // will create a temporary gallery to test our view.
    gallery := models.Gallery{
        Title: "A temporary fake gallery with ID: " + idStr,
    }
    // We will build the views.Data object and set our gallery
    // as the Yield field, but technically we do not need
    // to do this and could just pass the gallery into the
    // Render method because of the type switch we coded into
```

```
// the Render method.  
var vd views.Data  
vd.Yield = gallery  
g.ShowView.Render(w, vd)  
}
```

While our code isn't rendering a gallery from our database, we are now able to view and verify that our show gallery page is working by navigating to a path like: localhost:3000/galleries/123

On that page we should see a gallery title like the one we defined in our Show method, and if we change the ID in our path it should update the title. With the show page completed we are ready to work on writing code that allows us to retrieve a gallery from our database via an ID.

Source code

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

Completed - book.usegolang.com/14.7.2

Changes - book.usegolang.com/14.7.2-diff

14.7.3 Looking up galleries by ID

Our next objective is to add a method to our GalleryService that allows us to look up an existing gallery via its ID. This will be nearly identical to the ByID method we wrote for the UserService, with the only real difference being the type of resource we use to store the information retrieved from the database. GORM will handle deciding which database table to use for us based off of that resource's type.

Open up the galleries model. We will start by adding the ByID method to our GalleryDB interface.

```
$ atom models/galleries.go
```

```
// GalleryDB is used to interact with the galleries database.
//
// For pretty much all single gallery queries:
// If the gallery is found, we will return a nil error
// If the gallery is not found, we will return ErrNotFound
// If there is another error, we will return an error with
// more information about what went wrong. This may not be
// an error generated by the models package.
type GalleryDB interface {
    ByID(id uint) (*Gallery, error)
    Create(gallery *Gallery) error
}
```

Our ByID method is going to accept an unsigned integer for its **id** argument, and then will return either the gallery requested or an error. The errors possible are the same as the errors with the UserDB interface, but it is a good idea to document that here as well.

Next we need to implement this method on our galleryGorm type.

```
func (gg *galleryGorm) ByID(id uint) (*Gallery, error) {
    var gallery Gallery
    db := gg.db.Where("id = ?", id)
    err := first(db, &gallery)
    if err != nil {
        return nil, err
    }
    return &gallery, nil
}
```

As mentioned early, the ByID method is nearly identical to the one we implemented for our userGorm type, but this one uses a Gallery instead of a User variable. When we then call the **first** method, which subsequently calls the

First method provided by GORM, the type of this variable will be used to determine what table in our database we want to access. We don't see that code because it happens inside the **gorm** package, but that is how our code is deciding which database table to query.

The last thing we are going to do in this section is update our Show method on our galleries controller to look up a real gallery and then render it.

```
$ atom controllers/galleries.go
```

```
func (g *Galleries) Show(w http.ResponseWriter, r *http.Request) {
    vars := mux.Vars(r)
    idStr := vars["id"]
    id, err := strconv.Atoi(idStr)
    if err != nil {
        http.Error(w, "Invalid gallery ID", http.StatusNotFound)
        return
    }
    // Our new code starts here
    gallery, err := g.gs.ByID(uint(id))
    if err != nil {
        switch err {
        case models.ErrNotFound:
            http.Error(w, "Gallery not found", http.StatusNotFound)
        default:
            http.Error(w, "Whoops! Something went wrong.",
                http.StatusInternalServerError)
        }
        return
    }
    var vd views.Data
    vd.Yield = gallery
    g.ShowView.Render(w, vd)
}
```

Rather than building a gallery ourselves, we are now using our gallery service to retrieve one. The ByID method expects an unsigned integer, but our id is stored as a signed integer (**int**) so we also need to convert it into a **uint** before passing it into the ByID method call.

Once we get our results back from the ByID method call we then check to see if we had an error. If there was an error and it was ErrNotFound we will

render a message indicating that we couldn't find the gallery and return a 404 (StatusNotFound) status code. If the error was anything else we are going to render a generic error message for now, as we don't have an appropriate view to fall back on with an alert message.

The rest of the code used to render the ShowView when there isn't an error remains unchanged, but remember that our new Show method will only work if you provide it with a valid gallery ID! That means you need to create a gallery using our new gallery form and then use that ID when you visit the show gallery page.

Source code

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

Completed - book.usegolang.com/14.7.3

Changes - book.usegolang.com/14.7.3-diff

14.7.4 Generating URLs with params

In this section we are going to learn how to automatically redirect a user to a gallery after they create it. For example, after a user creates a gallery and it has the ID **34** assigned to it, we might manually redirect them to it with the code below.

```
gallery := ...
path := fmt.Sprintf("/galleries/%d", gallery.ID)
http.Redirect(w, r, path, http.StatusFound)
```

This would would because the Sprintf function would replace the **%d** with the gallery ID, giving us a path like “/galleries/34”. The downside to this approach

is that if we were to later change the path to be something like “/albums/34”, we would have to find all of the code like this and update it.

Rather than writing our code this way, it is easier to instead write a single function used to create these paths and then use them throughout all of our code. For instance, we might use the following code to create our galleries paths.

```
func galleryPath(id uint) string {
    return fmt.Sprintf("/galleries/%d", id)
}
```

The code isn’t actually any different, but now when if we change the path we only need to update our code in a single place.

The gorilla/mux package’s router already provides functionality similar to this. Rather than creating an individual function for each path, we simply use named routes.

```
r := mux.NewRouter()
r.HandleFunc("/dogs/{id:[0-9]+}", dogHandler).Name("show_dog")
```

By naming our route, we can later use the router to reconstruct that route.

```
url, err := r.Get("show_dog").URL("id", "34")
```

The first method, **Get**, is used to retrieve the named route. The second method, **URL**, is used to create a **url.URL**⁶, which is a type provided by the url package. The only data we need to pass into the URL method are the parameters we need to provide for the route. For each parameter we first provide the parameter name (eg “id”), and then we provide a value for it in string format (eg “34”).

⁶<https://golang.org/pkg/net/url/#URL>

Note: The URL method works this way because it can't know how many variables your route has, but by using a variadic parameter it can accept as many or as few as necessary.

Learning all of this implies a couple things:

1. We are going to need to have access to our router inside of our galleries controller if we want to rebuild the path to view a single gallery.
2. We need to name our show gallery route and make that name available to both the controller and our main.go source file.

Let's start with the first. We are going to add the router to our Galleries type provided by our controller package so that we can access that router later in our handler methods.

```
$ atom controllers/galleries.go
```

```
import (
    // Add this import
    "github.com/gorilla/mux"
)

func NewGalleries(gs models.GalleryService, r *mux.Router) *Galleries {
    return &Galleries{
        New:      views.NewView("bootstrap", "galleries/new"),
        ShowView: views.NewView("bootstrap", "galleries/show"),
        gs:       gs,
        r:        r,
    }
}

type Galleries struct {
    New      *views.View
    ShowView *views.View
    gs       models.GalleryService
    r        *mux.Router
}
```

Next we need to update `main.go` to provide the router to the `NewGalleries` function. In order to do that we have to create the router *before* we create our controllers, which isn't what we have been doing so far.

```
$ atom main.go
```

```
func main() {
    // ... We still setup our services the same way

    // Add this BEFORE we create the controllers. It was
    // previously coded after.
    r := mux.NewRouter()
    staticC := controllers.NewStatic()
    usersC := controllers.NewUsers(services.User)
    // We also need to update this function call to provide the
    // router.
    galleriesC := controllers.NewGalleries(services.Gallery, r)

    requireUserMw := middleware.RequireUser{
        UserService: services.User,
    }

    // We no longer need this router instantiation. You can
    // delete the following line.
    // r := mux.NewRouter()

    // ... our routes and everything else remain unchanged.
}
```

Next we need to define a name for our show galleries route. We will do that inside of the galleries controller source file, and will store it in a constant named `ShowGallery`.

```
$ atom controllers/galleries.go
```

```
const (
    ShowGallery = "show_gallery"
)
```

Next we need to go back to our main source file where we declare the route and name it.

```
$ atom main.go
```

Find the route in your main function that is coded similar to the one below.

```
r.HandleFunc("/galleries/{id:[0-9]+}", galleriesC.Show).Methods("GET")
```

We are going to update it to name the router after creating it.

```
// Add the Name method to our method chain
r.HandleFunc("/galleries/{id:[0-9]+}", galleriesC.Show).
    Methods("GET").Name(controllers>ShowGallery)
```

Finally, we head back to the galleries controller to update the Create method. We are going to add some code to create a URL using the ShowGallery named route, and then redirect a user to that page after they have created a gallery.

```
$ atom controllers/galleries.go
```

```
func (g *Galleries) Create(w http.ResponseWriter, r *http.Request) {
    // ... Leave most of this code the same

    // Remove any existing Fprintln statements like the one
    // below.
    // fmt.Fprintln(w, gallery)

    // Then add the following code to create a URL using the
    // router and the named ShowGallery route.
    url, err := g.r.Get>ShowGallery).URL("id",
        strconv.Itoa(int(gallery.ID)))
    // Check for errors creating the URL
    if err != nil {
```

```
    http.Redirect(w, r, "/", http.StatusFound)
    return
}
// If no errors, use the URL we just created and redirect
// to the path portion of that URL. We don't need the
// entire URL because your application might be hosted at
// localhost:3000, or it might be at lenslocked.com. By
// only using the path our code is agnostic to that detail.
http.Redirect(w, r, url.Path, http.StatusFound)
}
```

*Note: We are using the **Itoa** function in the code above, which is the opposite of the **Atoi** function we saw earlier. While Atoi will convert a string into an integer, Itoa will convert an integer into a string. We just need to cast our **uint** into an integer before passing it into the function.*

With that final change to our Create method we are done introducing the Show action to our controller. Restart your server and try creating a new gallery by visiting localhost:3000/galleries/new and submitting the new gallery form with a title. You will be redirected to the show gallery page with the ID portion of the URL filled in with the ID of the gallery you just created. The show gallery page will also render your new gallery's title as a heading for the page.

Source code

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

Completed - book.usegolang.com/14.7.4

Changes - book.usegolang.com/14.7.4-diff

14.8 Editing galleries

Editing galleries, much like editing any resource, is going to require us to create a few things.

First, we'll need to create a view used to render a form for editing galleries and an action to render the view. We will want a separate action for this because we will need to lookup the gallery and verify that the currently logged in user does have permission to edit the gallery they are requesting.

Note: Our show gallery action we created in the last section doesn't check permissions because all galleries in our application are currently public for viewing, but only the owner of a gallery can edit it.

After that we will need a way to update our galleries in the database. We will make this happen by adding an Update method to our GalleryDB interface in the models package, and this will ultimately be very similar to the Update method provided by our UserService.

Finally, we will need a way to process the edit gallery form, verify once again that the logged in user has permission to edit the form, and then update the gallery in the database using the Update method we introduced in the previous step.

We will perform each of these steps in a subsection below.

14.8.1 The edit gallery action

Like most new pages we create, we are going to start by creating the template used to render the view. In the case of our edit gallery page, we are going to have some basic bootstrap layout code along with an edit gallery form that submits the form data to:

```
POST /galleries/:ID/update
```

When we learned about REST we learned that updates are typically done with a PUT, but unfortunately most modern browsers don't support this very well. The only real way to make a PUT work is with a JavaScript hack, but it is often easier to just use a unique URL for updates and a POST for your web forms. Your web server could then still support the PUT route, but you aren't forced to introduce extra JavaScript to your site.

```
$ atom views/galleries/edit.gohtml
```

```
 {{define "yield"}}
<div class="row">
  <div class="col-md-6 col-md-offset-3">
    <div class="panel panel-primary">
      <div class="panel-heading">
        <h3 class="panel-title">Edit your gallery</h3>
      </div>
      <div class="panel-body">
        {{template "editGalleryForm" .}}
      </div>
    </div>
  </div>
{{end}}
```

```
 {{define "editGalleryForm"}}
<form action="/galleries/{{.ID}}/update" method="POST">
  <div class="form-group">
    <label for="title">Title</label>
    <input type="text" name="title" class="form-control" id="title" placeholder="What is the title of your gallery?" value="{{.Title}}"/>
  </div>
  <button type="submit" class="btn btn-primary">Update</button>
</form>
{{end}}
```

One important thing to note in our editGalleryForm template is that we have now set the current title of a gallery as the default value in our input field. This is done with the code that reads:

```
<input ... value="{{.Title}}">
<!-- ^ This part sets the initial value of the input -->
```

Doing this means that we need to have an existing gallery passed into our template, which we will do later. For now let's update our galleries controller so that it knows about our new view.

```
$ atom controllers/galleries.go
```

```
func NewGalleries(gs models.GalleryService, r *mux.Router) *Galleries {
    return &Galleries{
        New:      views.NewView("bootstrap", "galleries/new"),
        ShowView: views.NewView("bootstrap", "galleries/show"),
        EditView: views.NewView("bootstrap", "galleries/edit"),
        gs:       gs,
        r:        r,
    }
}

type Galleries struct {
    New      *views.View
    ShowView *views.View
    EditView *views.View
    gs       models.GalleryService
    r        *mux.Router
}
```

Now we can work on creating an action used to look up a gallery and then render the edit view with it. To do this, we need to first retrieve the ID of the gallery that the user is trying to edit from the path. We will use gorilla/mux to do this just like we did in the Show action. In fact, all of our code used to retrieve the ID and then lookup the gallery using our GalleryService is going to be pretty much identical, so why don't we move that into a reusable function.

```
func (g *Galleries) galleryByID(w http.ResponseWriter,
    r *http.Request) (*models.Gallery, error) {
    vars := mux.Vars(r)
```

```

idStr := vars["id"]
id, err := strconv.Atoi(idStr)
if err != nil {
    http.Error(w, "Invalid gallery ID", http.StatusNotFound)
    return nil, err
}
gallery, err := g.gs.ByID(uint(id))
if err != nil {
    switch err {
    case models.ErrNotFound:
        http.Error(w, "Gallery not found", http.StatusNotFound)
    default:
        http.Error(w, "Whoops! Something went wrong.", http.StatusInternalServerError)
    }
    return nil, err
}
return gallery, nil
}

```

As you can see, nearly all of the code in galleryByID came directly from our Show method. That means we can update the Show method to use the galleryByID method instead of doing this all on its own.

```

func (g *Galleries) Show(w http.ResponseWriter, r *http.Request) {
    gallery, err := g.galleryByID(w, r)
    if err != nil {
        // The galleryByID method will already render the error
        // for us, so we just need to return here.
        return
    }
    var vd views.Data
    vd.Yield = gallery
    g.ShowView.Render(w, vd)
}

```

By moving our code to a reusable method we were able to vastly simplify the Show method, but we still need to do one quick error check to determine if we received a gallery or not. If we did, we can render the ShowView. Otherwise we are going to rely on the galleryByID method to render the error and return.

Getting back to the Edit method, we are going to also use the galleryByID method here. After retrieving a gallery we will check for an error exactly like

we do in Show, and then we will retrieve the user from the request context and verify that the user has access to this gallery. We can do this by verifying that the UserID field of the gallery is the same as the user's ID, which means that the user owns the gallery. If that isn't true we will render an error, otherwise we will continue rendering the EditView.

```
// GET /galleries/:id/edit
func (g *Galleries) Edit(w http.ResponseWriter, r *http.Request) {
    gallery, err := g.galleryByID(w, r)
    if err != nil {
        // The galleryByID method will already render the error
        // for us, so we just need to return here.
        return
    }
    // A user needs logged in to access this page, so we can
    // assume that the RequireUser middleware has run and
    // set the user for us in the request context.
    user := context.User(r.Context())
    if gallery.UserID != user.ID {
        http.Error(w, "You do not have permission to edit "+
            "this gallery", http.StatusForbidden)
        return
    }
    var vd views.Data
    vd.Yield = gallery
    g.EditView.Render(w, vd)
}
```

Lastly, we need to update our router so that it knows about our edit gallery page.

```
$ atom main.go
```

Add the following line to the `main` function right below the existing routes. We are applying the `RequireUser` middleware (accessibly via the `requireUserMw` variable) here because a user needs to be logged in order to edit a gallery.

```
r.HandleFunc("/galleries/{id:[0-9]+}/edit",
    requireUserMw.ApplyFn(galleriesC.Edit)).Methods("GET")
```

The path we are using is very similar to the show gallery page but with a trailing `/edit` in the path after the ID. That may seem minor, but this will allow our router to differentiate between the following two paths:

```
yoursite.com/galleries/22      ->  show page  
yoursite.com/galleries/22/edit  ->  edit page
```

To test your code, create two different user accounts and create a gallery with each. Then log into one of the user accounts and verify that you can visit both galleries' show page.

```
localhost:3000/galleries/:id
```

Then try to visit the edit page for each gallery. You should only be able to visit the edit page for the gallery you created with that account. The other should render an error message indicating that you do not have permission to edit the gallery.

Source code

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

Completed - book.usegolang.com/14.8.1

Changes - book.usegolang.com/14.8.1-diff

14.8.2 Parsing the edit gallery form

Next up is the handler used to process the edit gallery form when we submit it.

```
$ atom controllers/galleries.go
```

It is fairly common to name your handler for rendering an edit form `Edit`, and then to name your handler that processes the form `Update`, so we will follow that same naming schema here.

```
// POST /galleries/:id/update
func (g *Galleries) Update(w http.ResponseWriter, r *http.Request) {
    // TODO: Implement this
}
```

We also know we are going to be processing incoming requests that have the existing gallery ID included in the path, so we can start by looking up the existing gallery in our database using the `galleryByID` method. Once we have the gallery, we will want to verify that the signed in user has permission to edit it as well. That means the first half of our `Update` method is going to match our `Edit` method, but we'll allow the duplicate code for now.

```
func (g *Galleries) Update(w http.ResponseWriter, r *http.Request) {
    gallery, err := g.galleryByID(w, r)
    if err != nil {
        return
    }
    user := context.User(r.Context())
    if gallery.UserID != user.ID {
        http.Error(w, "Gallery not found", http.StatusNotFound)
        return
    }
    // TODO: Finish implementing this
}
```

The last half is where we are going to see some variation between our `Update` and `Edit` methods. In the last half of the `Update` method we need to create a `GalleryForm` instance, parse the incoming form data with it, and then render any errors if there are any.

```
func (g *Galleries) Update(w http.ResponseWriter, r *http.Request) {
    // ... leave our existing code alone

    var vd views.Data
    vd.Yield = gallery
    var form GalleryForm
    if err := parseForm(r, &form); err != nil {
        // If there is an error we are going to render the
        // EditView again with an alert message.
        vd.SetAlert(err)
        g.EditView.Render(w, vd)
        return
    }
    gallery.Title = form.Title
    // TODO: Persist this gallery change in the DB after
    // we add an Update method to our GalleryService in the
    // models package.
    vd.Alert = &views.Alert{
        Level:  views.AlertLvlSuccess,
        Message: "Gallery updated successfully!",
    }
    g.EditView.Render(w, vd)
}
```

We don't have a way to actually update our gallery using the `GalleryService` yet, so we will code that in the next section then come back to finish our `Update` handler. In the meantime, we are rendering the edit page again with the submitted data and an alert message pretending that the gallery was successfully updated.

Let's tell our router about the new route so we can verify this is all working.

```
$ atom main.go
```

Add the following route into the `main` function near the existing gallery routes.

```
r.HandleFunc("/galleries/{id:[0-9]+}/update",
    requireUserMw.ApplyFn(galleriesC.Update)).Methods("POST")
```

Restart your server and try submitting the edit gallery form. You will now see the updated information in the form and an alert message saying the gallery was updated, but if you reload the page you will see that the update didn't actually happen. In the next section we will cover how to implement that portion of the code.

Source code

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

Completed - book.usegolang.com/14.8.2

Changes - book.usegolang.com/14.8.2-diff

14.8.3 Updating gallery models

In this section we will be writing the code necessary to push gallery changes to our database. We'll also be using validations again, as we don't want to allow galleries to be updated with invalid data.

We will be starting off in the galleries model source, where we will be adding an Update method to the GalleryDB interface.

```
$ atom models/galleries.go
```

```
type GalleryDB interface {
    ByID(id uint) (*Gallery, error)
    Create(gallery *Gallery) error
    Update(gallery *Gallery) error
}
```

Next we need to code this method. We will start with the simpler part - the galleryGorm implementation.

```
func (gg *galleryGorm) Update(gallery *Gallery) error {
    return gg.db.Save(gallery).Error
}
```

In the new Update method we are using the **Save** method provided by GORM and returning any errors encountered in the process.

Now we need to add validations for our Update method, which we do by adding an Update method to the galleryValidator type. We will be using two validations we already created; the first will verify that the UserID field is set correctly, and the second will verify that a title is provided for the gallery.

```
func (gv *galleryValidator) Update(gallery *Gallery) error {
    err := runGalleryValFns(gallery,
        gv.userIDRequired,
        gv.titleRequired)
    if err != nil {
        return err
    }
    return gv.GalleryDB.Update(gallery)
}
```

With our Update method added to the GalleryService (via the embedded GalleryDB) we can go back to the galleries controller and use this new method in our Update action there.

```
$ atom controllers/galleries.go
```

We will only be altering the last portion of the Update method where we placed a TODO before, so the first half of the method is left blank for brevity.

```
func (g *Galleries) Update(w http.ResponseWriter, r *http.Request) {
    // ... this code remains unchanged

    gallery.Title = form.Title
    err = g.gs.Update(gallery)
    // If there is an error our alert will be an error. Otherwise
    // we will still render an alert, but instead it will be
    // a success message.
    if err != nil {
        vd.SetAlert(err)
    } else {
        vd.Alert = &views.Alert{
            Level:  views.AlertLvlSuccess,
            Message: "Gallery successfully updated!",
        }
    }
    // Error or not, we are going to render the EditView with
    // our updated information.
    g.EditView.Render(w, vd)
}
```

Now our code supports editing galleries! Restart your server and give it a test run. Then navigate back to the view gallery page to verify that it is indeed persisting the new title when you submit the edit gallery form.

Source code

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

Completed - book.usegolang.com/14.8.3

Changes - book.usegolang.com/14.8.3-diff

14.9 Deleting galleries

Just like our Update action, implementing the delete feature for our galleries is going to involve touching our models package, our galleries controller, and a view template. Unlike our update action, we won't be creating a new view entirely; instead we will add a delete button to the edit gallery page, as the edit page is a fairly common place to go if you want to delete a resource.

This time we will start by implementing the Delete method in the models package.

```
$ atom models/galleries.go
```

First we introduce the method to our GalleryDB interface.

```
type GalleryDB interface {
    ByID(id uint) (*Gallery, error)
    Create(gallery *Gallery) error
    Update(gallery *Gallery) error
    Delete(id uint) error
}
```

Then we implement it on the galleryGorm type.

```
func (gg *galleryGorm) Delete(id uint) error {
    gallery := Gallery{Model: gorm.Model{ID: id}}
    return gg.db.Delete(&gallery).Error
}
```

And finally we write a validation function that ensures we don't try to delete with an invalid ID (one that is 0).

```

func (gv *galleryValidator) nonZeroID(gallery *Gallery) error {
    if gallery.ID <= 0 {
        return ErrIDInvalid
    }
    return nil
}

func (gv *galleryValidator) Delete(id uint) error {
    var gallery Gallery
    gallery.ID = id
    if err := runGalleryValFns(&gallery, gv.nonZeroID); err != nil {
        return err
    }
    return gv.GalleryDB.Delete(gallery.ID)
}

```

Next we are going to introduce a controller to process a delete request.

```
$ atom controllers/galleries.go
```

We will name our handler method Delete, since it is used to handle requests to delete a gallery. In our method we need to lookup the gallery being deleted, verify that the user has permission to delete it, and then use the Delete method provided by our GalleryService.

```

import (
    // You may need to add this import
    "fmt"
)

// POST /galleries/:id/delete
func (g *Galleries) Delete(w http.ResponseWriter, r *http.Request) {
    // Lookup the gallery using the galleryByID we wrote earlier
    gallery, err := g.galleryByID(w, r)
    if err != nil {
        // If there is an error the galleryByID will have rendered
        // it for us already.
        return
    }
    // We also need to retrieve the user and verify they have
    // permission to delete this gallery. This means we will
    // need to use the RequireUser middleware on any routes
}

```

```
// mapped to this method.
user := context.User(r.Context())
if gallery.UserID != user.ID {
    http.Error(w, "You do not have permission to edit "+
        "this gallery", http.StatusForbidden)
    return
}

var vd views.Data
err = g.gs.Delete(gallery.ID)
if err != nil {
    // If there is an error we want to set an alert and
    // render the edit page with the error. We also need
    // to set the Yield to gallery so that the EditView
    // is rendered correctly.
    vd.SetAlert(err)
    vd.Yield = gallery
    g.EditView.Render(w, vd)
    return
}
// TODO: We will eventually want to redirect to the index
// page that lists all galleries this user owns, but for
// now a success message will suffice.
fmt.Fprintln(w, "successfully deleted!")
}
```

Next we are going to add a route to our router that maps to our Delete method on the galleries controller. As mentioned earlier, browser forms don't play well with any HTTP methods other than POST, so we are going to use a special route for deleting that accepts POST requests in order to avoid needed any special JavaScript code.

```
$ atom main.go
```

Add the following route near your other gallery routes.

```
r.HandleFunc("/galleries/{id:[0-9]+}/delete",
    requireUserMw.ApplyFn(galleriesC.Delete)).Methods("POST")
```

Finally, we need to update our edit page to add a delete button. In order to make the button perform a POST request, we will be placing our button inside of its

own HTML **form** that doesn't have any data, but will have perform a POST to the delete path.

```
$ atom views/galleries/edit.gohtml
```

Find the editGalleryForm template and update it with the following code.

```
 {{define "editGalleryForm"}}
<form action="/galleries/{{.ID}}/update" method="POST">
  <div class="form-group">
    <label for="title">Title</label>
    <input type="text" name="title" class="form-control" id="title" placeholder="What is the title of this gallery?">
  </div>
  <button type="submit" class="btn btn-primary">Update</button>
</form>
<!-- This is our new Delete button that submits a POST request to the /galleries/:id/delete path. --&gt;
&lt;form action="/galleries/{{.ID}}/delete" method="POST"&gt;
  &lt;button type="submit" class="btn btn-danger"&gt;Delete&lt;/button&gt;
&lt;/form&gt;
{{end}}</pre>
```

Restart your web server and head over to the edit page for an existing gallery. Click the delete button to test out the code we just wrote. We are now able to delete galleries after creating them!

Note: The delete button is somewhat ugly right now with poor spacing, but we will clean that up later in the book. For now we just want to verify that it is working.

Source code

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

Completed - book.usegolang.com/14.9

Changes - book.usegolang.com/14.9-diff

14.10 Viewing all owned galleries

The next page we are going to introduce is what is commonly referred to as the index page. Index pages are typically used to list all of a specific resource, though they may often be filtered or sorted.

In our case we are going to be introducing an index page for our gallery resource. Rather than listing all galleries, we will also be filtering our index page to make it more useful. Whenever a user visits the index page, instead of seeing every gallery they will instead see only galleries that they own.

Along with listing each gallery, we will also provide links to view and edit each gallery, giving our users a way to navigate between and manage all of their galleries.

We are going to break the code necessary for this new page into NNN sections. We will start by updating the models package, introducing a new method used to retrieve all of a specific user's galleries. After that we will write a stubbed out index view along with our controller code that connects all pieces together. During that step we will also add the index route to our web application's router. In the final step we will implement a new view used to render the index page; in the process we will learn how to use the **range** action inside of a Go template to iterate over slices of data and render each individually.

14.10.1 Querying galleries by user ID

In order to query for galleries using a user ID, we are going to once again need to add a method to our GalleryDB interface so that it is available in the GalleryService we use to interact with the database.

```
$ atom models/galleries.go
```

We know that our method needs to return a slice of galleries when it is successful

because a user could have many galleries. We also know that it is possible to encounter an error when interacting with a database, whether it is due to a mistake of our own or because the database has crashed, so we can start off with a method that accepts a user ID and returns those two pieces of data.

```
type GalleryDB interface {
    ByID(id uint) (*Gallery, error)
    ByUserID(userID uint) ([]Gallery, error)
    Create(gallery *Gallery) error
    Update(gallery *Gallery) error
    Delete(id uint) error
}
```

Next we want to implement this method on our galleryGorm type. Doing this is actually very similar to the code we have been writing so far, but rather than calling the **First** method on our ***gorm.DB** type, we are instead going to be calling the **Find** method. The primary difference between these two methods is that First is intended to retrieve a single resource (the first one!), while Find is intended to return many instances of the resource. As a result, First expects a single resource passed in, while Find expects a slice.

```
func (gg *galleryGorm) ByUserID(userID uint) ([]Gallery, error) {
    var galleries []Gallery
    // We build this query *exactly* the same way we build
    // a query for a single user.
    db := gg.db.Where("user_id = ?", userID)
    // The real difference is in using Find instead of First
    // and passing in a slice instead of a single gallery as
    // the argument.
    if err := db.Find(&galleries).Error; err != nil {
        return nil, err
    }
    return galleries, nil
}
```

Note: Long term it might be necessary to introduce features like pagination, but this typically only becomes problematic when users have hundreds of galleries. I typically prefer to launch an application first, and then come back to add these features afterwards, which is what we will do in this book.

We won't be adding any validations to our ByUserID method at this time because there isn't a great need for them. If you wanted, you could use the nonZeroID validation we created earlier, but unlike in the Delete method this isn't necessary it won't be destructive. It will simply return galleries without a UserID set properly, which shouldn't happen now that we have validations on our Create and Update methods.

If you would like to verify that your code is working you can update your `main.go` source file to make a test query and print out the results, but we won't be committing this code.

```
// This is test code that won't be committed
testUserID := 1 // pick an ID that maps to a user in your DB
galleries, _ := services.Gallery.ByUserID(testUserID)
fmt.Println(galleries)
```

Source code

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

Completed - book.usegolang.com/14.10.1

Changes - book.usegolang.com/14.10.1-diff

14.10.2 Adding the Index handler

Next we need to introduce a function used to handle incoming web requests to the index page. We are also going to need a view for this page, but we won't be implementing it until the next section. For now we will simply stub it out.

```
$ atom views/galleries/index.gohtml
```

```
{{define "yield"}}
TODO: Implement the Index view.
Data provided: {{.}}
{{end}}
```

Next we need to add this view to our galleries controller.

```
$ atom controllers/galleries.go
```

```
func NewGalleries(gs models.GalleryService, r *mux.Router) *Galleries {
    return &Galleries{
        New:           views.NewView("bootstrap", "galleries/new"),
        ShowView:      views.NewView("bootstrap", "galleries/show"),
        EditView:      views.NewView("bootstrap", "galleries/edit"),
        IndexView:     views.NewView("bootstrap", "galleries/index"),
        gs:            gs,
        r:             r,
    }
}

type Galleries struct {
    New           *views.View
    ShowView      *views.View
    EditView      *views.View
    IndexView     *views.View
    gs            models.GalleryService
    r             *mux.Router
}
```

Once we have the view ready, we are going to add a new method to our Galleries type. The method will be named `Index`, and will be used to retrieve a user's galleries before rendering the `IndexView`.

In order to retrieve all of a user's galleries, we will first retrieve the current user using the request context and our context package. After we have the user we

will query for all of that user's galleries using the `ByUserID` method we added to the `GalleryService` earlier. Once we have verified there weren't any errors with our query we will build the data for our view and render it.

```
// GET /galleries
func (g *Galleries) Index(w http.ResponseWriter, r *http.Request) {
    user := context.User(r.Context())
    galleries, err := g.gs.ByUserID(user.ID)
    if err != nil {
        http.Error(w, "Something went wrong.", http.StatusInternalServerError)
        return
    }
    var vd views.Data
    vd.Yield = galleries
    g.IndexView.Render(w, vd)
}
```

Note: Our view will just be a big data dump for now because we haven't implemented it fully, but we will fix that in the next section.

Finally, we need to update our `main.go` source file to inform our router of the new page we just added.

```
$ atom main.go
```

Add the following route where all the existing gallery routes are.

```
r.Handle("/galleries",
    requireUserMw.ApplyFn(galleriesC.Index)).Methods("GET")
```

Restart your server and head to the `localhost:3000/galleries` page. While the page won't look very pretty, we can see that our view is indeed receiving a slice of galleries. In the next section we will learn how to iterate over the elements of a slice inside a template, allowing us to render each gallery individually and making our index page more usable.

Source code

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

Completed - book.usegolang.com/14.10.2

Changes - book.usegolang.com/14.10.2-diff

14.10.3 Iterating over slices in Go templates

We now have all the code in place to render our index page, but our template could use some updating. Rather than just dumping all that data on the screen, it would be nice if we could instead iterate over each gallery, possibly displaying each as a row in a table.

We can achieve this with Go's html/template package by using the **range** action⁷, which at a high level is similar to the **range** keyword you have seen in Go code, but isn't exactly the same.

An example of iterating over a slice in a Go template is shown below, assuming that the **.Slice** field storing a slice.

```
 {{range .Slice}}
   Item: {{.}} <br/>
 {{end}}
```

Inside the range block the dot (.) is replaced with an individual element from the slice, and the range block is run once for each item in the slice. That means that while our code above might look like it is only writing a single item, it is similar to a for loop and will be run once for each item in the slice.

⁷<https://golang.org/pkg/text/template/#hdr-Actions>

For example, if we ran the code above with the **slice** field set to the integer slice:

```
data.slice = []int{11, 16, 22}
```

The output of our template would be:

```
Item: 11 <br/>
Item: 16 <br/>
Item: 22 <br/>
```

The range keyword allows us to write code representing how a single item in a slice should look, and then it handles running that portion of the template for each item in the slice.

We are going to update our template to take advantage of the range keyword, but first we are going to setup an HTML table using Bootstrap's hover row⁸ feature.

Open up your index template to get started.

```
$ atom views/galleries/index.gohtml
```

We will start by defining our bootstrap row and colum, and then inside of these we are going to add a table with the "table-hover" class. We will also add some headers to our table, a row of fake data that we will replace later, and a link to the new gallery page.

```
{{define "yield"}}
<div class="row">
  <div class="col-md-12">
```

⁸<https://getbootstrap.com/docs/3.3/css/#tables-hover-rows>

```


| ID | Title | View | Edit |
|----|-------|------|------|
|----|-------|------|------|


  New Gallery


```

Note: The above is all pure HTML and Bootstrap, and isn't specific to Go at all. If you are having trouble with it I suggest checking out some resources on HTML and the Bootstrap docs.

We want to replace the fake row we have with the current user's galleries. Those galleries can be accessed via a slice stored in the dot (.) inside of our template, so we would first start by adding a range.

```

{{range .}}
<tr>
  <th scope="row">123</th>

```

```
<td>Fake gallery title</td>
<!-- ... -->
</tr>
{{end}}
```

Inside of our range we can access each individual gallery via the dot (.) because the original dot will be replaced with an individual gallery while the range block is being executed. This can be confusing at first, but just remember that inside of a range block you aren't accessing the same data as outside of it.

Knowing that we can access the individual gallery, we can update our template to create real gallery rows by replacing each piece of faked data with the field it would normally stored in. We need to do this for both the data we display, as well as the data used to generate our hrefs inside the **a** tags.

```
<!-- ... your code above this can remain unchanged -->
<tbody>
  {{range .}}
    <tr>
      <th scope="row">{{.ID}}</th>
      <td>{{.Title}}</td>
      <td>
        <a href="/galleries/{{.ID}}">
          View
        </a>
      </td>
      <td>
        <a href="/galleries/{{.ID}}/edit">
          Edit
        </a>
      </td>
    </tr>
  {{end}}
</tbody>
<!-- ... your code below this can remain unchanged -->
```

*Note: Be sure to replace the IDs in the **** links as well.*

Save your view and restart your server and go check out our new, much more usable index page. Verify that all the links work as expected before proceeding.

[H]

LensLocked.com

localhost:3000/galleries

LensLocked.com Home Contact Log In Sign Up

ID	Title	View	Edit
3	Company Outings	View	Edit
6	Demo Gallery	View	Edit
7	Dogs and Cats	View	Edit
8	Bad Sports Teams	View	Edit

New Gallery

Copyright 2016 lenslocked.com

Figure 14.4: Galleries index page

Source code

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

Completed - book.usegolang.com/14.10.3

Changes - book.usegolang.com/14.10.3-diff

14.11 Improving the user experience and cleaning up

Our last section of this chapter will focus not on adding new features, but instead on improving the user experience for the ones that we have and cleaning up our code a bit.

While it might be possible to create new galleries, view them on the index page, and edit/delete them, none of this is very intuitive and we could improve our application a bit. For example, we don't even have a link to our galleries in the navigation bar, so without knowing the proper URL a user could never create a gallery. We also don't want to render the galleries link unless a user is signed in, as this page doesn't work for users who aren't signed in.

In this section we will update our code to:

1. Clean up a few redirects to make the process of creating, editing, and deleting galleries more intuitive for users.
2. Only render certain navigation bar links when a user is signed in.

14.11.1 Adding intuitive redirects

When a user creates or deletes a gallery we currently respond by redirecting them to the show gallery page, or rendering a text “success” message. While these two responses aren’t invalid, they aren’t very useful. Chances are after a user creates a new gallery they aren’t going to want to edit the gallery, but would instead like to upload new images and otherwise edit the gallery.

Likewise, when a user deletes a gallery a plain-text “success” message isn’t very helpful. It would make more sense to redirect the user to the galleries index, allowing them to continue working with their remaining galleries.

In this section we will once again use gorilla/mux’s named routes feature to make both of these happen. We will start by adding constants for our route names and then updating our router to use these names.

```
$ atom controllers/galleries.go
```

We will need a named route for both the index and edit gallery pages.

```
const (
    IndexGalleries = "index_galleries"
    ShowGallery   = "show_gallery"
    EditGallery   = "edit_gallery"
)
```

Next we will update our router to use these names with the corresponding routes.

```
$ atom main.go
```

Update the following two routes in your main function. We are adding the last part - the **Name** method call - to each of the routes in order to give it a name.

```
// Don't forget the period at the end of some of these lines
r.Handle("/galleries",
    requireUserMw.ApplyFn(galleriesC.Index)).
    Methods("GET").
    Name(controllers.IndexGalleries)

r.HandleFunc("/galleries/{id:[0-9]+}/edit",
    requireUserMw.ApplyFn(galleriesC.Edit)).
    Methods("GET").
    Name(controllers.EditGallery)
```

Note: Don't forget the trailing periods at the end of a few lines.

Box 14.2. Why do we need trailing periods?

Go doesn't require semicolons, but because of how the compiler inserts these semi-colons for you it can force us to write our code in a specific manner. For example, you can't place your curly braces on a newline after an if statement.

```
// This DOES NOT work
if a < b
{
    // do stuff
}
```

Similarly, we need to end chained method or field references that wrap to a new line with a period (.) indicating that we aren't done with our line of code.

If we were instead to write our code with the period on the new line (shown below), it would end up resulting in a compilation error much like one we would see if we ended the first line with a semicolon.

```
// This doesn't work
r.Handle("/", someHandler)
    .Methods("GET")

// This is roughly the same as the code above in the
// compiler's eyes. Notice the semicolon at the end of the
// first line
r.Handle("/", someHandler);
    .Methods("GET")
```

The simplest solution to this is to place the period at the end of the previous line, informing our compiler that we aren't done with that expression just yet.

```
// Notice the period at the end.
r.Handle("/", someHandler).
    Methods("GET")
```

With our routes now named we can head back to our galleries controller and update our handlers to redirect to the proper pages. We'll start with the Create method, updating it to instead redirect us to the edit gallery page.

```
$ atom controllers/galleries.go
```

```
func (g *Galleries) Create(w http.ResponseWriter, r *http.Request) {
    // ... most of this code is unchanged

    // Find the line below:
    // url, err := g.r.Get>ShowGallery).URL("id",
    //     strconv.Itoa(int(gallery.ID)))
    // And update it to instead use the EditGallery route.
    url, err := g.r.Get>EditGallery).URL("id",
        strconv.Itoa(int(gallery.ID)))

    // ... the rest of our code is unchanged
}
```

Next head to the Delete method, where we will be removing the `Fprintln` function call code and instead redirecting users to the index of all their galleries after they delete one.

```
func (g *Galleries) Delete(w http.ResponseWriter, r *http.Request) {
    // ... this is all unchanged

    // Delete this line:
    // fmt.Fprintln(w, "successfully deleted!")
    // And replace it with the following
    url, err := g.r.Get(IndexGalleries).URL()
    if err != nil {
        http.Redirect(w, r, "/", http.StatusFound)
        return
    }
    http.Redirect(w, r, url.Path, http.StatusFound)
}
```

Note: You may need to remove the `fmt` package from the imports as well.

Finally, we are going to add a redirect to the users controller, redirecting users to the galleries index after they have signed up for a new account or signed in.

```
$ atom controllers/users.go
```

For simplicity, we are going to hard-code the routes for this controller. If you wanted you could instead add the router to the `Users` type and then use gorilla/mux's named routes just like we did for the `Galleries` type. We aren't doing that here because these routes are easy to create (they don't have any variables in them), and they don't tend to change frequently in practice.

Note: This is mostly a judgment call, so what you end up doing is up to you.

```
func (u *Users) Create(w http.ResponseWriter, r *http.Request) {
    // ... nothing changes here except the last line
    http.Redirect(w, r, "/galleries", http.StatusFound)
}
```

```
func (u *Users) Login(w http.ResponseWriter, r *http.Request) {
    // ... nothing changes here except the last line
    http.Redirect(w, r, "/galleries", http.StatusFound)
}
```

Restart your server and experiment with creating and deleting galleries. Also verify that your changes to the login and sign up forms worked.

Aren't those new redirects useful? Now we just need to add a link to our galleries index to the navigation bar when a user is signed in.

Source code

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

Completed - book.usegolang.com/14.11.1

Changes - book.usegolang.com/14.11.1-diff

14.11.2 Navigation for signed in users

The entire goal of this section is going to be to update our navbar template to add the following code to it:

```
{{if .User}}
<li><a href="/galleries">Galleries</a></li>
{{end}}
```

In the code we check to see if there is a User field present, and if so we show a link to the galleries index.

While this might appear simple at first, adding this feature is actually going to require editing several different source files, creating a new middleware, and learning how to can apply a middleware to all of our routes, not just specific ones.

Our current code has a single middleware - the `RequireUser` middleware - and in it we look up the current user by their remember token, assign it to the request context, and redirect the user if they are not signed in. While this has served us well so far, we are now seeing that we have instances where we want to see if a user is signed in, but we don't necessarily want to redirect them if they aren't logged in.

The best way to make this happen in our code is to break our existing middleware into two separate middleware. In the first, we will look up the current user via their remember token cookie and, assuming the user is signed in, store their user model in the request context. This middleware won't redirect the user or try to enforce that a user is present, but is instead used to just lookup data and set it on the request context. In the second middleware we will handle verifying that a user is logged in, and if not we will redirect them to the login page.

Let's start by writing the first, the middleware used simply to look up a signed in user.

```
$ atom middleware/require_user.go
```

```
// User middleware will lookup the current user via their
// remember_token cookie using the UserService. If the user
// is found, they will be set on the request context.
// Regardless, the next handler is always called.
type User struct {
    models.UserService
}

func (mw *User) Apply(next http.Handler) http.HandlerFunc {
    return mw.ApplyFn(next.ServeHTTP)
}

func (mw *User) ApplyFn(next http.HandlerFunc) http.HandlerFunc {
```

```

return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
    cookie, err := r.Cookie("remember_token")
    if err != nil {
        next(w, r)
        return
    }
    user, err := mw.UserService.ByRemember(cookie.Value)
    if err != nil {
        next(w, r)
        return
    }
    ctx := r.Context()
    ctx = context.WithUser(ctx, user)
    r = r.WithContext(ctx)
    next(w, r)
})
}

```

All of this code should look familiar, because it is a subset of the code we used for our `RequireUser` middleware. The only real difference is that we remove the two lines of code that would redirect a user if they weren't signed in.

Now we can move on to writing the new version of our `RequireUser` middleware. We know that if we run the `User` middleware first that a user will already be set in the request context, so we can drastically simplify our code. We no longer need to look up the user in our database, but can instead simply check to see if a user was ever assigned in the request context. If not, we can assume the user isn't signed in and redirect them to the login page.

```

// RequireUser will redirect a user to the /login page
// if they are not logged in. This middleware assumes
// that User middleware has already been run, otherwise
// it will always redirect users.
type RequireUser struct{}

func (mw *RequireUser) Apply(next http.Handler) http.HandlerFunc {
    return mw.ApplyFn(next.ServeHTTP)
}

func (mw *RequireUser) ApplyFn(next http.HandlerFunc) http.HandlerFunc {
    return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
        user := context.User(r.Context())
        if user == nil {

```

```

    http.Redirect(w, r, "/login", http.StatusFound)
    return
}
next(w, r)
})
}

```

We have already applied the `RequireUser` middleware to the routes that need it, but we haven't applied the `User` middleware yet. Rather than picking and choosing which routes need it, we are instead going to apply this middleware to *all* pages. That way when we render our navigation bar we can decide whether to show the galleries link regardless of what page the user is visiting.

```
$ atom main.go
```

First let's update our code used to setup our middleware.

```

userMw := middleware.User{
    UserService: services.User,
}
requireUserMw := middleware.RequireUser{}
// The line above is the same as:
// var requireUserMw middleware.RequireUser

```

After that we need to apply our `userMw` to every route. The best way to do this is to have the middleware run *before* our router even runs. That is, we are going to apply our middleware to our router, and then pass the resulting handler into our `ListenAndServe` function call.

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

Now our `userMw` will be run before our router even routes a user to the appropriate page, guaranteeing that the user is set in the request context if they are logged in.

The next piece of code we need to update is inside of our `views` package. Whenever we render our templates, they don't currently have access to the `User` field. We are going to add this field to the `Data` type, and then update our Render method to always make sure it gets set if a user is signed in.

We'll start with the update to the Data type.

```
$ atom views/data.go
```

All we are doing here is adding the User field to the Data type.

```
import "lenslocked.com/models"

type Data struct {
    Alert *Alert
    User  *models.User
    Yield interface{}
}
```

Now we need to update our views so that they always set this field for us. We could manually do it, but that would be incredibly tedious. It would be much simpler if our Render method did that for us.

Luckily we can do this, but it requires us to update the Render method to take a third argument - the HTTP request object passed into all of our handlers.

```
$ atom views/view.go
```

```
func (v *View) Render(w http.ResponseWriter,
    r *http.Request, data interface{}) {
    // Now we can get access to the user via the r.Context()
}
```

We are going to need to tweak some of the existing code, but really all we are doing is verifying that the data passed in is of the `Data` type much like we were

before, and then assigning it to a new variable - the **vd** variable. After that we set the User field of the **vd** variable using the request context and the context package we created earlier. We then pass the updated data to our template as we execute it.

```

import (
    // You need to add this import
    "lenslocked.com/context"
)

func (v *View) Render(w http.ResponseWriter, r *http.Request, data interface{}) {
    w.Header().Set("Content-Type", "text/html")
    var vd Data
    switch d := data.(type) {
    case Data:
        // We need to do this so we can access the data in a var
        // with the type Data.
        vd = d
    default:
        // If the data IS NOT of the type Data, we create one
        // and set the data to the Yield field like before.
        vd = Data{
            Yield: data,
        }
    }
    // Lookup and set the user to the User field
    vd.User = context.User(r.Context())
    var buf bytes.Buffer
    err := v.Template.ExecuteTemplate(&buf, v.Layout, vd)
    if err != nil {
        http.Error(w, "Something went wrong. If the problem "+
            "persists, please email support@lenslocked.com",
            http.StatusInternalServerError)
        return
    }
    io.Copy(w, &buf)
}

```

The change we made to the Render method will now be causing a few compilation errors because we were only passing in two arguments before. We first need to update the ServeHTTP method in the views package.

```
func (v *View) ServeHTTP(w http.ResponseWriter, r *http.Request) {
    // Add the new argument - r - here.
    v.Render(w, r, nil)
}
```

At this point you are also going to need to update several lines of code in your controllers. The easiest way to find them is to try to compile, find the line number and source file in the compilation error, and then update the corresponding code that calls the Render method. There is also a list of places that need updated below to help with the updates.

```
$ atom controllers/galleries.go
```

```
// In the Index method:
g.IndexView.Render(w, r, vd)

// In the Show method:
g.ShowView.Render(w, r, vd)

// In the Edit method:
g.EditView.Render(w, r, vd)

// In the Update method, inside an if block:
g.EditView.Render(w, r, vd)
// In the Update method, at the end:
g.EditView.Render(w, r, vd)

// In the Create method, inside an if block:
g.New.Render(w, r, vd)
// In the Create method, inside a second if block:
g.New.Render(w, r, vd)

// In the Delete method, inside an if block:
g.EditView.Render(w, r, vd)
```

```
$ atom controllers/users.go
```

```
// In the New method:  
u.NewView.Render(w, r, nil)  
// Technically you don't need the New handler and could just  
// use the NewView as the handler when declaring the route.  
  
// In the Create method, inside an if block:  
u.NewView.Render(w, r, vd)  
// In the Create method, inside a second if block:  
u.NewView.Render(w, r, vd)  
  
// In the Login method, inside an if block:  
u.LoginView.Render(w, r, vd)  
// In the Login method, inside a second if block:  
u.LoginView.Render(w, r, vd)  
// In the Login method, inside a third if block:  
u.LoginView.Render(w, r, vd)
```

We are finally ready to update our templates. We'll start by updating the bootstrap layout template, telling it to pass all the data it has into the navbar template.

```
$ atom views/layouts/bootstrap.gohtml
```

Find the line where we call the navbar template and update it to add the dot (.) as the last argument.

```
{{template "navbar" .}}
```

Now we need to update our navbar template to display the galleries link if the User field is set.

```
$ atom views/layouts/navbar.gohtml
```

Find the **ul** with the home and contact page links. Update that code with the following HTML.

```
<ul class="nav navbar-nav">
  <li><a href="/">Home</a></li>
  <li><a href="/contact">Contact</a></li>
  {{if .User}}
    <li><a href="/galleries">Galleries</a></li>
  {{end}}
</ul>
```

Restart your server and head back to any page. We now have a galleries link as long as we are logged in. After deleting our cookies it will go away because we are no longer signed in.

Note: We do not have a logout button yet, so you will need to manually delete cookies if you want to test this. I suggest the EditThisCookie plugin for Chrome for development for this reason.

Source code

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

Completed - book.usegolang.com/14.11.2

Changes - book.usegolang.com/14.11.2-diff

Chapter 15

Adding images to galleries

Next on our agenda is adding images to our galleries. This will entail adding new forms, handler functions to process those forms, and even adding a new service to our models package.

What makes the image resource unique is that it will give us a chance to look at a resource that doesn't necessarily map to an item in our database, but instead maps to an images stored on our hard drive (or possibly another service like Amazon S3 in the future). Many developers, even experienced ones, allow themselves to start believing that each model maps directly to an item in the database because this is often the case. While models will often have some data, even if just metadata, stored in the database, you can introduce a new model to an application that doesn't interact with the database at all.

In this chapter we are going to be doing just that - adding an image resource that isn't persisted in our database at all. Instead, we will be storing images uploaded by end users on our hard drive and learning how to lookup and serve those images by interacting directly with our file system.

15.1 Image upload form

The first piece of code we are going to introduce is a form to upload new images. In the past we have almost always stuck with a single form for every view we create, but for images this doesn't make as much sense. Our users aren't going to be uploading images on their own, but are instead going to be adding them to a specific gallery.

With that in mind, we are going to add the image upload form to the edit gallery template, and then later when we introduce a way to delete images from a gallery we will also handle that on the edit gallery page. This will enable our users to manage everything gallery related from a single page.

We will be working inside the edit gallery template, so we can start by opening that source file.

```
$ atom views/galleries/edit.gohtml
```

The first change we are going to make is introducing the new form for uploading images.

```
{{define "uploadImageForm"}}
<form action="/galleries/{{.ID}}/images" method="POST"
    enctype="multipart/form-data" class="form-horizontal">
    <div class="form-group">
        <label for="images" class="col-md-1 control-label">Add Images</label>
        <div class="col-md-10">
            <input type="file" multiple="multiple" id="images" name="images">
            <p class="help-block">Please only use jpg, jpeg, and png.</p>
            <button type="submit" class="btn btn-default">Upload</button>
        </div>
    </div>
</form>
{{end}}
```

In creating the input HTML for a file it is often used to reference the file upload

example in the bootstrap docs¹. One of the few things we did add to the input field is the **multiple** attribute, which is our way of telling the browser that a user can select multiple images instead of just one. This is handy because users will frequently be uploading many images to a gallery.

The other thing worth noting is that this HTML form is using the “form-horizontal” class², which is another style of form provided by bootstrap that looks a little different. Most notably, the name of each input is displayed to the left of the input. We will be using this to try to make our form a little easier for users to follow and use.

We don’t want our existing forms to look out of place, so we also need to go back and edit them to match the new form-horizontal style. While doing this we will also break the edit form and the delete button apart into two separate templates so we can render them wherever we want on the page moving forward.

```

{{define "editGalleryForm"}}
<form action="/galleries/{{.ID}}/update" method="POST"
    class="form-horizontal">
    <div class="form-group">
        <label for="title" class="col-md-1 control-label">Title</label>
        <div class="col-md-10">
            <input type="text" name="title" class="form-control" id="title"
                placeholder="What is the title of your gallery?" value="{{.Title}}">
        </div>
        <div class="col-md-1">
            <button type="submit" class="btn btn-default">Save</button>
        </div>
    </div>
</form>
{{end}}


{{define "deleteGalleryForm"}}
<form action="/galleries/{{.ID}}/delete" method="POST"
    class="form-horizontal">
    <div class="form-group">
        <div class="col-md-10 col-md-offset-1">
            <button type="submit" class="btn btn-danger">Delete</button>
        </div>
    </div>
</form>

```

¹<https://getbootstrap.com/docs/3.3/css/#forms>

²<https://getbootstrap.com/docs/3.3/css/#forms-horizontal>

```
</form>
{{end}}
```

Finally we are going to update the “yield” template. We previously had our form limited to 6 columns wide on a medium or larger screen, but with our new design we are going to bump that up to use the entire screen and we will be breaking our content into a few sections using Bootstrap **rows**.

Our new sections will start with the edit gallery section where we will provide a form used to edit a gallery. After that we will display our gallery’s existing images (or filler text for now). The third section will be for uploading new images, and then finally we will have the “dangerous buttons” section which includes a button to delete the gallery.

```
{{define "yield"}}
<div class="row">
  <div class="col-md-10 col-md-offset-1">
    <h2>Edit your gallery</h2>
    <hr>
  </div>
  <div class="col-md-12">
    {{template "editGalleryForm" .}}
  </div>
</div>
<div class="row">
  <div class="col-md-1">
    <label class="control-label pull-right">
      Images
    </label>
  </div>
  <div class="col-md-10">
    {{template "galleryImages" .}}
  </div>
</div>
<div class="row">
  <div class="col-md-12">
    {{template "uploadImageForm" .}}
  </div>
</div>
<div class="row">
  <div class="col-md-10 col-md-offset-1">
    <h3>Dangerous buttons...</h3>
    <hr>
```

```
</div>
<div class="col-md-12">
  {{template "deleteGalleryForm" .}}
</div>
</div>
{{end}}
```

We referenced a “galleryImages” template above but haven’t added that yet, so we can do that next. This will change in a few sections to include small previews of each image in the gallery and a way to delete an image from the gallery, but for now we’ll just use filler text.

```
{{define "galleryImages"}}
<p>Coming soon...</p>
{{end}}
```

We don’t have any code in place to process our image upload form, but if we restart our server our edit gallery page should look a lot more functional than it did before.

Note: If you don’t like the look of this page you are welcome to redesign it. Just be sure to keep similar forms, otherwise our handlers might not work in the next few sections.

Source code

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

Completed - book.usegolang.com/15.1

Changes - book.usegolang.com/15.1-diff

[H]

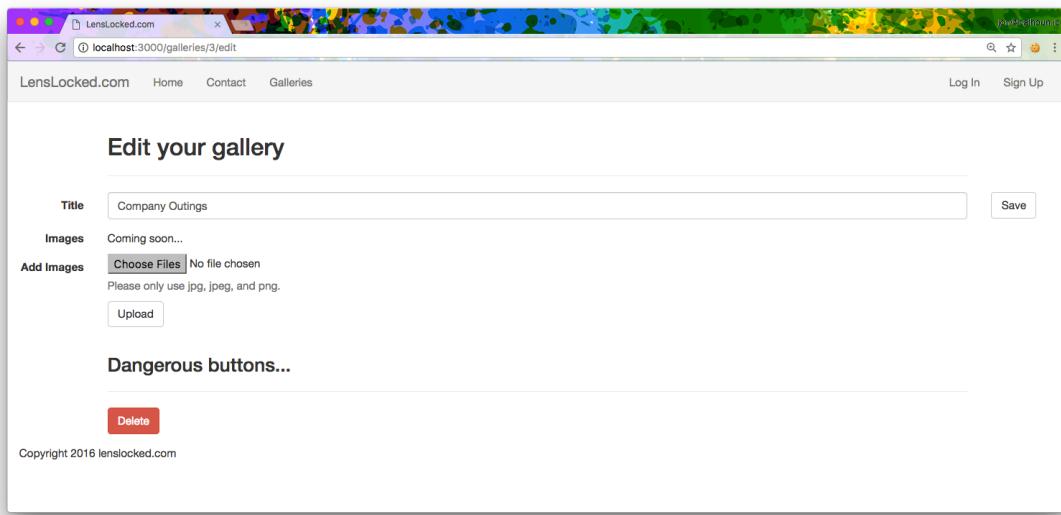


Figure 15.1: Edit gallery page

15.2 Processing image uploads

In this section we are going to focus on handling image uploads for our galleries. We will be writing most of this code directly inside of a controller action, and then in the next section we will refactor our code a bit to move relevant parts to an image service. We also won't be writing any code to display these images yet, but will in a future section.

Typically we would introduce a new controller for our images, but in this particular instance we aren't going to. Whenever users upload images, those images are always going to be associated with a gallery because images are a sub-resource of the gallery. Without a gallery, a user cannot upload a new image. That is why the route we will post image uploads to has the gallery ID in it.

```
POST /galleries/:id/images
```

In addition to knowing that images are a sub-resource to galleries, we also know that our images aren't going to need very many actions. The only things a user can really perform with an image is uploading a new one, or deleting an existing one.

With all of this in mind, it makes more sense to add the handler for our image uploads to the galleries controller. That way we have access to all of the gallery specific methods, like looking up a gallery via an ID stored in the path, without needing to recreate them.

Note: Like many design decisions, this one is based purely on preference and experience. You might find that you prefer a completely different design for all of your handler functions.

Let's start off by stubbing out our handler function and creating a route for it. We'll start with the stubbed method.

```
$ atom controllers/galleries.go
```

```
// POST /galleries/:id/images
func (g *Galleries) ImageUpload(w http.ResponseWriter, r *http.Request) {
    // TODO: Implement this
}
```

And then we will declare a route for it.

```
$ atom main.go
```

```
r.HandleFunc("/galleries/{id:[0-9]+}/images",
    requireUserMw.ApplyFn(galleriesC.ImageUpload)).
    Methods("POST")
```

Now that we have the groundwork laid, we need to figure out how we are going to process image uploads. To start, we are going to write code used to retrieve the current gallery using the ID portion of the path and verify that the current user has permission to edit that gallery. This code could even be copied from the Upload method if you wanted to avoid retyping it.

```
$ atom controllers/galleries.go
```

```
func (g *Galleries) ImageUpload(w http.ResponseWriter, r *http.Request) {
    gallery, err := g.galleryByID(w, r)
    if err != nil {
        return
    }
    user := context.User(r.Context())
    if gallery.UserID != user.ID {
        http.Error(w, "Gallery not found", http.StatusNotFound)
        return
    }
    // TODO: Finish implementing this
}
```

Next we want to parse our form. Unlike our past forms, this form is encoded as a multipart form, which can be inferred from the **enctype** attribute of the **form** html tag.

```
<form action="/galleries/{{.ID}}/images" method="POST"
    enctype="multipart/form-data" class="form-horizontal">
    ^^^^^^^^^^^^^^^^^^^^^^
    <!-- This is the encoding type -->
```

Note: This code snippet is for illustration purposes; you have already coded it.

Go's **net/http** package provides us with a the ParseMultipartForm method³ that can be used to parse forms encoded as multipart/form-data.

Note: Be sure to take a moment to read over the docs for the ParseMultipartForm method. It is always a good idea to read the docs for methods you will be using in your code so you fully understand them.

The ParseMultipartForm method takes one argument - maxMemory - which tells our code the maximum number of bytes of any files to store in memory. If files that exceed this size are uploaded the remainder will be stored on disk (the hard drive) in temporary files that we can retrieve.

We need to decide what our max memory is going to be and store that in a variable or constant. The higher the number you use, the more memory your application will use, but lower numbers mean you will need to read from disk more frequently. In this case we are just going to use 1 megabyte (1048576 bytes), but you could always change it later if you needed to. We will be storing this as a constant.

```
const (
    maxMultipartMem = 1 << 20 // 1 megabyte
)
```

Box 15.1. What is a bit shift?

A bit shift is an operation in which you take an existing number and shift the binary bits used to represent it right or left. In Go this is done with the `<<` and `>>` operators, which represent a left shift and right shift respectively.

Bit shifts are often easier to understand with an example, so imagine we started with the number **5**. In binary this number would be represented as **101**. Now what would happen if we were to shift the number left or right?

³<https://golang.org/pkg/net/http/#Request.ParseMultipartForm>

```
5 << 1
```

When we shift left our code takes the binary representation of 5 - **101** - and moved it “left” inserting zeros where digits are missing. This gives us a new number of **1010** in binary, which is 10 in decimal.

More generally speaking, a left bit shift will double a numbers value, while a right bit shift will divide it in half.

When we do **1<<20** we are saying shift the number 1 left 20 times, which is the same as **2^20**. This number is also equal to exactly one megabyte.

Now we can call the ParseMultipartForm method in our code. We will also want to check for errors and render them if we run into any.

```
func (g *Galleries) ImageUpload(w http.ResponseWriter, r *http.Request) {
    // ... existing code remains unchanged

    var vd views.Data
    vd.Yield = gallery
    err = r.ParseMultipartForm(maxMultipartMem)
    if err != nil {
        // If we can't parse the form just render an error alert on the
        // edit gallery page.
        vd.SetAlert(err)
        g.EditView.Render(w, r, vd)
        return
    }
}
```

Note: Our current code will have a bug where new images won’t be added to the gallery, so when we render the gallery page again we won’t see newly uploaded images unless we navigate back to the edit page and reload it. We will address this bug in the next two chapters.

After this we would like to iterate over all the uploaded files and store them somewhere, so we need to create a directory to store our files. Rather than keeping all of our images in a single directory, we are going to create a one for each gallery.

Having a unique folder for each gallery will make it easier for us to keep all of our images separated without having to worry about filename collisions. It will also make it easier to find all of the images for a specific gallery; all we would need to do is look up all the images in that gallery's directory.

In order to create a directory we will be using the **MkdirAll** function⁴, which is provided by the **os** package in the standard library. This function will create all of the folders in a path, meaning that if we wanted to create the following folders:

```
folder-a/nested-folder-b/nested-further-c
```

The MkdirAll function would create any of these folders that do not exist. If you are familiar with Linux at all, this is similar to using **mkdir -p**.

MkdirAll also requires a FileMode, which is basically just a set of permissions for the directory. We will be using 0755, which gives the current user permission to do anything with the file while giving others read and execute permission.

Box 15.2. File permissions

File permissions are more common in a Linux environment, but are required for the MkdirAll function regardless of OS. Even if you don't use Linux, they are handy to be aware of because you will likely end up deploying to a Linux server at some point.

⁴<https://golang.org/pkg/os/#MkdirAll>

For more info on file permissions I recommend the following:

- <https://help.ubuntu.com/community/FilePermissions>
- <http://www.grymoire.com/Unix/Permissions.html>

We know we want to use the `MkdirAll` function to create our directory, but we still need to build a string representing the path to our directory. We will do this using the `Join` function⁵ from the `path/filepath` package along with `fmt.Sprintf` to convert our `uint` into a string.

Putting it all together we get the following code:

```
import (
    "fmt"
    "os"
    "path/filepath"
)

func (g *Galleries) ImageUpload(w http.ResponseWriter, r *http.Request) {
    // ... existing code remains unchanged

    // Create the directory to contain our images

    // filepath.Join will return a path like:
    //   images/galleries/123
    // We use filepath.Join instead of building the path
    // manually because the slashes and other characters
    // could vary between operating systems.
    galleryPath := filepath.Join("images", "galleries",
        fmt.Sprintf("%v", gallery.ID))
    // Create our directory (and any necessary parent dirs)
    // using 0755 permissions.
    err = os.MkdirAll(galleryPath, 0755)
    if err != nil {
        // If we get an error, render the edit gallery page again
        vd.SetAlert(err)
    }
}
```

⁵<https://golang.org/pkg/path/filepath/#Join>

```

    g.EditView.Render(w, r, vd)
    return
}
}

```

Next we need to iterate over all the uploaded files and store them in our directory we just created for our gallery. ParseMultipartForm, which we called earlier, will parse the uploaded form for us, but it won't do anything with the files for us. We still need to copy them to their permanent location.

To do this we are going to use the MultipartForm⁶ field of the http.Request. We can't use this prior to calling the ParseMultipartForm, but after calling the parse function we can use the MultipartForm field of a Request to access a multipart.Form⁷.

```
r.MultipartForm // a *multipart.Form
```

The multipart.Form is defined as:

```

type Form struct {
    Value map[string][]string
    File  map[string][]*FileHeader
}

```

Of the two field provided by the multipart.Form, we are going to be using the **File** field. This field stores a map where each key maps to the **name** attribute of the HTML input tag (shown below), and the value of each key is a slice of FileHeaders⁸ which we can use to access each uploaded file.

In our case, the HTML input tag has the name set to "**images**".

⁶<https://golang.org/pkg/net/http/#Request>

⁷<https://golang.org/pkg/mime/multipart/#Form>

⁸<https://golang.org/pkg/mime/multipart/#FileHeader>

```
<input type="file" multiple="multiple" id="images" name="images">
^~~~~~
```

Putting that all together, if we want to access the slice of FileHeaders corresponding to our image uploads we would access them with the following Go code.

```
files := r.MultipartForm.File["images"]
```

We will then iterate over each of our FileHeader values and open the file associated with that FileHeader by using the `Open`⁹ method. This will return us a File¹⁰, and/or an error. If there isn't an error we need to be sure to close the file to prevent memory leaks, so we can defer that call now to make sure it happens. If there is an error we will render the edit page again.

```
func (g *Galleries) ImageUpload(w http.ResponseWriter, r *http.Request) {
    // ... existing code remains unchanged

    // Iterate over uploaded files to process them.
    files := r.MultipartForm.File["images"]
    for _, f := range files {
        // Open the uploaded file
        file, err := f.Open()
        if err != nil {
            vd.SetAlert(err)
            g.EditView.Render(w, r, vd)
            return
        }
        defer file.Close()

        // TODO: Copy the file we just opened to the gallery directory
        // we created.
    }
}
```

⁹<https://golang.org/pkg/mime/multipart/#FileHeader.Open>

¹⁰<https://golang.org/pkg/mime/multipart/#File>

Note: I know this is a lot to take in all at once, but unfortunately there isn't a great way to gradually introduce handling multipart forms.

Now that we have an opened file, we want to copy it to our gallery directory. We can't do that without first creating a file to copy the data over to. To create the file we will use the `os.Create` function¹¹, but before that we will need to use the `galleryPath` we created earlier along with the `Filename` field from our `FileHeader` to build the new file path. We also want to defer closing this file as well to avoid a memory leak.

```
func (g *Galleries) ImageUpload(w http.ResponseWriter, r *http.Request) {
    // ... existing code remains unchanged

    files := r.MultipartForm.File["images"]
    for _, f := range files {
        // ... open the uploaded file with existing code

        // Create a destination file
        dst, err := os.Create(filepath.Join(galleryPath, f.Filename))
        if err != nil {
            vd.SetAlert(err)
            g.EditView.Render(w, r, vd)
            return
        }
        defer dst.Close()
    }
}
```

We are finally ready to copy the data from the uploaded file into the permanent file we are storing on disk. We can do that using the `io.Copy` function¹², which will take in a destination writer and a source reader and copy the data from one to the other. In this case our destination is the new file we just created, and the source is the uploaded file we originally opened.

```
import (
    "io"
)
```

¹¹<https://golang.org/pkg/os/#Create>

¹²<https://golang.org/pkg/io/#Copy>

```

func (g *Galleries) ImageUpload(w http.ResponseWriter, r *http.Request) {
    // ... existing code remains unchanged

    files := r.MultipartForm.File["images"]
    for _, f := range files {
        // ... open the uploaded file with existing code
        // ... create the destination file with existing code

        // Copy uploaded file data to the destination file
        _, err = io.Copy(dst, file)
        if err != nil {
            vd.SetAlert(err)
            g.EditView.Render(w, r, vd)
            return
        }
    }

    vd.Alert = &views.Alert{
        Level:  views.AlertLvlSuccess,
        Message: "Images successfully uploaded!",
    }
    g.EditView.Render(w, r, vd)
}

```

If everything goes according to plan we will print out a success alert and render the edit page again. We can visit the images directory on our computer to verify that the images were written there. It is important to test this code before moving on as a majority of our upcoming code will not work if this portion doesn't work.

Box 15.3. If you are confused...

Before moving on, it is worth noting that this section covered a lot of code that might be unfamiliar to you, especially if you are a new developer and are brand new to Go. Unfortunately I can't go into each of these in great detail, as many of them are deserving of their own mini-book, but what I can tell you is that all of this gets easier with practice.

If you are feeling a bit confused about one of the standard library functions we used or anything else I would first recommend looking at the docs for the corresponding

[H]

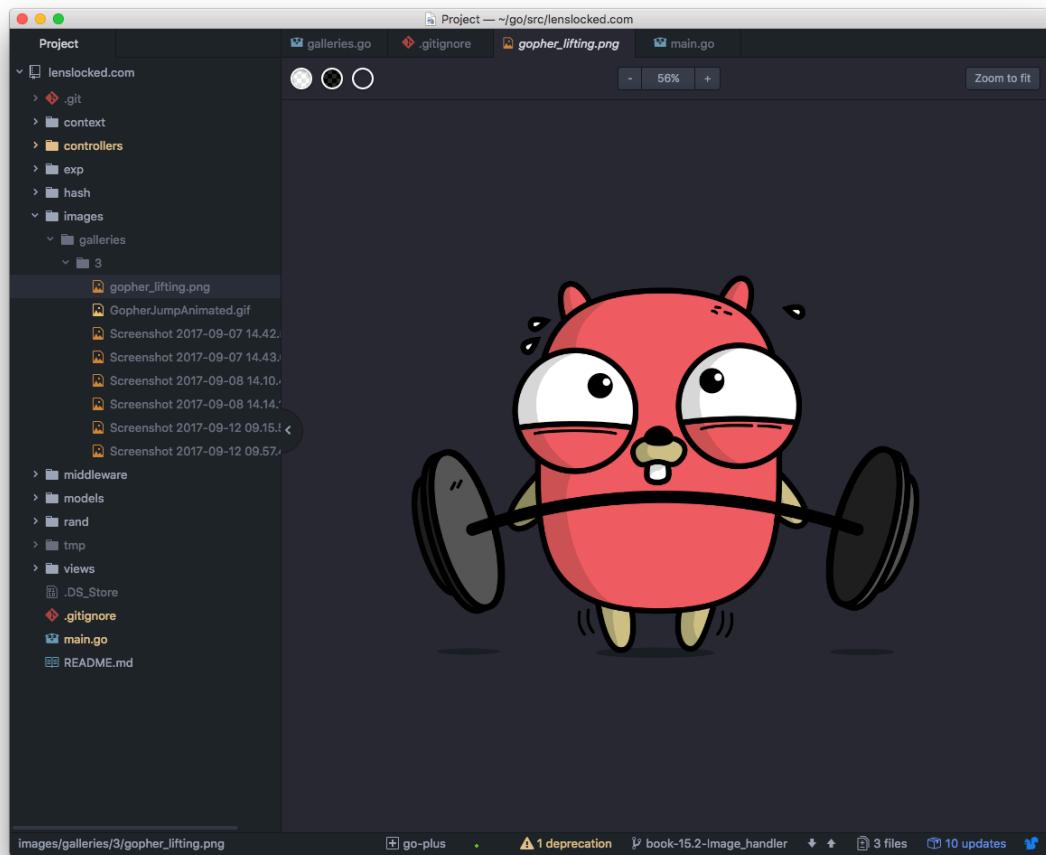


Figure 15.2: An uploaded image

code. Look for examples there as well, as they often help illustrate how a function can be used.

After that, try searching online for relevant resources. Stack overflow and similar sites will often have very great explanations of how a specific function works if you find the right post. There are also many other great sites like [Go by Example](#) that try to show how specific tasks can be completed using Go, and in many of these you will see functions like `io.Copy` being used.

Box 15.4. If you are using git

If you are using git you likely won't want to commit the images we are creating, so I suggest adding the images directory to your `.gitignore` file. This will tell git to not add any files from that directory to your repo.

```
images/**
```

Source code

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

Completed - book.usegolang.com/15.2

Changes - book.usegolang.com/15.2-diff

15.3 Creating the image service

While our code works at it is currently coded, our ImageUpload method is doing quite a bit of work. That was fine when we were just learning how to process image uploads, but now that we have working code we are going to take some time to refactor a bit and introduce the ImageService.

Our image service is going to start with a single method - **Create** - but unlike our other models the create method isn't going to accept an **Image** struct type as its only argument, and is instead going to accept the data needed to create an image.

What data is that? Well, for starters we know we need the gallery ID so we know what directory to create the image in. We also need the filename for the image as well. But what about the data for the file? How will we get that?

To answer that question we are going to look back at the **io.Copy** function¹³ we used previously. We are going to be using it again inside our image service to copy data, so we would like to know what the bare minimum it requires is.

Looking at the source code, it looks like Copy is defined as:

```
func Copy(dst Writer, src Reader) (written int64, err error)
```

The source is of the type **Reader**¹⁴, which is any type with a **Read** method that matches the following definition:

```
type Reader interface {
    Read(p []byte) (n int, err error)
}
```

¹³<https://golang.org/pkg/io/#Copy>

¹⁴<https://golang.org/pkg/io/#Reader>

That means we can define our Create method for the image service using the io.Reader interface. We will be doing this inside of a new source file we will be creating for our image-specific code. It will be inside the models package.

```
$ atom models/images.go
```

```
package models

// We will need some of these imports later
import (
    "fmt"
    "io"
    "os"
    "path/filepath"
)

type ImageService interface {
    Create(galleryID uint, r io.Reader, filename string) error
}
```

While we could make our second argument, the io.Reader, more specific but that generally isn't a great idea. Our Create method doesn't need anything more than a data type with the Read method in order to create a new image, so by using a more specific data type in our Create method we would be writing code that works in fewer use cases with no real benefit.

Now that we know what our ImageService interface is going to look like, let's start implementing it.

```
func NewImageService() ImageService {
    return &imageService{}
}

type imageService struct{}

func (is *imageService) Create(galleryID uint,
    r io.Reader, filename string) error {

    // TODO: Implement this using code we wrote in the ImageUpload handler
    return nil
}
```

The first part of the implementation we are going to work on is the part that creates a directory to store our images in. We already wrote this inside the ImageUpload method, so we will just be moving that code into a method on the imageService.

```
func (is *imageService) mkImagePath(galleryID uint) (string, error) {
    galleryPath := filepath.Join("images", "galleries",
        fmt.Sprintf("%v", galleryID))
    err := os.MkdirAll(galleryPath, 0755)
    if err != nil {
        return "", err
    }
    return galleryPath, nil
}
```

After that we can implement the Create method for the imageService. Nearly all of this code is identical to the code we wrote in the ImageUpload method inside the for loop. We are just moving it to the image service to organize our code a bit better and make it easier to test in the future.

```
func (is *imageService) Create(galleryID uint,
    r io.Reader, filename string) error {
    path, err := is.mkImagePath(galleryID)
    if err != nil {
        return err
    }
    // Create a destination file
    dst, err := os.Create(filepath.Join(path, filename))
    if err != nil {
        return err
    }
    defer dst.Close()
    // Copy reader data to the destination file
    _, err = io.Copy(dst, r)
    if err != nil {
        return err
    }
    return nil
}
```

Now that we have a working Create method, we are going to want to make the

image service accessible inside the Galleries controller so we can use it to create images instead of the existing code.

```
$ atom controllers/galleries.go
```

We'll start with the NewGalleries function and the Galleries type. We need to add an image service field to each of these.

```
func NewGalleries(gs models.GalleryService,
    is models.ImageService, r *mux.Router) *Galleries {
    return &Galleries{
        New:           views.NewView("bootstrap", "galleries/new"),
        ShowView:      views.NewView("bootstrap", "galleries/show"),
        EditView:      views.NewView("bootstrap", "galleries/edit"),
        IndexView:     views.NewView("bootstrap", "galleries/index"),
        gs:            gs,
        is:            is,
        r:             r,
    }
}

type Galleries struct {
    New           *views.View
    ShowView      *views.View
    EditView      *views.View
    IndexView     *views.View
    gs            models.GalleryService
    is            models.ImageService
    r             *mux.Router
}
```

Next we need to update the Services type and its corresponding NewServices function with this new service.

```
$ atom models/services.go
```

```
func NewServices(connectionInfo string) (*Services, error) {
    db, err := gorm.Open("postgres", connectionInfo)
    if err != nil {
        return nil, err
    }
    db.LogMode(true)
    return &Services{
        User:      NewUserService(db),
        Gallery:   NewGalleryService(db),
        Image:     NewImageService(),
        db:        db,
    }, nil
}

type Services struct {
    Gallery GalleryService
    User    UserService
    Image   ImageService
    db      *gorm.DB
}
```

After that we can pass the image service into our NewGalleries function call in our main function.

```
$ atom main.go
```

```
// Remove this line in the main() function:
// galleriesC := controllers.NewGalleries(services.Gallery, r)
// Replace it with this line:
galleriesC := controllers.NewGalleries(services.Gallery, services.Image, r)
```

Finally, we can go back to our ImageUpload handler and have it use the image service instead of managing all the image creation logic on its own. Our handler will still parse the multipart form and open those files because this is very specific to this handler, but once it has a reader for the file it will use the ImageService to create the image.

```
$ atom controllers/galleries.go
```

```
// Note: You need to remove the following imports:
// - "fmt"
// - "io"
// - "os"
// - "path/filepath"

func (g *Galleries) ImageUpload(w http.ResponseWriter, r *http.Request) {
    // ... Lookup the the gallery and parse the form like before.

    // Delete the code used to create the gallery path. It is
    // commented out below.
    // galleryPath := fmt.Sprintf("images/galleries/%v/", gallery.ID)
    // err = os.MkdirAll(galleryPath, 0755)
    // if err != nil {
    //     vd.SetAlert(err)
    //     g.EditView.Render(w, r, vd)
    //     return
    // }

    files := r.MultipartForm.File["images"]
    for _, f := range files {
        // Open the uploaded file
        file, err := f.Open()
        if err != nil {
            vd.SetAlert(err)
            g.EditView.Render(w, r, vd)
            return
        }
        defer file.Close()

        // We replace all of the code from here onwards with a call
        // to the ImageService's Create method.
        // Create the image
        err = g.is.Create(gallery.ID, file, f.Filename)
        if err != nil {
            vd.SetAlert(err)
            g.EditView.Render(w, r, vd)
            return
        }
    }
    // ... render the success alert the same way as before
}
```

Functionally our application won't really be any different, but we have successfully introduced the ImageService and moved the logic specific to creating

images to it. Now if we wanted to change where we saved images, for instance if we wanted to store them on Amazon's S3, we would only need to update the ImageService to make that happen.

Note: Be sure to double check your code with the source below, as this was a reasonably involved refactor.

Source code

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

Completed - book.usegolang.com/15.3

Changes - book.usegolang.com/15.3-diff

15.4 Looking up images by gallery ID

The next thing we want to add to our application is a way to view a gallery's images. In order to do this, we are going to need to do two things:

1. Create a way to lookup images by gallery ID
2. Write code that allows our application to serve static files.

We will be doing the first task in this section, and then the second task in the next section.

When looking up images, we first need to decide what return type to use. In other resources we returned something like a Gallery type, but for images we don't have a type. We could create one, but we probably don't want to do that without a good reason. Instead, we are going to just return a slice of strings that

represents the path that a user can used to access the image. We will also need to check for errors, so we will add an error as the second argument.

```
$ atom models/images.go
```

```
type ImageService interface {
    Create(galleryID uint, r io.Reader, filename string) error
    ByGalleryID(galleryID uint) ([]string, error)
}
```

Next we need to implement the `ByGalleryID` method. To do this, we are going to get the directory for our gallery's images and then glob it like we did for our layout templates. This will give us a list of all files in the directory, which should only contain images that the user has uploaded.

While writing this code we are also going to pull the code used to build the image directory out into its own method so that we can build the path string without trying to create the directory every time.

```
func (is *imageService) ByGalleryID(galleryID uint) ([]string, error) {
    path := is.imagePath(galleryID)
    strings, err := filepath.Glob(filepath.Join(path, "*"))
    if err != nil {
        return nil, err
    }
    return strings, nil
}

// Going to need this when we know it is already made
func (is *imageService) imagePath(galleryID uint) string {
    return filepath.Join("images", "galleries",
        fmt.Sprintf("%v", galleryID))
}

// Use the imagePath method we just made
func (is *imageService) mkImagePath(galleryID uint) (string, error) {
    galleryPath := is.imagePath(galleryID)
    err := os.MkdirAll(galleryPath, 0755)
    if err != nil {
        return "", err
    }
    return galleryPath, nil
}
```

```
    }
    return galleryPath, nil
}
```

Note: I probably should have named these methods `imageDir` and `mkImageDir` instead of `path`, but I am leaving the code as-is to keep it in sync with the screen-casts.

Next we are going to update our Gallery resource to add the `Images` field. In this field we will store the slice of strings that will be used later to build URLs to show the images in our HTML, but for now we will just be displaying the raw text to verify things are working.

```
$ atom models/galleries.go
```

```
type Gallery struct {
    gorm.Model
    UserID uint      `gorm:"not_null;index"`
    Title  string    `gorm:"not_null"`
    Images []string  `gorm:"-"`
}
```

Be sure to include the GORM struct tag which tells GORM not to try to save this data to the database.

Next we are going to update our galleries controller to fill in the `Images` field when we retrieve a specific gallery.

```
$ atom controllers/galleries.go
```

We are going to add the code to our existing `galleryByID` method so that we don't have to manually call it on pages that need images. Generally speaking we only use this method when we are intending to render a single gallery (not

the galleries index page), and we tend to want all of a gallery's images in those situations.

```
func (g *Galleries) galleryByID(w http.ResponseWriter,
    r *http.Request) (*models.Gallery, error) {
    // ... most of this stays the same

    // Add this to the end of the file, right before the existing return
    images, _ := g.is.ByGalleryID(gallery.ID)
    gallery.Images = images
    return gallery, nil
}
```

Now that we have images, even if they are just image paths, let's display them in our edit and show pages to verify everything is working.

```
$ atom views/galleries/edit.gohtml
```

In the edit page we will update the galleryImages template, telling it to range over the images slice and render the string for each as a list item.

```
{{define "galleryImages"}}
<ul>
    {{range .Images}}
        <li>
            {{.}}
        </li>
    {{end}}
</ul>
{{end}}
```

Next is the show page.

```
$ atom views/galleries/show.gohtml
```

In this page we need to find the “Images coming soon...” text. We are going to replace it with the same thing.

```
 {{define "yield"}}
<div class="row">
  <div class="col-md-12">
    <h1>
      {{.Title}}
    </h1>
    <ul>
      {{range .Images}}
        <li>
          {{.}}
        </li>
      {{end}}
    </ul>
  </div>
{{end}}
```

Restart your application and verify that both your edit and show gallery pages now display a list of images paths. For this to work you may need to upload a few images to the gallery first.

Source code

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

Completed - book.usegolang.com/15.4

Changes - book.usegolang.com/15.4-diff

15.5 Serving static files in Go

Before we can show users images in a gallery, we must first implement a way to serve static files with our application. The simplest way to do this is through the use of the FileServer function¹⁵ provided by the net/http package.

The FileServer function accepts a FileSystem¹⁶, which we can construct by converting a directory into an `http.Dir` type¹⁷, and then returns an http.Handler that we can use like we would any other handler.

For example, if we wanted to create a file server using the root directory we might write the following code:

```
fs := http.FileServer(http.Dir("/"))
```

We could then use this handler in our router much like we do the handlers returned by our middleware. In our case we want to create a handler for the images directory that stores our images.

```
imageHandler := http.FileServer(http.Dir("./images/"))
```

Now that we have a file server for our images directory, we want to register it with our router. We are going to send any requests with a path starting with `/images/` to our file server, so we want to start off with gorilla/mux's Path-Prefix method, which allows us to define routes that match a specific prefix.

```
r.PathPrefix("/images/")
```

¹⁵<https://golang.org/pkg/net/http/#FileServer>

¹⁶<https://golang.org/pkg/net/http/#FileSystem>

¹⁷<https://golang.org/pkg/net/http/#Dir>

We then want to add a handler for this prefix, but we can't just use the imageHandler we created. The FileServer handler expects the full path of a web request to match *exactly* with the files in its directory. For example, if a user visited:

```
oursite.com/images/galleries/123/cat.png
```

And we used the PathPrefix code above along with the handler we created:

```
r.PathPrefix("/images/").Handler(imageHandler)
```

Note: Do not code this - we will change it in a second.

The code above would result in our file server starting in the images directory, then looking for another directory inside of it that matches the following:

```
images/galleries/123/cat.png
```

That is, our code would actually be looking for images here:

```
images/images/galleries/123/cat.png  
^^^^^^^^^
```

Notice how this has two images directories?

To prevent this, we are going to use another function offered by the net/http package - the StripPrefix function¹⁸. With this function we can tell it to strip away part of the web requests path before the next handler receives the request, so if we were to instead use the following code:

¹⁸<https://golang.org/pkg/net/http/#StripPrefix>

```
withoutPrefix := http.StripPrefix("/images/", imageHandler)
r.PathPrefix("/images/").Handler(withoutPrefix)
```

What would happen is our StripPrefix code would act similar to our middleware, first stripping the “/images” portion of the path from the web request, then passing the request on to the next handler. In this case that next handler is our imageHandler, which is a file server.

Below is the final code we will be using to create our images route and responding to it with a file server. Add it to the main function near your existing routes.

```
$ atom main.go
```

```
// Image routes
imageHandler := http.FileServer(http.Dir("./images/"))
r.PathPrefix("/images/").Handler(http.StripPrefix("/images/", imageHandler))
```

Next we are going to tweak our ByGalleryID method in our image service. We are going to update it to prefix all paths with a leading slash (/) so that we can use these paths directly in our HTML to build valid **img** tags.

```
$ atom models/images.go
```

```
func (is *imageService) ByGalleryID(galleryID uint) ([]string, error) {
    path := is.imagePath(galleryID)
    strings, err := filepath.Glob(filepath.Join(path, "*"))
    if err != nil {
        return nil, err
    }
    // Add a leading "/" to all image file paths
    for i := range strings {
        strings[i] = "/" + strings[i]
    }
    return strings, nil
}
```

And the last step required to display our images is to update our templates.

```
$ atom views/galleries/edit.gohtml
```

We will be introducing an img tag to our HTML that uses the image string as the source for the image. This is why we wanted to prefix our paths with the slash (/) earlier.

```
{{define "galleryImages"}}
{{range .Images}}
  
{{end}}
{{end}}
```

And similarly, we will be updating our show template to render images.

```
$ atom views/galleries/show.gohtml
```

```
{{define "yield"}}
<div class="row">
  <div class="col-md-12">
    <h1>
      {{.Title}}
    </h1>
    {{range .Images}}
      
    {{end}}
  </div>
</div>
{{end}}
```

Note: We will clean this up so it looks better and displays the images in columns in the next section.

Source code

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

Completed - book.usegolang.com/15.5

Changes - book.usegolang.com/15.5-diff

15.6 Rendering images in columns

Now that we have our images, along with a static file handler that permits us to serve them, we are going to look at how to render our images a little more elegantly. For instance, we likely only want to show smaller versions of the photos when looking at an entire gallery, but we would like to be able to click on an image to view the full size version.

We would also like to be able to split our images into columns, but the actual number of columns is going to vary. For instance, on the edit page the user already knows what the images look like, so chances are we can split them into four columns with slightly smaller preview images, but when we render the gallery for public visitors on the show page we might want larger previews so we can only use three columns.

We will start with the second feature, but will ultimately end up implementing both when we get to editing our HTML templates.

When we are rendering HTML with our templates, we end up filling things in one column at a time. For instance, if we wanted three columns we might have HTML that looks like this:

```
<div class="row">
  <div class="col-md-4">
    <!-- Fill in HTML for the first column -->
```

```
</div>
<div class="col-md-4">
    <!-- Fill in HTML for the second column -->
</div>
<div class="col-md-4">
    <!-- Fill in HTML for the third column -->
</div>
</div>
```

When filling in our columns with images we would like to fill them in in a different order. We would like to start at the first column and add a single image, move on to the second column to add another, then the third column, then finally wrapping back around to the first. For example:

Column 1	Column 2	Column 3
Image 1	Image 2	Image 3
Image 4	Image 5	Image 6
Image 7	Image 8	Image 9
Image 10	Image 11	Image 12

Doing this in our code is a little tricky, because it doesn't match the normal way you might fill in the data. One way to fix this would be to create a new Bootstrap row between each row of images.

```
<div class="row">
    <!-- Images 1, 2, and 3 in cols -->
</div>
<div class="row">
    <!-- Images 4, 5, and 6 in cols -->
</div>
<!-- ... -->
```

The downside to this approach is that it can end up looking bad when images have varying sizes and aspect ratios. You can easily end up with weird spacing between two images in the same column.

In order to prevent this, we are going to add a function to our Gallery model that will allow us to split our images into two dimensional slices. The first slice representing the column we are filling up, while the second dimension represents the image in that column.

This is easier to understand with an example, so let's imagine we had images 1 through 12 in our gallery and we wanted to split them into three columns like in the example above. What we want our function to spit out is the following two dimensional slice.

```
twoDimensionalSlice := [][]string{
    []string{"Image 1", "Image 4", "Image 7", "Image 10"},
    []string{"Image 2", "Image 5", "Image 8", "Image 11"},
    []string{"Image 3", "Image 6", "Image 9", "Image 12"},
}
```

Our resulting slice has two dimensions, hence the double square brackets (`[][]`), and each slice inside of the two dimensional slice has all of the images for a specific column. For example, the second image in the first column could be accessed by calling:

```
// 0 is the column in our final output
// 1 is the image in that column
// Both are zero-based, so this is the first column, second image
twoDimensionalSlice[0][1] // has the value "Image 4"
```

If we were to instead just reference a column we would get an entire slice as a result.

```
twoDimensionalSlice[0]
// This has the value: []string{"Image 1", "Image 4", "Image 7", "Image 10"}
```

In order to write code that splits up our images this way we are going to use what is known as the remainder operator (`%`), which will give us the remainder

after performing integer division. For instance, if we take $5/3$ the result is 1 $2/3$, or in other words it is 1 with a remainder of 2 .

This is useful because we can use it along with the index of the current element in the slice to determine what column it is used for.

```
// % is the remainder operator in Go
// eg:
0 % 3 = 0
1 % 3 = 1
2 % 3 = 2
3 % 3 = 0
4 % 3 = 1
5 % 3 = 2
```

Notice how our remainder rotates from 0 , to 1 , to 2 , then back to 0 , much like we want our images to be placed in column 1 , then 2 , then 3 . We can use this to place our images into the correct slice in our two dimensional slice.

Open up your galleries model. We are going to add a method to our Gallery type which does this so we can use it to render our image columns easier.

Note: Technically speaking it might make sense to split methods like this up and to use template functions or a decorator layer in our code in larger applications, but in my experience the need for that doesn't arise until a project has been around along time so I don't recommend doing that early on.

```
$ atom models/galleries.go
```

```
func (g *Gallery) ImagesSplitN(n int) [][]string {
    // Create out 2D slice
    ret := make([][]string, n)
    // Create the inner slices - we need N of them, and we will
    // start them with a size of 0.
    for i := 0; i < n; i++ {
        ret[i] = make([]string, 0)
    }
    // Iterate over our images, using the index % n to determine
```

```
// which of the slices in ret to add the image to.
for i, img := range g.Images {
    // % is the remainder operator in Go
    // eg:
    //     0%3 = 0
    //     1%3 = 1
    //     2%3 = 2
    //     3%3 = 0
    //     4%3 = 1
    //     5%3 = 2
    bucket := i % n
    ret[bucket] = append(ret[bucket], img)
}
return ret
}
```

Next we are going to alter our HTML to use the new ImageSplitN method to break our images into columns. While we are at it, we will also add some HTML to link to the full sized image along with a link to view a gallery that you are editing.

```
$ atom views/galleries/edit.gohtml
```

We will start with the link to view the gallery you are editing.

```
{{define "yield"}}
<div class="row">
  <div class="col-md-10 col-md-offset-1">
    <h2>Edit your gallery</h2>
    <a href="/galleries/{{.ID}}">
      View this gallery
    </a>
    <hr>
  </div>

  <!-- ... The rest of the yield template remains unchanged -->

</div>
{{end}}
```

Next we will tweak the galleryImages template, altering it to use the ImagesSplitN method. While at it, we will also add a tiny bit of inlined CSS styling to make sure our images aren't rendered too large.

```
 {{define "galleryImages"}}
{{range .ImagesSplitN 6}}
<div class="col-md-2">
  {{range .}}
    <a href="{{.}}>
      
    </a>
  {{end}}
</div>
{{end}}
<!--
Normally styles in the middle of your code is bad,
but for dev I think this is fine. We will fix it up
when we deploy
-->
<style>
  .thumbnail {
    width: 100%;
  }
</style>
{{end}}
```

Note: As stated in the code, styles inline in your code often is frowned upon but we will fix this before we deploy to production.

Next up is the show gallery page, which will have very similar changes.

```
$ atom views/galleries/show.gohtml
```

```
 {{define "yield"}}
<div class="row">
  <div class="col-md-12">
    <h1>
      {{.Title}}
    </h1>
    <hr>
  </div>
</div>
```

```
<div class="row">
  {{range .ImagesSplitN 3}}
    <div class="col-md-4">
      {{range .}}
        <a href="{{.}}>
          
        </a>
      {{end}}
    </div>
  {{end}}
</div>

<!--
Normally styles in the middle of your code is bad,
but for dev I think this is fine. We will fix it up
when we deploy
-->
<style>
  .thumbnail {
    width: 100%;
  }
</style>
{{end}}
```

Restart your server and verify your show gallery and edit gallery pages are rendering images in columns without any styling issues. They should look similar to the screenshot below, but the show page will be slightly different.

Source code

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

Completed - book.usegolang.com/15.6

Changes - book.usegolang.com/15.6-diff

[H]

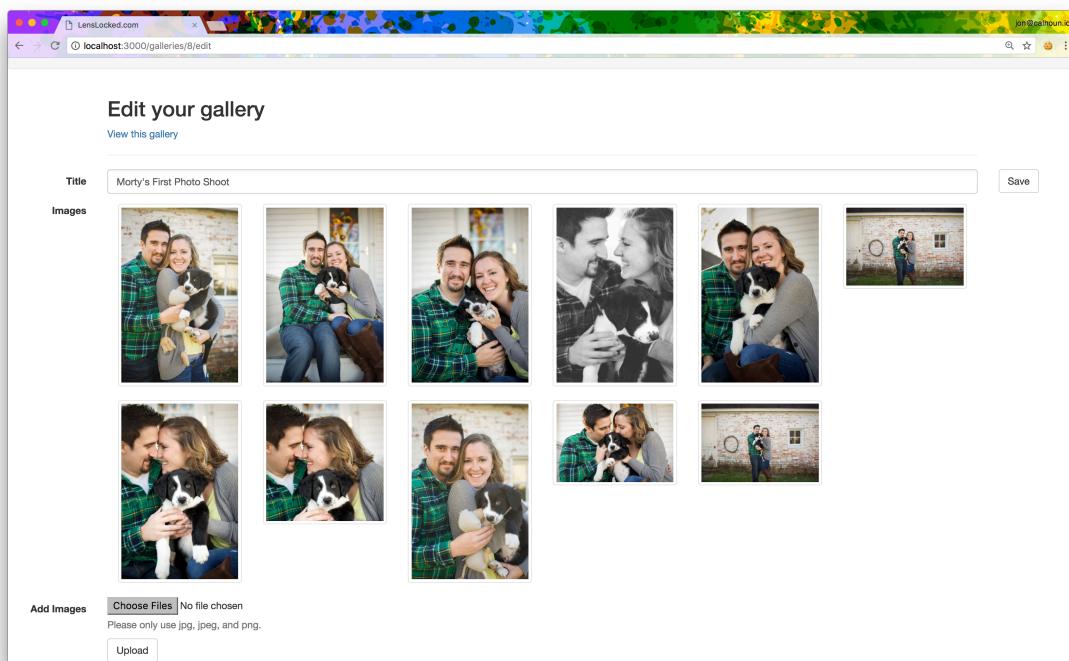


Figure 15.3: Updated edit gallery page

15.7 Letting users delete images

The last feature we are going to add in this chapter is the ability to delete images. It is likely that a user will accidentally upload the wrong image, so we need to provide them with a way to remove images from their gallery while editing it.

In order to implement this feature we are going to first introduce a new type - the Image type - in our models package. The primary reason for doing this is to add a bit more metadata to an image so that we can render our UIs easier. Specifically, we want to be able to reference an image's gallery ID whenever we are rendering the delete image form so that we can create a path like:

```
/galleries/:gallery_id/images/:filename/delete
```

While constructing the Image type we will also split some of the logic we previously wrote regarding images to make our image resource easier to use in the future.

First, open up the images model source file.

```
$ atom models/images.go
```

We are going to start by introducing the Image type. In it we need two fields - the GalleryID and the Filename. With those two we could build the path to the image, build the delete image route, or any others we need.

```
// Image is used to represent images stored in a Gallery.
// Image is NOT stored in the database, and instead
// references data stored on disk.
type Image struct {
    GalleryID uint
    Filename  string
}
```

Next we will add methods to build things like the path we display in the browser, as well as the relative path on our filesystem. I tend to break these into two methods in case we move our images to another directory in the future and may need to change one of the two methods, but not both.

```
// Path is used to build the absolute path used to reference this image
// via a web request.
func (i *Image) Path() string {
    return "/" + i.RelativePath()
}

// RelativePath is used to build the path to this image on our local
// disk, relative to where our Go application is run from.
func (i *Image) RelativePath() string {
    // Convert the gallery ID to a string
    galleryID := fmt.Sprintf("%v", i.GalleryID)
    return filepath.ToSlash(filepath.Join("images", "galleries", galleryID, i.Filename))
}
```

*Note: **ToSlash** is likely only needed in Windows systems, as they use a backslash (\) instead of a forward slash (/) to separate directories and files. The **ToSlash** function works by replacing all forward slashes with backslashes on operating systems that use backslashes.*

Next we will update our existing ByGalleryID method to return a slice of **Images** instead of strings.

```
type ImageService interface {
    Create(galleryID uint, r io.Reader, filename string) error
    ByGalleryID(galleryID uint) ([]Image, error)
}
```

The implementation changes are a little more involved. Instead of iterating over our files and adding the slash (/) prefix, we are now going to be removing the entire path from our string and only retaining the filename. We will do this using the **Base** function¹⁹ that is provided by the path/filepath package. This will return the last element in a path, which in our case will be our filename.

¹⁹<https://golang.org/pkg/path/filepath/#Base>

```
func (is *imageService) ByGalleryID(galleryID uint) ([]Image, error) {
    path := is.imagePath(galleryID)
    strings, err := filepath.Glob(filepath.Join(path, "*"))
    if err != nil {
        return nil, err
    }
    // Setup the Image slice we are returning
    ret := make([]Image, len(strings))
    for i, imgStr := range strings {
        ret[i] = Image{
            Filename: filepath.Base(imgStr),
            GalleryID: galleryID,
        }
    }
    return ret, nil
}
```

Next we need to update the `Gallery` type's `Images` field, along with the `ImagesSplitN` method.

```
$ atom models/galleries.go
```

```
type Gallery struct {
    gorm.Model
    UserID uint      `gorm:"not_null;index"`
    Title  string    `gorm:"not_null"`
    Images []Image   `gorm:"-"`
}

func (g *Gallery) ImagesSplitN(n int) [][]Image {
    ret := make([][]Image, n)
    for i := 0; i < n; i++ {
        ret[i] = make([]Image, 0)
    }
    for i, img := range g.Images {
        bucket := i % n
        ret[bucket] = append(ret[bucket], img)
    }
    return ret
}
```

After updating our models we can move on to updating our HTML templates. They will need a few updates to account for our model changes, as well as

some new HTML uses to render the delete image button which will be inside an HTML form. This will cause a POST web request to be made when the button is clicked and the form is submitted.

We'll start by updating the show gallery template.

```
$ atom views/galleries/show.gohtml
```

Find the div with the images in it and update it to reflect the code below. We now need to call the Path method of an Image in order to get the path and can't just us the dot (.) operator like we were doing before.

```
<div class="row">
  {{range .ImagesSplitN 3}}
    <div class="col-md-4">
      {{range .}}
        <a href="{{.Path}}>
          
        </a>
      {{end}}
    </div>
  {{end}}
</div>
```

Next is the edit gallery tempalte, where we will start by adding the deleteImageForm template.

```
$ atom views/galleries/edit.gohtml
```

Add the following template to the source file:

```
{{define "deleteImageForm"}}
<form action="/galleries/{{.GalleryID}}/images/{{.Filename}}/delete"
  method="POST">
  <button type="submit" class="btn btn-default btn-delete">
```

```

    Delete
  </button>
</form>
{{end}}

```

Then update our galleryImages template to include this template and use the **Path** method we introduced earlier. We are also going to slightly tweak the style section to add a margin under our images, which will make our delete button and thumbnail combo look a little nicer.

```

{{define "galleryImages"}}
{{range .ImagesSplitN 6}}
  <div class="col-md-2">
    {{range .}}
      <a href="{{.Path}}>
        
      </a>
      {{template "deleteImageForm" .}}
    {{end}}
  </div>
{{end}}
<style>
  .thumbnail {
    width: 100%;
    margin-bottom: 6px;
  }
  .btn-delete {
    margin-bottom: 6px;
  }
</style>
{{end}}

```

At this point our edit page should now be rendering both images and delete buttons for each image. The UI isn't amazing looking, but making them work is more important so we are going to now start working on the a method for our ImageService that will enable us to delete images.

Open up the images model again.

```
$ atom models/images.go
```

We are going to add a Delete method to our ImageService. Our Delete method will accept an image as its only argument, and this is meant to define the image being deleted. The method will return an error if something goes wrong.

```
type ImageService interface {
    Create(galleryID uint, r io.Reader, filename string) error
    ByGalleryID(galleryID uint) ([]Image, error)
    Delete(i *Image) error
}
```

Now we are going to implement this method using the **Remove** function²⁰ from the **os** package. This function can be used to remove a specific file or directory. That means all we need to do is call the Remove function with the relative path of our image.

```
func (is *imageService) Delete(i *Image) error {
    return os.Remove(i.RelativePath())
}
```

After that we need to introduce a controller to handle any incoming image deletion requests. Open up the galleries controller.

```
$ atom controllers/galleries.go
```

We are going to name this method ImageDelete, and it will start off very similar to our ImageUpload method - we will lookup the gallery and verify the user has permission to edit it before moving on to any new code.

²⁰<https://golang.org/pkg/os/#Remove>

```
// POST /galleries/:id/images/:filename/delete
func (g *Galleries) ImageDelete(w http.ResponseWriter, r *http.Request) {
    gallery, err := g.galleryByID(w, r)
    if err != nil {
        return
    }
    user := context.User(r.Context())
    if gallery.UserID != user.ID {
        http.Error(w, "You do not have permission to edit "+
            "this gallery or image", http.StatusForbidden)
        return
    }
    // TODO: Finish implementing this
}
```

Once we know the current user has permission to edit the gallery we will get the filename from the path, build our Image, and then call the Delete method provided by our ImageService. If there is an error deleting the image we will render an error alert on the edit gallery page, otherwise we will redirect the user back to the edit gallery page that should no longer include that image.

```
func (g *Galleries) ImageDelete(w http.ResponseWriter, r *http.Request) {
    // ... This code remains unchanged

    // Get the filename from the path
    filename := mux.Vars(r)["filename"]
    // Build the Image model
    i := models.Image{
        Filename: filename,
        GalleryID: gallery.ID,
    }
    // Try to delete the image.
    err = g.is.Delete(&i)
    if err != nil {
        // Render the edit page with any errors.
        var vd views.Data
        vd.Yield = gallery
        vd.SetAlert(err)
        g.EditView.Render(w, r, vd)
        return
    }
    // If all goes well, redirect to the edit gallery page.
    url, err := g.r.Get(EditGallery).URL("id", fmt.Sprintf("%v", gallery.ID))
    if err != nil {
        http.Redirect(w, r, "/galleries", http.StatusFound)
```

```
    return
}
http.Redirect(w, r, url.Path, http.StatusFound)
}
```

Finally, we need to add a new route to our router.

```
$ atom main.go
```

```
r.HandleFunc("/galleries/{id:[0-9]+}/images/{filename}/delete",
requireUserMw.ApplyFn(galleriesC.ImageDelete)).
Methods("POST")
```

Restart your server and verify that you can now delete images from your gallery. This will also delete images from your local disk, so we have also learned how to delete files on your hard drive using Go.

Source code

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

Completed - book.usegolang.com/15.7

Changes - book.usegolang.com/15.7-diff

15.8 Known Bugs

There are few known bugs at this point in our application that we will be addressing in the next remainder of the course, but just in case you are experiencing them they are:

1. Special characters

Images with special characters in their name may not be rendering correctly. Eg the filename “/asfjalf/_@1!&?&.jpg” won’t work with our application right now.

2. Newly uploaded images aren’t rendered immediately.

Whenever we upload images our ImageUpload handler renders the edit gallery page without refreshing the gallery’s image list. As a result, the rendered edit gallery page won’t have the new images displayed.

The bigger issue here is that right now we have two options after a handler is successful:

1. Render the view with an alert
2. Redirect the user, but we can’t display an alert when we do this

We will eventually explore how to update our code so that we can both redirect a user - causing them to visit a new URL in our application - while also displaying an alert message on that page.

Chapter 16

Deploying to production

While our application isn't complete, we are now at a point where we can start getting ready to deploy it to a production server. This is important because going to production entails thinking about a few aspects of development that don't need as much consideration while in development.

For instance, when in local development we didn't need to worry about storing secrets in our code, but when we start working with other developers and deploying to production we will need a way to provide configuration data for our application. We will also need to start considering how our application will be restarted if it crashes, or if the server running it restarts.

None of these are things we needed to consider in development because it was just us working on the application, and if anything went wrong we would be around to restart the server or fix it. Whenever we go to production that all changes. Nobody is sitting at a computer waiting to restart the server, but instead it is running 24/7 while we sleep, eat, and do other work.

In this chapter we will explore how to start preparing your application for this different environment, and then we will look at how to setup and deploy a server on Digital Ocean.

Note: If you have never used Digital Ocean before, use this link - do.co/webdevgo - to get a free \$10 credit, which is enough to spin up the server we use for a free month or two.

16.1 Error handling

Before deploying to production, it is often a good idea to take a few moments and try to make sure your application's error handling is in a good spot. In development if we get an error we can often reproduce it many times and slowly add in new logging statements to see what is going on, but in production it might only be a specific user seeing an error message so it is a good idea to have some logging in place ahead of time. That way you can look at the logs and try to figure out what went wrong.

Take a few moments to look through our application and find instances where we have errors that could be handled better. This could mean anything from removing a panic and instead returning an error, adding in some logs to print out what the error was to help us with debugging, or even just redirecting a user to a more useful page whenever an error does occur. If you can think of any reason why you might want to add more logging or change the way an error is handled go ahead and try it out.

I'm not going to walk through making these changes because these won't make or break your application, and I believe you will gain more from this section by examining your own code more closely for a bit. The primary goal here is to understand what your code is doing and how it handles each error situations.

What I will do is provide you with my own updated source code, along with ample comments, so you have a reference as to what changes I chose to make. Your code doesn't have to match this perfectly, but it is worth looking at the changes and considering why they were made.

Note: Remember that the SetAlert method on our View type will also log any

non-public errors, so we don't need to manually log those.

Source code

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

Completed - book.usegolang.com/16.1

Changes - book.usegolang.com/16.1-diff

16.2 Serving static assets

Our application currently has a few CSS styles hard-coded into the HTML templates. This obviously isn't something we want to do long term; instead, we would like to be able to serve some static asset files, like a **styles.css** file. In this section we are going to introduce a new file server used to do just that.

The first thing we are going to do is create our styles file.

```
$ mkdir assets
$ atom assets/styles.css
```

```
/* LensLocked.com styles */
.thumbnail {
    width: 100%;
    margin-bottom: 6px;;
}
.btnDelete {
    margin-bottom: 6px;
}
.footer {
    padding-top: 60px;
}
```

After that we need to delete any inline styles. These can be found in the edit and show gallery templates.

```
$ atom views/galleries/edit.gohtml  
$ atom views/galleries/show.gohtml
```

In each of these files, find and remove the `<style>` sections.

Next up is adding our file server. This is going to be nearly identical to how we created our images file server; we are even going to need to use the `StripPrefix` handler.

```
$ atom main.go
```

Add the following handler with your other routes.

```
// Assets  
assetHandler := http.FileServer(http.Dir("./assets/"))  
assetHandler = http.StripPrefix("/assets/", assetHandler)  
r.PathPrefix("/assets/").Handler(assetHandler)
```

Box 16.1. When would we not use StripPrefix?

You might be asking yourself when you would use a FileServer without the `StripPrefix` handler as well. In the example above, we need both because our `http.Dir` starts in the assets directory already, but we also require the `/assets/` prefix in our path.

If we were to instead use the following setup, we wouldn't need to use the `StripPrefix` handler:

```
# Store our styles.css file here:  
./public/assets/styles.css
```

With our assets folder nested in a public folder, we could instead setup our handlers like below.

```
assetHandler := http.FileServer(http.Dir("./public/"))  
r.PathPrefix("/assets/").Handler(assetHandler)
```

Because our file server starts in the public directory, which has an assets directory in it, we don't need to remove the path prefix. Instead when we get a path like “/assets/styles.css” it will look in the assets directory inside of the public dir, and then find the styles.css file inside of that directory.

With our new routes defined we are finally ready to update our bootstrap template to tell it about our css styles file.

Note: We won't be covering asset pipelines in this course, but if you wanted to use one you would likely want to have it output finished CSS and JS files to the assets directory we just created.

```
$ atom views/layouts/bootstrap.gohtml
```

Find the `<head>` section of your template and add the following link tag.

```
<head>  
  <title>LensLocked.com</title>  
  <link href="//maxcdn.bootstrapcdn.com/bootstrap/3.3.7/css/bootstrap.min.css"  
        rel="stylesheet">  
  <!-- Add this line to include our styles.css stylesheet -->
```

```
<link href="/assets/styles.css" rel="stylesheet">  
</head>
```

Congrats, your server now has the ability to serve static assets! You can verify this by trying to access the styles.css file directly.

localhost:3000/assets/styles.css

Source code

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

Completed - book.usegolang.com/16.2

Changes - book.usegolang.com/16.2-diff

16.3 CSRF protection

We talked about CSRF earlier when we discuss authentication systems and cookies. CSRF protection will help us prevent other websites from creating a form that submits to our server and tricking users into performing actions they didn't intend to perform.

The basic idea is that our web server will generate a specific token for each user and then store it in a way that is recoverable. For instance, it is common to store a CSRF token in a user's cookies since these are only readable by our web server. Then later when we create forms, we will also embed the CSRF token into the HTML form as a hidden field. When a user submits a form, both the form they submitted and their cookies will have this CSRF token in them.

If either of these tokens are not present, any form submissions will be rejected.

This will prevent other website from creating fake forms because they wouldn't know what CSRF token a user has in their cookies to fake it.

We are going to implement this using the gorilla/csrf package. Technically we could try to write this ourselves but in my experience it is better to use a battle-tested security library like this than to write it yourself. It is easy to introduce a bug otherwise, and in my experience most web developers don't ever write a CSRF library themselves but instead learn to use trusted libraries to do this for them.

First we need to install the package.

```
$ go get github.com/gorilla/csrf
```

Once you install it you may want to reference the docs a bit. They can be found at: <https://github.com/gorilla/csrf>

In the docs we can find the following example:

```
CSRF := csrf.Protect([]byte("32-byte-long-auth-key"))
http.ListenAndServe(":8000", CSRF(r))
```

The **csrf** package provides us with a Protect function that is used to create a CSRF middleware. We then apply that middleware to our router (or whatever top level handler we use) so that the CSRF protection happens before any other handlers get called. By doing this, we ensure that all incoming requests have their CSRF tokens validated before we run any custom code.

When creating the middleware we need a random 32 byte authentication key. This is used to make sure our application isn't generating predictable CSRF tokens.

We are going to use this example as a starting point and add the CSRF middleware to our own application.

```
$ atom main.go
```

We will tweak the name a bit, and we will also be using the `rand` package we created to generate our 32 bytes.

```
csrfMw := csrf.Protect(rand.Bytes(32))
```

Note: This may invalidate any existing CSRF tokens in forms when we restart our server, but I haven't seen this issue pop up frequently. We don't restart our server too often, and if you are really concerned you could add a static token into our config when we cover config files later in this chapter.

Once we have our `csrfMw` function we can apply it to our router to make sure it runs before any other handlers.

```
http.ListenAndServe(":3000", csrfMw(userMw.Apply(r)))
```

Now any time a form is submitted or our server gets an HTTP POST web request the `csrfMw` will check for a valid CSRF token. If it is present, the next handler will be called, otherwise the entire request will be rejected before it even gets to our handlers. This enables us to write our handlers assuming this security issue has already been implemented for us.

We need to make one final tweak to our code, which is telling the `csrf` package whether we are running in production or not. We are going to do this with a temporary variable that will eventually be replaced with a configuration variable when introduce those.

Find the line where you instantiated your `csrfMw` variable and update it with the following code.

```
// TODO: Update this to be a config variable
isProd := false
b, err := rand.Bytes(32)
if err != nil {
    panic(err)
}
csrfMw := csrf.Protect(b, csrf.Secure(isProd))
```

The **csrf** package also provides us with a helper to generate tokens for our HTML forms. The package's docs then explain that their recommended way of providing this to your HTML forms is to add it as a data field that you pass in to your templates. For instance, we might add this field to our **views.Data** type.

We are going to take a slightly different approach where we instead add it as a template function. The primary benefit to this approach over the recommended approach is that we can access it in any template, even if no data was passed into the template. This makes it easier to generate forms without having to worry about whether or not the CSRF token was provided as data to the form.

To do this we are going to look at the **Funcs** function¹ provided by the template package. With it we can introduce new functions that are available inside of our templates.

We are going to name the function **csrfField** and we will start by adding it into all of our templates that need it. We are going to assume that calling the function **csrfField** will return everything we need to generate a CSRF token hidden input field; that is, we are assuming it returns the entire HTML blob we need, not just a token.

```
$ atom views/galleries/edit.gohtml
```

In the edit gallery template we need to add this function call for every form we create. All four of these templates are shown below, but it is important to note

¹<https://golang.org/pkg/html/template/#Template.Funcs>

that we are only adding the csrfField function call directly after the form tag. Everything else remains unchanged.

```

{{define "editGalleryForm"}}
<form action="/galleries/{{.ID}}/update" method="POST" class="form-horizontal">
  <!--
    This line will return:
      <input type="hidden" ...>
    with the CSRF token as its value.
  -->
  {{csrfField}}
  <!-- ... remains unchanged -->
</form>
{{end}}
```

```

{{define "deleteGalleryForm"}}
<form action="/galleries/{{.ID}}/delete" method="POST" class="form-horizontal">
  {{csrfField}}
  <!-- ... remains unchanged -->
</form>
{{end}}
```

```

{{define "uploadImageForm"}}
<form action="/galleries/{{.ID}}/images" method="POST" enctype="multipart/form-data" class="form-horizontal">
  {{csrfField}}
  <!-- ... remains unchanged -->
</form>
{{end}}
```

```

{{define "deleteImageForm"}}
<form action="/galleries/{{.GalleryID}}/images/{{.Filename}}/delete" method="POST">
  {{csrfField}}
  <!-- ... remains unchanged -->
</form>
{{end}}
```

Next up is the new gallery form.

```
$ atom views/galleries/new.gohtml
```

```

{{define "galleryForm"}}
<form action="/galleries" method="POST">
  {{csrfField}}
  <!-- ... remains unchanged -->
```

```
</form>
{{end}}
```

And then the login form.

```
$ atom views/users/login.gohtml
```

```
{{define "loginForm"}}
<form action="/login" method="POST">
  {{csrfField}}
  <!-- ... remains unchanged -->
</form>
{{end}}
```

Finally, the sign up form.

```
$ atom views/users/new.gohtml
```

```
{{define "signupForm"}}
<form action="/signup" method="POST">
  {{csrfField}}
  <!-- ... remains unchanged -->
</form>
{{end}}
```

If we try to restart our server at this point we will see an error message roughly equivalent to:

```
function "csrfField" not defined
```

Whenever we first parse our templates they will check to see if all the functions we call are defined. At this point our csrfField function isn't defined, but we

can't truly define it until a web request is made. This is due to the fact that we need to read the user's CSRF token so we know which token to include in the form.

To handle this, we are going to create a stubbed out version of the `csrfField` function *before* we parse our templates, and then later when we get a web request we will update the template function before executing it for real.

Note: This technique is a little more advanced, but is incredibly useful once you understand it fully as it enables us to create template-wide functions with user-specific data. For instance we could replace the `.User` field of our views.Data type with function like `isLoggedIn` and `isAdmin` and use those in any template, even if the `.User` field wasn't provided.

```
$ atom views/view.go
```

We will start with new imports.

```
import (
    // Add the following imports
    "errors"
    "github.com/gorilla/csrf"
)
```

After that we will update the `NewView` function so that it provides a stubbed `csrfField` function. If this actually gets called without being replaced it will result in an error, but we will be replacing it when we call the `Render` method of our views.

```
func NewView(layout string, files ...string) *View {
    addTemplatePath(files)
    addTemplateExt(files)
    files = append(files, layoutFiles()...)
    // We are changing how we create our templates, calling
    // New("") to give us a template that we can add a function to
```

```
// before finally passing in files to parse as part of the template.
t, err := template.New("").Funcs(template.FuncMap{
    "csrfField": func() (template.HTML, error) {
        // If this is called without being replace with a proper implementation
        // returning an error as the second argument will cause our template
        // package to return an error when executed.
        return "", errors.New("csrfField is not implemented")
    },
    // Once we have our template with a function we are going to pass in files
    // to parse, much like we were previously.
}).ParseFiles(files...)
if err != nil {
    panic(err)
}

return &View{
    Template: t,
    Layout:   layout,
}
}
```

Next we need to replace this csrfField function with a real implementation whenever we finally render our view.

```
func (v *View) Render(w http.ResponseWriter, r *http.Request, data interface{}) {
    // ... The code used to set the header and Data remains unchanged

    vd.User = context.User(r.Context())
    var buf bytes.Buffer

    // We need to create the csrfField using the current http request.
    csrfField := csrf.TemplateField(r)
    tpl := v.Template.Funcs(template.FuncMap{
        // We can also change the return type of our function, since we no longer
        // need to worry about errors.
        "csrfField": func() template.HTML {
            // We can then create this closure that returns the csrfField for
            // any templates that need access to it.
            return csrfField
        },
    })
    // Then we continue to execute the template just like before.
    err := tpl.ExecuteTemplate(&buf, v.Layout, vd)
    if err != nil {
        http.Error(w, "Something went wrong. If the problem "+
            "persists, please email support@lenslocked.com",
            http.StatusInternalServerError)
    }
}
```

```

    return
}
io.Copy(w, &buf)
}

```

We have one final issue to address before continuing. The gorilla/schema package we are using to parse our forms is setup by default to error if there are any keys submitted in a form that we don't attempt to decode. This isn't an awful default when not using CSRF tokens, but now that we have introduced them it is problematic because we have a new key added to our forms that our we won't ever want to manually decode, and this will cause the **schema.Decoder** to error when decoding the form.

To change this setting we are going to call the `IgnoreUnknownKeys` function² which tells our decoder to simply ignore keys we don't ask it to decode.

```
$ atom controllers/helpers.go
```

```

func parseForm(r *http.Request, dst interface{}) error {
    if err := r.ParseForm(); err != nil {
        return err
    }

    dec := schema.NewDecoder()
    // Call the IgnoreUnknownKeys function to tell schema's decoder
    // to ignore the CSRF token key
    dec.IgnoreUnknownKeys(true)
    if err := dec.Decode(dst, r.PostForm); err != nil {
        return err
    }
    return nil
}

```

That's it for implementing CSRF protection. Restart your server and verify that your forms are all still working. Be sure to refresh any pages you already had

²<http://www.gorill toolkit.org/pkg/schema#Decoder.IgnoreUnknownKeys>

open, otherwise the HTML won't be updated and you won't have a CSRF field in your HTML. You can also inspect the HTML source generated by your server to verify that CSRF input fields are being added to each form.

Source code

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

Completed - book.usegolang.com/16.3

Changes - book.usegolang.com/16.3-diff

16.4 Limiting middleware for static assets

Our Users middleware is currently checking to see if the user is logged in every time a user requests a public image or asset. While this might be useful if we had private images, we don't and this ends up causing many extra database lookups. One simple way to prevent this is to update the middleware to not run on specific routes.

```
$ atom middleware/require_user.go
```

We will grab the current path of the web request and just check to see if it starts with `/assets/` or `/images/`. If it does, we will call the next middleware and return without ever looking up the user.

```
import (
    // Add this import
    "strings"
)
```

```
func (mw *User) ApplyFn(next http.HandlerFunc) http.HandlerFunc {
    return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
        path := r.URL.Path
        // If the user is requesting a static asset or image
        // we will not need to lookup the current user so we skip
        // doing that.
        if strings.HasPrefix(path, "/assets/") ||
            strings.HasPrefix(path, "/images/") {
            next(w, r)
            return
        }

        // ... everything else remains unchanged
    })
}
```

Note: Another way of handling this would be to only apply the middleware to routes that require it, much like we do with our RequireUser middleware. It ends up being more code, but reduces how tightly linked our middleware and routes are. In your own applications you will have to decide which route makes the most sense.

Restart your server and visit a page with images, like a gallery you have created. Before making these changes you likely saw many log messages like:

```
SELECT * FROM "users" WHERE ...
```

Now that we have updated our code you should see these less frequently because our application is querying the database less often.

Source code

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

Completed - book.usegolang.com/16.4

Changes - book.usegolang.com/16.4-diff

16.5 Fixing bugs

The only web applications that will ship to production without any bugs are likely going to be ones with no features whatsoever. The instant you add new features there is always a risk that something will go wrong, or that you won't think about an obscure edge case.

While this might sound terrible, the truth is this is just part of the development process and it is why Go stresses being highly readable over nearly everything else. Developers spend a majority of their time working on an existing code-base, reading code others wrote (or that they wrote a long time ago), adding new features, fixing bugs, and generally maintaining an application.

In our own application we have managed to introduce a few bugs along the way that we are going to take some time to address in the next two sub-sections.

Yes, I could rewrite the book to address these bugs before they arise but I have intentionally allowed for bugs because I find it is a very real part of the development process and learning how to recognize, replicate, and fix them is a valuable skill.

16.5.1 URL encoding image paths

The first bug we need to look at is one you are unlikely to catch yourself. If a user uploads an image with certain special characters, our application will currently fail to handle it correctly.

We can see this in action by creating a file with a name that includes forward slashes or question marks in it.

```
# Some operating systems may not like this filename either
/asfjalf/_@1!&?&.jpg
```

What makes this filename interesting is that it has special characters in it that even in a terminal could be problematic. As a result, it is unlikely that end users will have filenames with slashes in them, but they might have question marks or other problematic characters in their filenames. These characters have special meanings in URL paths, so we need to encode them before creating the **src** tag of our **** HTML.

We are going to first update the **Path** function we defined on our **Image** type.

```
$ atom models/images.go
```

To create a properly encoded URL path we are going to be using the **url.URL** type. We will create a new URL, setting the Path field with our own path, and then we will call the **String** method provided by the URL type which will output an encoded URL. Since we never defined a domain (eg “abc.com”) or a protocol (eg “https”) the output will only have the path of the URL.

```
import (
    // Add this import
    "net/url"
)

func (i *Image) Path() string {
    temp := url.URL{
        Path: "/" + i.RelativePath(),
    }
    return temp.String()
}
```

Now anywhere in our templates or other code that we call the **Path** method we don’t need to worry about URL encoding the path - it is already done for us.

Lastly we need to update the edit gallery template. In it we manually build a path for deleting a gallery image, and we are going to use this instance to learn about how we can add yet another function to our templates to help us escape our paths correctly.

First, open the views source file.

```
$ atom views/view.go
```

What we want to add to our templates is a function that allows us to properly escape a path, much like the `url.URL` type did for us when building a path. Luckily the `net/url` package provides us with a `PathEscape` function³ which does just that - escapes a string appropriately for a path.

In order to add this to our template we are going to update the `NewView` function, adding the function like we did the `csrfField` function, but this time we will go ahead and implement it.

```
import (
    // Add this
    "net/url"
)

func NewView(layout string, files ...string) *View {
    addTemplatePath(files)
    addTemplateExt(files)
    files = append(files, layoutFiles()...)
    t, err := template.New("").Funcs(template.FuncMap{
        "csrfField": func() (template.HTML, error) {
            return "", errors.New("csrfField is not implemented")
        },
        "pathEscape": func(s string) string {
            return url.PathEscape(s)
        },
    }).ParseFiles(files...)
    // ... the rest remains unchanged
}
```

³<https://golang.org/pkg/net/url/#PathEscape>

Now we can update our edit template to use this new function.

```
$ atom views/galleries/edit.gohtml
```

In our deleteImageForm template we are going to update our code to match the code below.

```
 {{define "deleteImageForm"}}
<form
  action="/galleries/{{.GalleryID}}/images/{{pathEscape .Filename}}/delete"
  method="POST">
```

All we did here was add the **pathEscape** portion, passing in the Filename as its only argument. This will call the pathEscape function we added which will escape the filename and return the results which will then be inserted into our path.

After making these changes our code will properly support filenames with special characters in them and address this bug!

Source code

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

Completed - book.usegolang.com/16.5.1

Changes - book.usegolang.com/16.5.1-diff

16.5.2 Redirecting after image uploads

The last bug we are going to address before getting back to our production setup is one found after we upload new images.

Whenever we upload new images, our code will currently render the edit gallery view again, but our code never refreshes the list of images for that gallery. As a result, we will see a “success” alert message after submitting the form, but we won’t see any images.

Realistically, what we should be doing after a successful image upload is redirecting the user back to the edit gallery page. The downside to this is that we don’t have a way to both redirect a user and show them an alert message. We won’t be introducing that until the next chapter, but for now we are going to update our code to redirect a user rather than displaying an alert message, as this is a much more usable version of the code.

Note: If you are following the screencasts we may already be doing this in that source code.

Open up your galleries controller source file.

```
$ atom controllers/galleries.go
```

Find the ImageUpload method and update the code at the end to instead redirect a user to the edit page after successfully uploading images.

```
func (g *Galleries) ImageUpload(w http.ResponseWriter,
    r *http.Request) {
    // ... this is all unchanged

    // Remove the code used to create a success alert and
    // render the EditView and replace it with the following.
    url, err := g.r.Get(EditGallery).URL("id",
        fmt.Sprintf("%v", gallery.ID))
    if err != nil {
        http.Redirect(w, r, "/galleries", http.StatusFound)
        return
    }
    http.Redirect(w, r, url.Path, http.StatusFound)
}
```

Another side effect of this change is that whenever a user submits the image

upload form their URL will get changed back to the edit gallery URL, making it easier to copy/paste a working link.

Source code

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

Completed - book.usegolang.com/16.5.2

Changes - book.usegolang.com/16.5.2-diff

16.6 Configuring our application

Up until now we have been hard-coding quite a few configuration variables into our application. While this is faster when getting started with an application, it isn't a great idea long term. We don't want to have to rebuild our entire application just to change a single configuration value, and it is a bad idea to store secret keys in our source code where any developer has access to it. We also don't want to make development harder than necessary, so it is a good idea to provide a viable set of default configuration values to run in development.

In the next subsection we will focus on finding all of the variables in our application that need to be pulled out of our application and provided via configuration variables and providing default values for each of these to use in development.

In the second subsection we will explore functional options, and see how we can use them to make providing configuration variables to nested portioned of our application much easier.

In the final subsection we will see how to use a JSON file to provide alternative configurations and how to load that with our Go code.

16.6.1 Finding variables that need provided

When introducing configuration variables to an application, I find it is often easier to first go through your application looking for any variables that need to be provided and jotting them down. Doing this all upfront will help us figure out what variables we need to account for, but will also help us get a mental map of what we are about to code.

Below are all of the variables and constants we have in our application that are likely going to need to be defined via an application config.

```
// models/users.go
const (
    userPwPepper = "secret-random-string"
    hmacSecretKey = "secret-hmac-key"
)
```

```
// models/services.go
// The "postgres" dialect should likely be a config. We also
// likely want to build the connectionInfo with a config.
db, err := gorm.Open("postgres", connectionInfo)
// The LogMode could be determined by our environment.
// Eg prod = false, dev = true
db.LogMode(true)
```

```
// main.go
const (
    host      = "localhost"
    port      = 5432
    user      = "postgres"
    password  = "your-password"
    dbname   = "lenslocked_dev"
)
// Whether or not we are running in production should be
// determined via a config value of some sort.
isProd := false
// We won't be doing it, but you could define the CSRF
// bytes via a config as well.
b, err := rand.Bytes(32)
```

Note: Don't code anything above. These snippets are pulled from our code to demonstrate places we could use config variables.

While that might appear to be a lot at first, it actually won't take very long to pull this all out of our application and instead replace it with configuration variables.

Let's start with the PostgreSQL info defined in our main.go source file. We are going to define a type that makes it easier to provide these values, and then we will write a function to provide our default values.

First we will create the config source file.

```
$ atom config.go
```

```
package main

import "fmt"

type PostgresConfig struct {
    Host      string `json:"host"`
    Port      int    `json:"port"`
    User      string `json:"user"`
    Password  string `json:"password"`
    Name      string `json:"name"`
}

func (c PostgresConfig) Dialect() string {
    return "postgres"
}

func (c PostgresConfig) ConnectionInfo() string {
    // We are going to provide two potential connection info
    // strings based on whether a password is present
    if c.Password == "" {
        return fmt.Sprintf("host=%s port=%d user=%s dbname=%s " +
            "sslmode=disable", c.Host, c.Port, c.User, c.Name)
    }
    return fmt.Sprintf("host=%s port=%d user=%s password=%s " +
        "dbname=%s sslmode=disable", c.Host, c.Port, c.User,
        c.Password, c.Name)
}
```

Note: We will discuss the JSON tags on the PostgresConfig type in an upcoming section, but have added them here so they are available later.

The only two pieces of this config that our application needs are the Dialect and the ConnectionInfo, so we are providing both of these via a function. Then down the road if we want to switch to say MySQL we could create a MySQLConfig that returns the correct dialect and connection info for that database.

The rest of the fields - Host, Port, and others - are all used to build the connection info, but because we don't really need them for the Dialect we can just hard-code that portion. We won't be using the PostgresConfig for another database dialect's config.

Now we need a way to create our default configuration. Individual developers could override it with something custom for their machine if they wanted, but if they have a setup similar to ours it would be a good starting point.

```
func DefaultPostgresConfig() PostgresConfig {
    return PostgresConfig{
        Host:      "localhost",
        Port:      5432,
        User:      "jon",
        Password: "your-password",
        Name:      "lenslocked_dev",
    }
}
```

Next up is a configuration for the rest of our application - things like the port, environment (dev vs prod), and eventually our password pepper and an HMAC key.

```
type Config struct {
    Port int
    Env  string
}

func (c Config) IsProd() bool {
    return c.Env == "prod"
}
```

```
func DefaultConfig() Config {
    return Config{
        Port: 3000,
        Env:  "dev",
    }
}
```

Note: We will add values for the HMAC key and password pepper in the next section.

With most of our configs setup, we can update our existing to use them. We'll start with the NewServices function, updating it to accept a dialect string. After that we will update the main function to instantiate default config instances and using them to set the values we previously hard-coded.

```
$ atom models/services.go
```

```
import (
    "github.com/jinzhu/gorm"
    _ "github.com/jinzhu/gorm/dialects/postgres"
)

func NewServices(dialect, connectionInfo string) (*Services, error) {
    db, err := gorm.Open(dialect, connectionInfo)
    // ... nothing else changes
}
```

And then the main function...

```
$ atom main.go
```

```
func main() {
    cfg := DefaultConfig()
    dbCfg := DefaultPostgresConfig()
    services, err := models.NewServices(dbCfg.Dialect(),
```

```
dbCfg.ConnectionInfo())
// ... Unchanged code here

// ... our routes are unchanged

// Delete the isProd var
// isProd := false
b, err := rand.Bytes(32)
if err != nil {
    panic(err)
}
// Use the config's IsProd method instead
csrfMw := csrf.Protect(b, csrf.Secure(cfg.IsProd()))

// Our port is not provided via config, so we need to
// update the last bit of our main function.
fmt.Printf("Starting the server on :%d...\n", cfg.Port)
http.ListenAndServe(fmt.Sprintf(":%d", cfg.Port),
    csrfMw(userMw.Apply(r)))
}
```

Moving forward we will need to run and build our application by providing *all* of the source files in our main package as arguments, but we will also be able to provide all of these variables via configurations!

```
$ go run *.go
```

This tells our code that we need to build multiple Go files, not just main.go, when building and running our program.

Source code

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

Completed - book.usegolang.com/16.6.1

Changes - book.usegolang.com/16.6.1-diff

16.6.2 Using functional options

Next we need a way to provide config variables to our services when building them. Whenever we setup our UserService, we want to provide it with things like the password pepper and an HMAC secret key, but our current code design doesn't allow for that.

One way to do this is to add these all in as arguments to the NewServices function we wrote, but by doing this we will quickly end up with a long list of arguments required for the functions.

```
// And this list will only grow in size!
func NewServices(dialect, connectionInfo, pepper, hmacKey string,
    logMode bool) (*Services, error) {
    // ...
}
```

Note: Do not code this - it is for illustration purposes.

We obviously don't want to be using a function that grows to require 10+ arguments. It would be pretty hard to use, and especially confusing to remember which arguments are required vs which ones are optional.

Another alternative to this would be to create a config data type for our NewServices function call and use that.

```
type ServicesConfig struct {
    Dialect string
    ConnectionInfo string
    Pepper string
    // ...
}

func NewServices(cfg ServicesConfig) (*Services, error) { ... }
```

Note: Do not code this - it is for illustration purposes.

This looks a little cleaner at first, but it is still prone to many of the same issues. We need to be aware of which variables need set, or which variables need paired together. For instance, if we don't provide a dialect but do provide a connection info what happens? It isn't completely clear looking our ServicesConfig type that these two are paired together.

Rather than using either of these approaches, we are going to use what are known as functional options.

Box 16.2. More on functional options

Functional options are basically a fancy name for using functions as a parameter. That parameter is often variadic, meaning you can provide as many of those arguments (functions) as you want, and the function receiving them will often iterate over them in the order received to perform an operation or build something.

If you would like to read more on the topic, I suggest the following two articles. You can also find a video of Dave's post if you prefer talks.

- <https://book.usegolang.com/func-opts-dave>
- <https://book.usegolang.com/func-opts-jon>

Before getting to the functional options, lets go ahead and add the last two config variables - the pepper and HMAC secret key - to our config type so we have access to them.

```
$ atom config.go
```

```

type Config struct {
    Port    int      `json:"port"`
    Env     string   `json:"env"`
    Pepper   string   `json:"pepper"`
    HMACKey string   `json:"hmac_key"`
}

func DefaultConfig() Config {
    return Config{
        Port:    3000,
        Env:     "dev",
        Pepper:  "secret-random-string",
        HMACKey: "secret-hmac-key",
    }
}

```

Note: We will discuss the JSON tags on the Config type in an upcoming section, but have added them here so they are available later.

Instead of passing in a config filled with data options, functional options allow us to provide a series of functions that will help us build our services. For example, if we wanted a function to add a UserService to our Services object, we might write the following code:

```

// WithUser accepts pepper and hmacKey as arguments, then
// returns a function as its return value. The function it
// returns is one that accepts a Services pointer as its
// only argument and returns an error.
func WithUser(pepper, hmacKey string) func(*Services) error {
    return func(s *Services) error {
        // Our NewUserService doesn't match this definition yet
        s.User = NewUserService(s.db, pepper, hmacKey)
        return nil
    }
}

```

Note: We will eventually write this function and use it in our code, but it will be presented again at that time.

This is very similar to what we did when we built out our validation code; most notably when we created the idGreaterThanOrEqual validation. We are creating a func-

tion that returns a new function that uses some of the data passed into the first function. This is often referred to as a closure.

But what does this get us? Think back on when we wrote our runUserValFns function; by creating a function that accepted many functions, we were able to easily iterate over all of the functions and run them sequentially without caring about what each individual function did. All we knew was that those functions were validating our data and returning an error if there was an issue.

When using functional options to build our Services object we can do something very similar. We can write a NewServices function that accepts a list of functions intended to build the Services object, but we don't have to worry about how they are building it. We just need to run each function in the order provided and return either an error or the resulting Services object.

Let's start writing the new source code, which will help make this all clearer.

```
$ atom models/services.go
```

```
// ServicesConfig is really just a function, but I find using
// types like this are easier to read in my source code.
type ServicesConfig func(*Services) error

// NewServices now will accept a list of config functions to
// run. Each function will accept a pointer to the current
// Services object as its only argument and will edit that
// object inline and return an error if there is one. Once
// we have run all configs we will return the Services object.
func NewServices(cfgs ...ServicesConfig) (*Services, error) {
    var s Services
    // For each ServicesConfig function...
    for _, cfg := range cfgs {
        // Run the function passing in a pointer to our Services
        // object and catching any errors
        if err := cfg(&s); err != nil {
            return nil, err
        }
    }
    // Then finally return the result
    return &s, nil
}
```

Now we don't need our NewServices function to know about a mega-config file or many different variables being passed in. It doesn't have to have logic like:

```
if we have a dialect and a connection info
    then build a gorm
if we have a pepper and an hmac key
    then build a user service
```

This logic can all be handled explicitly through functions design to perform each of these tasks. For instance, when we updated our NewServices function we delete the source code used to open a GORM connection to our database along with the code used to set the log mode. We can add both of those back in with functional options.

```
$ atom models/services.go
```

```
func WithGorm(dialect, connectionInfo string) ServicesConfig {
    return func(s *Services) error {
        db, err := gorm.Open(dialect, connectionInfo)
        if err != nil {
            return err
        }
        s.db = db
        return nil
    }
}

func WithLogMode(mode bool) ServicesConfig {
    return func(s *Services) error {
        s.db.LogMode(mode)
        return nil
    }
}
```

Now we can use these two functional options in our main function like so:

```
$ atom main.go
```

```
func main() {
    func main() {
        cfg := DefaultConfig()
        dbCfg := DefaultPostgresConfig()
        // This isn't complete, but we will come back to it shortly
        services, err := models.NewServices(
            models.WithGorm(dbCfg.Dialect(), dbCfg.ConnectionInfo()),
            // Only log when not in prod
            models.WithLogMode(!cfg.IsProd()),
        )
        // ... this remains unchanged
    }
}
```

We are going to proceed through the rest of our services, writing a function to setup each on our Services type, but first we need to update the user service code to allow us to pass in a pepper and HMAC key.

```
$ atom models/users.go
```

First, delete the constants we were previously using.

```
// Delete these
const (
    userPwPepper = "secret-random-string"
    hmacSecretKey = "secret-hmac-key"
)
```

This will cause some errors in our code when we compile it, making it easier to find code we need to update. Next up is the NewUserService function along with the userService type.

```

func NewUserService(db *gorm.DB, pepper, hmacKey string) UserService {
    ug := &userGorm{db}
    hmac := hash.NewHMAC(hmacKey)
    // This won't compile, but we will add the pepper param
    // to the newUserValidator function shortly
    uv := newUserValidator(ug, hmac, pepper)
    return &userService{
        UserDB: uv,
        pepper: pepper,
    }
}

type userService struct {
    UserDB
    pepper string
}

```

We need to update our userValidator code to account for these changes as well.

```

func newUserValidator(ldb UserDB, hmac hash.HMAC,
    pepper string) *userValidator {
    return &userValidator{
        UserDB: ldb,
        hmac:   hmac,
        pepper: pepper,
        emailRegex: regexp.MustCompile(
            `^[\w\.-%]+\@[a-zA-Z\.\-]+\.[a-zA-Z]{2,16}$`),
    }
}

type userValidator struct {
    UserDB
    hmac      hash.HMAC
    emailRegex *regexp.Regexp
    pepper    string
}

```

After that we need to find our authentication code that previously used the pepper constant and replace it with our userService's pepper field.

```

func (us *userService) Authenticate(email,
    password string) (*User, error) {
    // ...
}

```

```
err = bcrypt.CompareHashAndPassword(
    []byte(foundUser.PasswordHash),
    // Change this line to be us.pepper
    []byte(password+us.pepper))
// ...
}
```

And we also need to update the bcryptPassword method provided by our userValidator.

```
func (uv *userValidator) bcryptPassword(user *User) error {
    // ...
    // Change this to be uv.pepper
    pwBytes := []byte(user.Password + uv.pepper)
    // ...
}
```

We can now go back to our services source code and create config functions for each of our services.

```
$ atom models/services.go
```

```
func WithUser(pepper, hmacKey string) ServicesConfig {
    return func(s *Services) error {
        s.User = NewUserService(s.db, pepper, hmacKey)
        return nil
    }
}

func WithGallery() ServicesConfig {
    return func(s *Services) error {
        s.Gallery = NewGalleryService(s.db)
        return nil
    }
}

func WithImage() ServicesConfig {
    return func(s *Services) error {
        s.Image = NewImageService()
        return nil
    }
}
```

Finally, we need to finish updating our main function. We will want all of these services, so we want to add each of these functional options to our NewServices function call.

```
$ atom main.go
```

```
func main() {
    cfg := DefaultConfig()
    dbCfg := DefaultPostgresConfig()
    services, err := models.NewServices(
        models.WithGorm(dbCfg.Dialect(), dbCfg.ConnectionInfo()),
        models.WithLogMode(!cfg.IsProd()),
        // We want each of these services, but if we didn't need
        // one of them we could possibly skip that config func
        models.WithUser(cfg.Pepper, cfg.HMACKey),
        models.WithGallery(),
        models.WithImage(),
    )
    // ...
}
```

Restart the web application and verify that everything compiles and works as it used to. Be sure to test that you can still create galleries, login, and view images.

```
$ go run *.go
```

Source code

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

Completed - book.usegolang.com/16.6.2

Changes - book.usegolang.com/16.6.2-diff

16.6.3 JSON configuration files

Not every developer will have the same configuration, and we are likely to have a very different config in production, so we need a way for our application to load its config from some external source. In our case, we are going to use a JSON file that provides all the data we need.

We will start by creating our JSON config file, which is going to be very similar to the Config and PostgresConfig structs we defined earlier.

```
$ atom .config
```

This will be a JSON file storing all of the data our application needs.

```
{
  "port": 3000,
  "env": "dev",
  "pepper": "secret-random-string",
  "hmac_key": "secret-hmac-key",
  "database": {
    "host": "localhost",
    "port": 5432,
    "user": "jon",
    "password": "your-password",
    "name": "lenslocked_dev"
  }
}
```

We don't want to have many different JSON config files, so what we did here was embed the database information inside of the default config structure. This helps us keep our information organized, while also allowing us to define all our config data in a single JSON file.

We need to update our config.go source file to take this change into account.

```
$ atom config.go
```

```
type Config struct {
    Port      int          `json:"port"`
    Env       string       `json:"env"`
    Pepper    string       `json:"pepper"`
    HMACKey  string       `json:"hmac_key"`
    // We are adding the Database nested structure with this
    // new field
    Database PostgresConfig `json:"database"`
}

func DefaultConfig() Config {
    return Config{
        Port:      3000,
        Env:       "dev",
        Pepper:    "secret-random-string",
        HMACKey:  "secret-hmac-key",
        Database: DefaultPostgresConfig(),
    }
}
```

Next we are going to write the code used to load our JSON config file into the Config type and return it. If a JSON config file isn't present, we will have this function fall back on the default one. To do this, we are going to need to use the encoding/json package, which allows us to convert Go data structures into JSON, and vice versa.

In our case we want to go from JSON to a Go type, so we need to decode JSON data into a Go struct. To do this, we will use the **NewDecoder** function⁴ which accepts an io.Reader as its only argument and then returns a pointer to a **Decoder**⁵. The reader argument to the NewDecoder function call can be any type of reader, but in our case it is going to be the config JSON file we open up.

Note: You may sometimes hear this referred to as unmarshalling JSON data into a Go structure as well.

⁴<https://golang.org/pkg/encoding/json/#NewDecoder>

⁵<https://golang.org/pkg/encoding/json/#Decoder>

Once we have a Decoder we can use it to decode the file and place the results into a Config struct by calling the Decoer's **Decode** method⁶.

It sounds like a lot, but it is really only a few lines of code. Most of the code below is used to explain what we are doing.

```
import (
    // Add these imports as you need them
    "encoding/json"
    "fmt"
    "os"
)

func LoadConfig() Config {
    // Open the config file
    f, err := os.Open(".config")
    if err != nil {
        // If there was an error opening the file, print out a
        // message saying we are using the default config and
        // return it.
        fmt.Println("Using the default config...")
        return DefaultConfig()
    }
    // If we opened the config file successfully we are going
    // to create a Config variable to load it into.
    var c Config
    // We also need a JSON decoder, which will read from the
    // file we opened when decoding
    dec := json.NewDecoder(f)
    // We then decode the file and place the results in c, the
    // Config variable we created for the results. The decoder
    // knows how to decode the data because of the struct tags
    // (eg `json:"port") we added to our Config and
    // PostgresConfig fields, much like GORM uses struct tags
    // to know which database column each field maps to.
    err = dec.Decode(&c)
    if err != nil {
        panic(err)
    }
    // If all goes well, return the loaded config.
    fmt.Println("Successfully loaded .config")
    return c
}
```

⁶<https://golang.org/pkg/encoding/json/#Decoder.Decode>

Box 16.3. JSON encoding and decoding

If you are interested in learning more about using JSON in Go to encode or decode data I suggest the following resources:

- <https://golang.org/pkg/encoding/json/> - Standard docs for the json package
- <https://gobyexample.com/json> - Quick examples of how to encode and decode JSON
- <https://eager.io/blog/go-and-json/> - A tutorial on using JSON with Go
- <https://pocketgophers.com/json-references/> - A list of many valuable Go + JSON articles that will likely address any questions you have when learning to encode and decode JSON data in Go

Now that we have a way to load a JSON config, we are going to update our main function to use it.

```
$ atom main.go
```

```
func main() {
    cfg := LoadConfig()
    dbCfg := cfg.Database
    // ... the rest of this is unchanged
}
```

Try updating your config file to verify it works; change something obvious like the port and restart your server. Check that it starts up on a new port and prints out a message with that port.

*Note: Remember you need to **go run *.go** instead of main.go*

We have a working JSON config, but we still need to add one more thing - a flag dictating whether we are in production or development.

Wait a minute, isn't that a field in our config file?

Well, yes... technically it is, but what happens if we forget to provide a JSON config file in production? Our application will start up using the default config!

This is great for development, but in production we absolutely want to make sure we use a JSON config file, as our default values are unlikely to be secure for production use. After adding a production flag to our application, we are going to use it to ensure that all production deployments of our application MUST use a JSON config file, otherwise they will panic when starting up.

We'll start by updating our LoadConfig function to accept a parameter dictating whether or not the config is required.

```
$ atom config.go
```

```
func LoadConfig(configReq bool) Config {
    f, err := os.Open(".config")
    if err != nil {
        // This, and the new parameter, are the only changes
        // necessary.
        if configReq {
            panic(err)
        }
        fmt.Println("Using the default config...")
        return DefaultConfig()
    }
    // ... the rest remains unchanged
}
```

Now we need to create our flag, which means we need to take a moment to explore the **flag**⁷ package.

⁷<https://golang.org/pkg/flag/>

Note: Take a moment to actually look at the docs for the flag package. See if you can discover how it works on your own. It may take some practice, but learning to use new packages with only their official docs is a valuable skill to have.

The flag package can be used to write applications that accept flags whenever they are run. For instance, you might pass in the **-port=123** flag to dictate the port of an application if it had a flag defined like below.

```
var port = flag.Int("port", 3000, "the port of the server")
```

If the port flag was provided, it would use whatever value you provided, but otherwise it would default to 3000. The last part of the flag.Int function call is the help message that is displayed when a user provides the **--help** flag. The flags package automatically sets this flag up for us when we use it.

In our case we only need a single boolean flag dictating whether we are in production or not. Open up your main source file where we will be adding it.

```
$ atom main.go
```

This will need to be the first thing we run inside our main function, and we will also need to call the **Parse** function provided by the flag package to ensure our flags are all parsed correctly.

```
func main() {
    boolPtr := flag.Bool("prod", false, "Provide this flag "+
        "in production. This ensures that a .config file is "+
        "provided before the application starts.")
    flag.Parse()
    // boolPtr is a pointer to a boolean, so we need to use
    // *boolPtr to get the boolean value and pass it into our
    // LoadConfig function
    cfg := LoadConfig(*boolPtr)
    dbCfg := cfg.Database
```

```
// ... nothing else changes  
}
```

We can provide the `--help` flag to our application to verify that our flag is being parsed correctly.

```
$ go run *.go --help  
Usage of /var/.../exe/config:  
-prod  
    Provide this flag in production. This ensures that a  
    .config file is provided before the application starts.
```

Box 16.4. Whats the deal with the "Usage of..." part?

The long “Usage of” string looks odd because `go run` will build a temporary binary and then execute it. `go run` doesn’t interpret your code like python or ruby would, but instead build then runs it.

If you wanted to build a binary named `app` you could instead use the following commands:

```
$ go build -o app *.go  
$ ./app --help  
Usage of ./app:  
-prod  
    Provide this flag in production. This ensures that a  
    .config file is provided before the application starts.
```

Now when we want to start our application we can provide the `-prod` flag which ensures we load a `.config` JSON file. We can verify this by deleting or moving our `.config` file and trying to run our application with the `prod` flag.

```
$ mv .config unused.config  
$ go run *.go -prod  
panic: open .config: no such file or directory
```

Source code

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

Completed - book.usegolang.com/16.6.3

Changes - book.usegolang.com/16.6.3-diff

16.7 Setting up a server

We are finally ready to set up a production server and deploy our application! This section will be split into five major steps:

1. Initializing a server
2. Installing PostgreSQL on our server
3. Installing Go on our server
4. Building and setting up Caddy
5. Creating a deploy script that builds and deploys our web application and running it

Each of these steps is detailed in a sub-section below.

16.7.1 Digital Ocean droplet

In this section we are going to discuss setting up a server on Digital Ocean (called a droplet on DO), but in reality you could use almost any hosting provider you want. You could even set this up using a computer you have in your garage or home. The only real requirement is that the server has an IP address we can SSH into, and that it is a fresh install of Ubuntu 16.04 x64.

Note: Ubuntu 14.04 won't work because it doesn't use systemd, which we will be using to ensure our programs restart on reboot and when they crash. Ubuntu 17.04 may work, but I haven't verified.

Box 16.5. Free Digital Ocean credits

If you haven't ever used Digital Ocean before (or if you create a new account) you can get some free credits to start out by using a referral link. My own referral link - do.co/webdevgo - will get you \$10 in free credit, and you may be able to find a better referral deal if you search online. Regardless, \$10 should be plenty to get you started with a \$5/mo server for two months. We won't need a larger server than this starting out.

Assuming you have a Digital Ocean account, you are going to want to navigate to the new droplet page. From here you want to select the following options:

Once you have everything setup go ahead and click the create button and wait for your server to be setup.

When your server is ready you should be able to find an ipv4 port number for your server. In Digital Ocean it is often in the top left, right underneath the droplet name and clicking on it will copy it.

For the sake of simplicity, the following samples will all proceed assuming that

Table 16.1: Digital Ocean droplet settings

Setting	Values to select
Image	Under the "Distributions" tab select Ubuntu 16.04 x64. If the version has a minor version number, eg 16.04.3, that should be fine.
Size	Choose whatever you want. The cheapest server option should be fine.
Block storage	We won't be using this feature.
Datacenter region	Pick whichever you prefer. The one closest to your physical location is probably ideal, but if you plan to target customers in another region you may want to select differently.
Additional options	None of these are required for our app right now, and you *might* be charged for a few of these (like backups), but I typically select the IPv6 and Monitoring options.
SSH keys	If you have any SSH keys setup, these will enable you to login without a password but instead with your SSH key. This is by far the simplest option, but you may need to follow a tutorial explaining how to set this up (https://book.usegolang.com/do-ssh-tut). If you opt to not do this you will be emailed a password what is required to login to your server and will likely be required every time you deploy your server.
How many droplets?	We will only need 1 droplet
Hostname	Whatever you want. I often use something like the domain name I plan to use, eg "www.lenslocked.com" but anything is fine.

the IP address for our server is **123.123.22.33**, but you will want to replace this IP address with your own in any upcoming coding snippets.

Now if you want to use a domain name, we need to setup an “A” record⁸ mapping to our IP address. Doing this will vary from provider to provider, but nearly all providers have tutorials explaining how to do this. You typically only need a name (which is the subdomain you are using, if any), an IP address, and a TTL (time to live) which dictates how long DNS servers should cache this record for. I typically suggest a short TTL until things are setup correctly, then you can switch to a longer one.

Once you have either setup an A record for your domain or decided you aren’t going to you will want to verify you can SSH into your server.

```
$ ssh root@123.123.22.33  
# or if you setup a domain:  
$ ssh root@subdomain.yourdomain.com
```

Note: You may be asked about the authenticity of a host - if so type “yes” and hit enter.

If all went well, you should see a welcome message indicating that you have SSHed into your server. If you want to exit you can either press **ctrl+d** (even on Mac), or by typing “logout”.

16.7.2 Installing PostgreSQL in production

Once your server is setup we are going to setup Postgres. We will need to SSH into our server to do this, so do that if you haven’t already.

⁸<https://support.dnsimple.com/articles/a-record/>

```
$ ssh root@123.123.22.33
```

Box 16.6. Do not memorize this setup

Before we proceed I want to note that while it might feel like developers know how to do things like install Postgres by heart, the truth is most of them are just good at finding the right docs and tutorials whenever they need them. I myself know roughly what steps are involved in setting up Postgres, but I verify things nearly every time I install PostgreSQL. You shouldn't be discouraged if you don't remember all of these steps by heart. That is completely normal!

Box 16.7. External resources

If this guide ever feels out of date, feel free to reference my personal website where I maintain a few articles walking through how to install various versions of PostgreSQL on different operating systems.

<https://www.calhoun.io/using-postgresql-with-golang/>

Installation

The first thing we want to do before we start installing anything is to update **apt-get**. This will ensure that we are installing from an updated repository.

```
$ sudo apt-get update
```

After that finishes, we can go ahead and install all of the packages we are going to need.

```
$ sudo apt-get install postgresql postgresql-contrib
```

When you are prompted asking if we want to continue, type **y** and hit enter. This will go ahead and install both the `postgres` package and the `postgres-contrib` package, which adds some additional functionality to Postgres.

We now have Postgres installed, but by default you need to be logged into the `postgres` user account to access PostgreSQL. That isn't what we want - instead we want to set a password for the `postgres` role and then use that password to log into Postgres.

Setting up a password for the `postgres` role

By default, local connections to PostgreSQL use the **peer** authentication system. That means that instead of asking for a password, they check to see if we are currently logged into a user (a linux user) that matches a user name in Postgres.

We are going to change the way we do authentication and instead tell Postgres to use an encrypted password, but first we need to actually set a password for the `postgres` user. To do this we need to open up **psql** as the user `postgres`.

```
$ sudo -u postgres psql
```

We should see output that looks something like this:

```
could not change directory to "/root": Permission denied
psql (9.5.4)
Type "help" for help.

postgres=#
```

Don't worry about the permission denied error. This isn't important right now. What is important is that we are now connected to Postgres and we want to change the password for the user postgres. To do this, we run the following SQL code.

```
ALTER USER postgres WITH ENCRYPTED PASSWORD 'xxxxxxxx';
```

*Note: Be sure to replace the **xxxxxxxx** with an actual password.*

What we are doing here is telling Postgres that we want to update the user postgres by setting an encrypted password of **xxxxxxxx**. If we did this correctly we will get the following output:

```
ALTER ROLE
postgres=#
```

That is all we need to do inside of Postgres. Go ahead and quit by pressing **ctrl-d**.

Now that we have a password set for the postgres user we want to update Postgres to use this password. To do this we need to edit the pg_hba.conf file.

```
sudo nano /etc/postgresql/9.5/main/pg_hba.conf
```

Note: Feel free to replace nano with an editor of your choice, but we are using it here because I know it is already installed on our server.

Look for an uncommented line (a line that doesn't start with `#`) that has the contents shown below. The spacing will be slightly different, but the words should be the same.

```
local    all    postgres    peer
```

The last part of this line is what we want to change. This is what determines how we authenticate the `postgres` user when making a local connection. Instead of `peer`, which uses your system user name to authenticate you, we want to use `md5` which uses an encrypted password for authentication. Replace the word `peer` with `md5`.

```
local    all    postgres    md5
```

Press `ctrl+x` (even on Mac) to close nano, then hit `y` to confirm that you want to save, and hit enter to confirm the same file name as the original.

Now we need to restart Postgres so the changes take effect.

```
$ sudo service postgresql restart
```

Now if we want to connect to PostgreSQL we can use the `postgres` user and a password.

```
$ psql -U postgres
```

Note: When prompted for your password, type it in.

Once we have logged into `postgres` using a password, the final step we need to take is creating our production database.

```
CREATE DATABASE lenslocked_prod;
```

Our PostgreSQL server is setup and ready to go. It is even setup to restart automatically if our server crashes or PostgreSQL crashes using systemd, which we will learn more about as we set up Caddy Server in the next section.

16.7.3 Installing Go

Next we are going to install Go on our server so that we can build our application's binary for the server.

Note: In previous versions of this course we also built Caddy from source. We will no longer be doing that as their license has changed to make it easier to just use their prebuilt binaries.

Our own web application could be built on our local computer in a variety of ways and then uploaded to our server as a binary, but we will be building directly on our production server because this tends to limit the number of potential issues. For instance, one person might have a library installed locally while another doesn't and a build might fail, but if every developer is building on the production server we know that it is always setup the same way.

Note: This is one of the benefits of Docker and multi-stage builds, but we won't be covering Docker in this book.

Before we can build anything, we first need to install Go. We will be installing Go 1.15, which as of this writing is the latest version, and we will be following instructions provided by the Go docs⁹.

SSH into your server if you aren't already connected.

⁹<https://golang.org/doc/install>

```
$ ssh root@123.123.22.33
```

The first step we are going to take is to download a gzip (a compressed archive) that has Go already built for our machine. The Go team provides pre-built versions of Go for most operating systems, and you can specify the version you need with the URL you download from.

```
https://storage.googleapis.com/golang/gol.15.linux-amd64.tar.gz
          ^^^   ^^^   ^^^
          Version   OS     Architecture
```

In our case we will be installing Go version 1.9, for Linux, with the AMD64 architecture.

Note: The architecture of AMD64 is commonly referred to as x64, and is the instruction set used by a 64 bit processor. Even if you are using an Intel processor, you would still likely use this architecture for your builds.

We can download the file and extract it by running the following two lines in our SSH session with our server.

```
$ curl -O https://storage.googleapis.com/golang/gol.15.linux-amd64.tar.gz
$ tar -C /usr/local -xzf gol.15.linux-amd64.tar.gz
$ export PATH=$PATH:/usr/local/go/bin
```

When we run the second line - the **tar** line - we are telling our server to extract the archive into the /usr/local directory. This ends up creating the /usr/local/go/bin directory, and inside of it we have our Go binary that our server uses to run go commands.

The last line, the **export** line, is used to add the Go binary to our PATH environment variable. This basically enables us to type **go** from anywhere on our

server and it will know to run the program named **go** inside one of the directories listed in the PATH variable.

With Go installed we are going to make a directory to store our source code in. This *cannot* be changed without setting a custom \$GOPATH env variable, so follow along exactly if you are unfamiliar with what we are doing.

```
$ cd ~  
$ mkdir -p go/src
```

Test it out by running **go version**.

```
$ go version
```

Alternatively, you can define the absolute path to the binary when running the **go** command.

```
$ /usr/local/go/bin/go version
```

16.7.4 Setting up Caddy

Caddy¹⁰ is an HTTP web server written in Go that we will be using to setup HTTPS for free using Let's Encrypt. Caddy does all of this for us, allowing us to just set up our DNS server and then create a simple Caddyfile (a config file for Caddy) and start up Caddy.

Before we can use Caddy, we are going to need to install it. SSH into your server if you haven't already.

¹⁰<https://caddyserver.com/>

```
$ ssh root@123.123.22.33
```

Next we are going to follow the Caddy installation instructions¹¹ for Ubuntu.

```
$ sudo apt install -y debian-keyring debian-archive-keyring apt-transport-https
$ curl -lsLf \
'https://dl.cloudsmith.io/public/caddy/stable/cfg/gpg/gpg.155B6D79CA56EA34.key' \
| sudo apt-key add -
$ curl -lsLf \
'https://dl.cloudsmith.io/public/caddy/stable/cfg/setup/config.deb.txt?distro=debian&version=an' \
| sudo tee -a /etc/apt/sources.list.d/caddy-stable.list
$ sudo apt update
$ sudo apt install caddy
```

Now verify the install was successful.

```
$ caddy version
v2.3.0 h1:fnrqJLa3G5vfxcxmOH/+kJ0cunPLhSBnjgIvjXV/QTA=
```

Your version may be slightly different, but the command should print out a Caddy version.

You can also verify your installation by visiting your server in the browser. Simply type in the domain you set up with an A record, or the server's IP address.

You should see a landing page similar to the one in the following figure.

The commands we used to install both Caddy and Postgres both included systemd¹² services for each. systemd is used in Ubuntu 16.04 to manage system processes. Most notably, it is used to make sure that if Caddy or Postgres were to crash, that they are restarted. Or when our server reboots, it ensures that these services are started back up.

¹¹<https://caddyserver.com/docs/install#static-binaries>

¹²<https://en.wikipedia.org/wiki/Systemd>

[H]

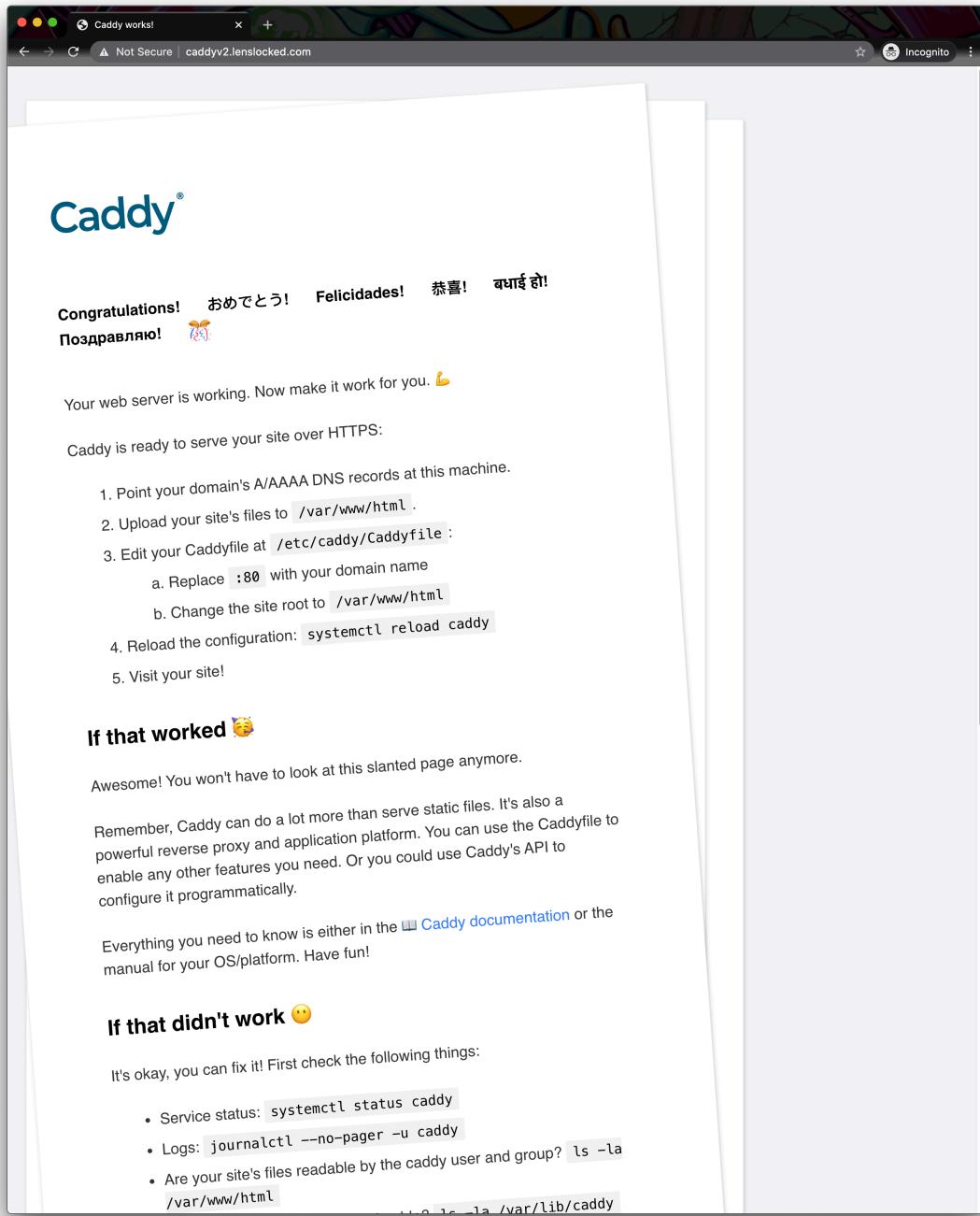


Figure 16.1: Caddy's default landing page

If you want to view the Caddy service file, you can use the **cat** command to print out the service file.

```
$ cat /etc/systemd/system/multi-user.target.wants/caddy.service

# caddy.service
#
# For using Caddy with a config file.
#
# Make sure the ExecStart and ExecReload commands are correct
# for your installation.
#
# See https://caddyserver.com/docs/install for instructions.
#
# WARNING: This service does not use the --resume flag, so if you
# use the API to make changes, they will be overwritten by the
# Caddyfile next time the service is restarted. If you intend to
# use Caddy's API to configure it, add the --resume flag to the
# `caddy run` command or use the caddy-api.service file instead.

[Unit]
Description=Caddy
Documentation=https://caddyserver.com/docs/
After=network.target network-online.target
Requires=network-online.target

[Service]
User=caddy
Group=caddy
ExecStart=/usr/bin/caddy run --environ --config /etc/caddy/Caddyfile
ExecReload=/usr/bin/caddy reload --config /etc/caddy/Caddyfile
TimeoutStopSec=5s
LimitNOFILE=1048576
LimitNPROC=512
PrivateTmp=true
ProtectSystem=full
AmbientCapabilities=CAP_NET_BIND_SERVICE

[Install]
WantedBy=multi-user.target
```

You don't need to understand everything this file is doing, but one important bit to notice is the location of the **Caddyfile**, which is `/etc/caddy/Caddyfile`. This service is designed to read this file when configuring Caddy, so if we want to customize it for our own app and domain we need to edit the Caddyfile.

Note: This is the same file mentioned by the Caddy landing page.

We can view the contents of this file with `cat`.

```
$ cat /etc/caddy/Caddyfile

# The Caddyfile is an easy way to configure your Caddy web server.
#
# Unless the file starts with a global options block, the first
# uncommented line is always the address of your site.
#
# To use your own domain name (with automatic HTTPS), first make
# sure your domain's A/AAAA DNS records are properly pointed to
# this machine's public IP, then replace the line below with your
# domain name.
:80

# Set this path to your site's directory.
root * /usr/share/caddy

# Enable the static file server.
file_server

# Another common task is to set up a reverse proxy:
# reverse_proxy localhost:8080

# Or serve a PHP site through php-fpm:
# php_fastcgi localhost:9000

# Refer to the Caddy docs for more information:
# https://caddyserver.com/docs/caddyfile
```

The rest of this section will involve setting up our Caddyfile. We won't be using the contents of the Caddyfile we just saw, but you should definitely read it over if you want to get a feel for how a Caddyfile works.

We will be creating our Caddyfile in our web application's source directory so that developers can make changes without logging into a production server. The changes to the Caddyfile will be pushed as part of our application's deployment process.

Log out of the server and head back to your local source code.

In your local source code create a Caddyfile.

```
$ atom Caddyfile
```

Your Caddyfile will vary based on the domain you use, or whether you are even using one. Below are two potential Caddyfiles, the first is intended for use with a real domain and the second will allow you to access your web application via just the IP address in your browser when we deploy.

```
# This is the domain we are using. Your domain will be different, or
# it might just be a port. You can also use the server's IP address
# and https should work with that.
www.lenslocked.com

# This enables gzip compression.
encode gzip

# This will redirect all incoming requests to the application running on
# our server at port :3000. In other words, it will forward all incoming
# requests to our Go application!
reverse_proxy localhost:3000
```

In this Caddyfile we define our domain and then define how Caddy should handle any incoming web requests to that domain. In our case we are asking it to gzip (compress) data before sending it back to end users to lower bandwidth required, and we are reverse proxying all requests to **localhost:3000**, which is where our Go app is listening for web requests.

Reverse proxying web request basically means our Caddy server will take any incoming web requests and forward them to our web application that will be running on the same server (localhost) on port 3000. By doing this, we could potentially have four or five servers all running on different ports, and have Caddy decide which server a user is trying to access by looking at the domain and proxying the request to the proper localhost:port combination.

Our Caddyfile without a domain is basically the same, but with your server's IP address.

```
# This is the domain we are using. Your domain will be different, or
# it might just be a port. You can also use the server's IP address
# and https should work with that.
123.123.22.33

# This enables gzip compression.
encode gzip

# This will redirect all incoming requests to the application running on
# our server at port :3000. In other words, it will forward all incoming
# requests to our Go application!
reverse_proxy localhost:3000
```

We won't push this to our server just yet, but will handle it in the next section when we create our deploy script.

Source code

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

Completed - book.usegolang.com/16.7.4

Changes - book.usegolang.com/16.7.4-diff

16.7.5 Creating a service for our app

Just like Caddy and Postgres have systemd services, we are going need one for our Go application. Our service will handle starting up our web application, restarting it if the server reboots, or if it crashes for any reasons.

To start creating the service, SSH into your server if you haven't already.

```
$ ssh root@123.123.22.33
```

Create a service file for our application.

```
$ sudo nano /etc/systemd/system/lenslocked.com.service
```

We are going to fill this in with a systemd service definition very similar to the ones we have seen, but a little simpler and designed specifically for our web application.

```
[Unit]
Description=lenslocked.com app

[Service]
WorkingDirectory=/root/app
ExecStart=/root/app/server -prod
Restart=always
RestartSec=30

[Install]
WantedBy=multi-user.target
```

Box 16.8. Explaining our service file

Our service file is broken into three sections: Unit, Service, and Install.

The first, unit, is used to describe the service we are creating. In our case we are only providing a general description.

In the second section we are defining how our service will work. The WorkingDirectory line defines where our service should start. That is, what folder should it **cd** to before running any commands.

ExecStart is the command our service should run. We will be using the absolute path to our Go server binary to avoid any issues.

The next two lines describe when and how often to restart if our service crashes. We are going to limit our restarts to once every 30 seconds, and tell the service to always restart our app.

In the final section, the install section, we are stating how our service should be enabled. You generally won't need to customize this section and details on how to are outside the scope of this book.

Our service is ready to run the binary `/root/app/server`, but we haven't built the binary yet. Don't worry though; whenever we write our deploy script it will upload our source code to the server and build the binary for us. Then we can restart our systemd service with the binary to have a new version of our web application released.

Now we need to reload systemctl and enable our service. Again, if this doesn't make a ton of sense that is okay. Most of these things only need done once then you won't need to mess with them again.

```
$ systemctl daemon-reload  
$ systemctl enable lenslocked.com.service
```

We aren't going to try to run this service yet because we don't have a binary to run. We will learn how to create that and restart (or start for the first time) our service in the next section.

16.7.6 Setting up a production config

We are going to need a config file on our production server for our application to use. Unlike the Caddyfile, this is a file I DO NOT suggest storing in your source control. Instead, it should be stored only on the production server and with very trusted individuals.

SSH into your server if you haven't already.

```
$ ssh root@123.123.22.33
```

We are going to create the `.config` file inside the `/root/app` directory where our application will be run from. We also need to create that directory.

```
$ mkdir /root/app
$ nano /root/app/.config
```

Fill it in with a valid config file for your application. Most of the necessary changes are shown below, but it is a good idea to generate your own pepper and HMAC key. The password used for PostgreSQL will likely vary as well.

```
{
  "port": 3000,
  "env": "prod",
  "pepper": "replace-this-with-a-secret-pepper",
  "hmac_key": "replace-this-with-a-secret-hmac-key",
  "database": {
    "host": "localhost",
    "port": 5432,
    "user": "postgres",
    "password": "replace-this-with-the-password-from-earlier",
    "name": "lenslocked_prod"
  }
}
```

Note: Your database password will be the one you set when running the `ALTER USER ...` command while setting up PostgreSQL.

Press **ctrl-x** to exit nano, then **y** to confirm you want to save and finally press **enter** to confirm we want to use the original filename provided.

16.7.7 Creating a deploy script

In this section we are going to create a deploy script that will:

1. Copy our source code to our production server.
2. Build our web application on the production server.
3. Copy the binary and any assets (images, css, etc) over to the **/root/app** directory.
4. Copy our Caddyfile to the correct location.
5. Restarts our web application on the production server.

Note: Our deploy script is going to be a bash file that hasn't been tested in windows. If you have trouble, feel free to reach out for help.

First we need to create our scripts directory where we will store our deploy script and any others we may make in the future. Once we have a directory we are going to create a bash release script file and give it modify the file to be executable, which allows us to run it instead of just opening it as a text file.

```
$ mkdir scripts
$ atom scripts/release.sh
# Save the file then...
# Make it executable
$ chmod +x scripts/release.sh
```

Note: We will be doing this in our own source code, so be sure to log out of your production server first.

We will start by telling our computer that this is a bash script that should be run with the **bash** binary. This line should be at the very top of your deploy script.

```
#!/bin/bash
```

Next we are going to make sure we are working from the correct source code directory.

```
# Change to the directory with our code that we plan to work from
# NOTE: DO NOT INCLUDE THIS LINE IF YOU ARE USING GO MODULES!
cd "$GOPATH/src/lenslocked.com"
```

After that we can start the release by deleting any local binaries that exist so they aren't uploaded. I tend to call mine lenslocked.com when I build them, so that is what I will be deleting. We are NOT deleting the lenslocked.com directory, but a binary inside of it named lenslocked.com.

```
echo "===== Releasing lenslocked.com ====="
echo " Deleting the local binary if it exists (so it isn't uploaded)..."
rm lenslocked.com
echo " Done!"
```

We are going to upload our source code to the server to build, but first we should delete any source code on the server to make sure we don't have any files accidentally leftover from a previous build.

```
echo " Deleting existing code..."
ssh root@123.123.22.33 "rm -rf /root/go/src/lenslocked.com"
echo " Code deleted successfully!"
```

After that we can upload our source code. We will be using **rsync**¹³, an incredibly useful tool for transferring and synchronizing files across different servers. If you haven't ever used it I highly recommend digging further into using it.

¹³<https://en.wikipedia.org/wiki/Rsync>

For now what I'll say is that the flags we are using cause rsync to print out a little more information, to work recursively (getting files in nested folders), and the exclude flags tell it not to sync files from a few directories that aren't intended to be copied to our server.

```
echo " Uploading code..."
# The \ at the end of the line tells bash that our
# command isn't done and wraps to the next line.
rsync -avr --exclude '.git/*' --exclude 'tmp/*' \
--exclude 'images/*' ./ \
root@123.123.22.33:/root/go/src/lenslocked.com/
echo " Code uploaded successfully!"
```

Once our code is uploaded we are ready to start running commands on the server and building our code. The first step in our build process is to **go get** any third party libraries our application is using. To do this we will use the **ssh** command and pass in whatever commands we need our server to execute as an additional argument.

```
echo " Go getting deps..."
ssh root@123.123.22.33 "export GOPATH=/root/go; \
/usr/local/go/bin/go get golang.org/x/crypto/bcrypt"
ssh root@123.123.22.33 "export GOPATH=/root/go; \
/usr/local/go/bin/go get github.com/gorilla/mux"
ssh root@123.123.22.33 "export GOPATH=/root/go; \
/usr/local/go/bin/go get github.com/gorilla/schema"
ssh root@123.123.22.33 "export GOPATH=/root/go; \
/usr/local/go/bin/go get github.com/lib/pq"
ssh root@123.123.22.33 "export GOPATH=/root/go; \
/usr/local/go/bin/go get github.com/jinzhu/gorm"
ssh root@123.123.22.33 "export GOPATH=/root/go; \
/usr/local/go/bin/go get github.com/gorilla/csrf"
ssh root@123.123.22.33 "export GOPATH=/root/go; \
/usr/local/go/bin/go get gopkg.in/mailgun/mailgun-go.v1"
```

*Note: Whenever you add new packages to your source code you will need to update this section of your release script, or alternatively you could look into using a package manager like **dep**¹⁴.*

¹⁴<https://github.com/golang/dep>

With all of our third party packages installed we can build our own application. We are going to use the `-o` flag to define the name of our resulting binary, and we will build from within our `/root/app` directory so that our binary is created there.

```
echo " Building the code on remote server..."  
ssh root@123.123.22.33 'export GOPATH=/root/go; \  
cd /root/app; \  
/usr/local/go/bin/go build -o ./server \  
$GOPATH/src/lenslocked.com/*.go'  
echo " Code built successfully!"
```

Once our server is built we need to move all of our assets, views, and Caddyfile into the `/root/app` directory. In this case we will NOT be deleting existing files because we don't want to delete any images uploaded by end users and extra assets shouldn't have a negative impact on our application.

```
echo " Moving assets..."  
ssh root@123.123.22.33 "cd /root/app; \  
cp -R /root/go/src/lenslocked.com/assets ."  
echo " Assets moved successfully!"  
  
echo " Moving views..."  
ssh root@123.123.22.33 "cd /root/app; \  
cp -R /root/go/src/lenslocked.com/views ."  
echo " Views moved successfully!"  
  
echo " Moving Caddyfile..."  
ssh root@screencast.lenslocked.com "cp /root/go/src/lenslocked.com/Caddyfile /etc/caddy/Caddyfile"  
echo " Caddyfile moved successfully!"
```

With our binary built and our files moved over we are ready to restart the service we created in the last subsection. We will also restart the caddy service in case we pushed any changes to our Caddyfile and print out a “Done releasing” message.

```
echo "  Restarting the server..."  
ssh root@123.123.22.33 "sudo service lenslocked.com restart"  
echo "  Server restarted successfully!"  
  
echo "  Restarting Caddy server..."  
ssh root@123.123.22.33 "sudo service caddy restart"  
echo "  Caddy restarted successfully!"  
  
echo "===== Done releasing lenslocked.com ====="
```

*Note: The **restart** command will start a service if it was not already running, or stop and start a service that was running.*

With that we are done creating our deploy script. Navigate to your lenslocked.com directory and run it with the following:

```
$ ./scripts/release.sh
```

After your release script finishes your web application will be live on your server. For instance, if your server had the IP address **123.123.22.33** and you *did not* setup your server to use a domain, you should be able to type that into your browser and see the application running. If you did setup a domain and create an A record to your server that should work instead, along with HTTPS!

Congrats on releasing your first web application!

Source code

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

Completed - book.usegolang.com/16.7.7

Changes - book.usegolang.com/16.7.7-diff

Chapter 17

Filling in the gaps

The final chapter of the book is going to be a bit different than the previous chapters, as most of the code we will be looking at involves adding optional features that could be implemented in a variety of ways.

For instance, when we start to look at sending emails we will be using Mailgun to send them, but there are plenty of other emailing services out there and in some companies they will have their own internal solution. Regardless, it is still useful to see an example of how you might send emails to your users.

17.1 Deleting cookies and logging out users

Our users are likely going to want a way to sign out at some point, so let's look at how to implement that feature.

Creating a sign out link will entail roughly two major steps:

1. Adding a button to sign out to the navigation bar whenever a user is signed in.

2. Adding a controller and route to process a sign out and delete the user's session cookie.

We already saw how to customize our navigation bar based on whether a user is signed in or not when we added a link to the galleries index page. Adding a link for signing out will be roughly the same, but because signing out is a destructive action (it deletes your session cookie) we are going to put this inside of a form so a user has to perform an HTTP POST to sign out.

Open up the navbar template first.

```
$ atom views/layouts/navbar.gohtml
```

First we are going to add a logoutForm template. We will need to include a csrfField in this because it is a form.

```
 {{define "logoutForm"}}
<form class="navbar-form navbar-left" action="/logout" method="POST">
  {{csrfField}}
  <button type="submit" class="btn btn-default">Log out</button>
</form>
{{end}}
```

*Note: We are using the **navbar-form** class here to make sure the form is rendered correctly inside of a navbar.*

After adding the template, find the **navbar-right** section of the navbar where we have our “Log In” and “Sign Up” links. We are going to update this to display a logout form if a user is signed in, otherwise we will render the existing links.

```
<ul class="nav navbar-nav navbar-right">
  {{if .User}}
    <li>{{template "logoutForm"}}</li>
```

```

{{else}}
<li><a href="/login">Log In</a></li>
<li><a href="/signup">Sign Up</a></li>
{{end}}
</ul>

```

Next we need to add a controller to handle logging a user out. This is done by deleting the user's session cookie, which really just means telling the browser to expire it. Just to be certain, we will also update the user's remember token so even if the cookie was retained it will no longer have a valid token.

```
$ atom controllers/users.go
```

```

// Logout is used to delete a user's session cookie
// and invalidate their current remember token, which will
// sign the current user out.
//
// POST /logout
func (u *Users) Logout(w http.ResponseWriter, r *http.Request) {
    // First expire the user's cookie
    cookie := http.Cookie{
        Name:      "remember_token",
        Value:     "",
        Expires:   time.Now(),
        HttpOnly:  true,
    }
    http.SetCookie(w, &cookie)
    // Then we update the user with a new remember token
    user := context.User(r.Context())
    // We are ignoring errors for now because they are
    // unlikely, and even if they do occur we can't recover
    // now that the user doesn't have a valid cookie
    token, _ := rand.RememberToken()
    user.Remember = token
    u.us.Update(user)
    // Finally send the user to the home page
    http.Redirect(w, r, "/", http.StatusFound)
}

```

Finally, we need to add a route for the controller.

```
$ atom main.go
```

```
r.HandleFunc("/logout",
    requireUserMw.ApplyFn(usersC.Logout)).
    Methods("POST")
```

Now our users can log out whenever they want!

Source code

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

Completed - book.usegolang.com/17.1

Changes - book.usegolang.com/17.1-diff

17.2 Redirecting with alerts

Whenever redirecting a user to a new page, it is common to want to also add an alert of some sort as well. For instance, after a user uploads new images to a gallery it is frequently easier to redirect them to the edit gallery page so that their URL is accurate, but it might also be useful to provide a notification that the images were uploaded or to inform a user that they tried to upload a duplicate.

Our current application does not handle both alerts and redirects at the same time. Whenever a user is redirected, any alert information is lost, and whenever we render a view with an alert we can't also redirect the user's URL. In this section we are going to explore how to add this feature.

Adding redirects with alerts can be done in roughly three steps:

1. First we must develop a way to persist, retrieve, and clear alerts between requests. In our case we will be using cookies to store this information, much like we did for our user's remember token.
2. Once we have alerts persisted between requests we are going to create a function that makes it easier to persist an alert and then redirect the user.
3. Finally, we will need to update our views package to look for persisted alerts and use them when rendering the next view. We will also want to clear the alert after using it so we don't see the same alert multiple times.

We will cover each of these steps below, and all of the code we will be adding will belong to the `views` package, as it all relates to rendering the UI.

Persisting, retrieving, and clearing alerts

The first part is the “trickiest”, which really just means that it involves the most code. Everything we will be doing in this sub-section should be relatively familiar as we have already learned to create, retrieve, and delete cookies which is all we will be doing.

Let's start with a function used to persist alerts.

```
$ atom views/data.go
```

```
import (
    // Add these two imports
    "net/http"
    "time"
)

func persistAlert(w http.ResponseWriter, alert Alert) {
```

```
// We don't want alerts showing up days later. If the
// user doesn't load the redirect in 5 minutes we will
// just expire it.
expiresAt := time.Now().Add(5 * time.Minute)
lvl := http.Cookie{
    Name:      "alert_level",
    Value:     alert.Level,
    Expires:   expiresAt,
    HttpOnly:  true,
}
msg := http.Cookie{
    Name:      "alert_message",
    Value:     alert.Message,
    Expires:   expiresAt,
    HttpOnly:  true,
}
http.SetCookie(w, &lvl)
http.SetCookie(w, &msg)
}
```

Next we need a way to clear these alerts. The simplest way to do this is often just to set the expiration date to the current time so the cookies are expired immediately.

```
func clearAlert(w http.ResponseWriter) {
    lvl := http.Cookie{
        Name:      "alert_level",
        Value:     "",
        Expires:   time.Now(),
        HttpOnly:  true,
    }
    msg := http.Cookie{
        Name:      "alert_message",
        Value:     "",
        Expires:   time.Now(),
        HttpOnly:  true,
    }
    http.SetCookie(w, &lvl)
    http.SetCookie(w, &msg)
}
```

And finally we will want to be able to retrieve an alert that has been persisted, so we will add a function to do that.

```
func getAlert(r *http.Request) *Alert {
    // If either cookie is missing we will assume the alert
    // is invalid and return nil
    lvl, err := r.Cookie("alert_level")
    if err != nil {
        return nil
    }
    msg, err := r.Cookie("alert_message")
    if err != nil {
        return nil
    }
    alert := Alert{
        Level:  lvl.Value,
        Message: msg.Value,
    }
    return &alert
}
```

Redirect function

In order to redirect a user with an alert we need to first persist an alert, then redirect the user. Rather than typing this manually in our controllers we will write a single function to handle doing both steps for us.

```
// RedirectAlert accepts all the normal params for an
// http.Redirect and performs a redirect, but only after
// persisting the provided alert in a cookie so that it can
// be displayed when the new page is loaded.
func RedirectAlert(w http.ResponseWriter, r *http.Request, urlStr string, code int, alert Alert)
    persistAlert(w, alert)
    http.Redirect(w, r, urlStr, code)
}
```

Anytime we want to redirect a user with an alert we can call the `RedirectAlert` function instead of the `http.Redirect` function.

```
alert := views.Alert{
    Level:  views.AlertLvlSuccess,
    Message: "Welcome to LensLocked.com!",
```

```

}
views.RedirectAlert(w, r, "/galleries", http.StatusFound, alert)

```

Note: This last bit of code is an example of how you might redirect a user with an alert after they sign up for a new account.

Rendering persisted alerts

Without informing our views of our persisted alerts they wouldn't serve any purpose. We need to update the **Render** method of the **View** type to inform it of the alerts and have it check for any persisted alerts. It will also be responsible for clearing any alerts after rendering them so a user doesn't see the same alert several times in a row.

```
$ atom views/view.go
```

Find the Render method and update it with the following code.

```

func (v *View) Render(w http.ResponseWriter, r *http.Request, data interface{}) {
    w.Header().Set("Content-Type", "text/html")
    var vd Data
    switch d := data.(type) {
        // ... this is unchanged
    }
    // Lookup the alert and assign it if one is persisted
    if alert := getAlert(r); alert != nil {
        vd.Alert = alert
        clearAlert(w)
    }
    vd.User = context.User(r.Context())
    // ... everything below this point is unchanged
}

```

Note: All we are adding here is the “if alert := getAlert(r) ... ” block which is 4 lines long. Everything else in the Render method remains unchanged.

We now have all the tools necessary to redirect users with an alert message.

Source code

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

Completed - book.usegolang.com/17.2

Changes - book.usegolang.com/17.2-diff

17.3 Emailing users

Emailing users is a tricky feature to write about because everyone does it differently. Some companies write their own internal tools to manage sending emails to customers, while others might use a variety of services like Mailgun and Sendgrid.

I personally tend to use Mailgun, which is free for your first 10k emails (as of this writing) while also being simple to use. As a result, that is what we will be using in this section of the course.

Box 17.1. Why don't we use the `net/smtp` package?

It is inevitable that some readers will be upset by my decisions to use a third party service in this book rather than covering the `net/smtp` package in the standard library.

I am opting to use a third party service because the purpose of this course is to teach you how to build a real web application, and real web applications that I have been

involved with almost all use a third party service for email delivery. Not only is it easier to avoid your emails from being flagged as spam this way, but their APIs almost always make it easier to deliver email without having to learn about the specifics of SMTP, which is not the purpose of this course.

The first thing you will need to do is head over to their website, create an account, and setup a domain to send email from.

<https://www.mailgun.com/>

After your account is setup, we need to install the Go client provided by the Mailgun team.

```
$ go get gopkg.in/mailgun/mailgun-go.v1
```

Next you will need to update your config source code to add a section for Mailgun.

```
$ atom config.go
```

```
type Config struct {
    Port      int          `json:"port"`
    Env       string       `json:"env"`
    Pepper    string       `json:"pepper"`
    HMACKey   string       `json:"hmac_key"`
    Database  PostgresConfig `json:"database"`
    Mailgun   MailgunConfig  `json:"mailgun"`
}

type MailgunConfig struct {
    APIKey     string `json:"api_key"`
    PublicAPIKey string `json:"public_api_key"`
    Domain     string `json:"domain"`
}
```

You could setup some default values for Mailgun, but I tend to avoid putting credentials for third party services into my default config. Instead, I recommend adding the mailgun section to your `.config` file.

```
{  
    ...  
    "database": {  
        ...  
    },  
    "mailgun": {  
        "api_key": "your-api-key",  
        "public_api_key": "your-public-key",  
        "domain": "your-domain-setup-with-mailgun"  
    }  
}
```

In order to create a mailgun client you would then use the config variables and the mailgun package.

```
import "gopkg.in/mailgun/mailgun-go.v1"  
  
cfg := LoadConfig(...)  
mgCfg := cfg.Mailgun  
mgClient := mailgun.NewMailgun(mgCfg.Domain, mgCfg.APIKey, mgCfg.PublicAPIKey)
```

We won't be using the client for a second, but it is necessary to send emails.

Next we would build an email to send.

```
from := "support@lenslocked.com"  
subject := "Welcome to LensLocked.com!"  
text := `Hi there!  
Welcome to our awesome website! Thanks for joining!  
  
Enjoy,  
LensLocked Support`  
to := "jon@calhoun.io"  
msg := mailgun.NewMessage(from, subject, text, to)
```

We could optionally provide a display name for any email address with the following format:

```
to := "Jon Calhoun <jon@calhoun.io>"
```

We can also attach an HTML version of the email to the message using the **SetHtml** method¹.

```
html := `Hi there!<br/>
Welcome to <a href="https://www.lenslocked.com">lenslocked.com</a>!

...
msg.SetHtml(html)
```

With the SetHtml method we could either add static HTML, or we could use the **html/template** package to generate dynamic HTML and embed that.

Once our email is built to our satisfaction we would use the Mailgun client we created earlier to send it.

```
respMsg, id, err := mgClient.Send(msg)
```

Most of the time we won't need to worry much about the response message or id, but we should check for errors to verify that the message was sent successfully.

Now that you have an idea of how to use the Mailgun client, try to integrate it into your application. Where would it make sense to share the client? Does it make sense to wrap the Mailgun code in a custom **email** package?

We won't be covering all of the code used for a complete integration here, but if you reference the source code at the end of this section you will see how I chose to integrate Mailgun and share it with my controllers so we can initiate sending emails whenever necessary.

¹<https://godoc.org/github.com/mailgun/mailgun-go#Message.SetHtml>

Source code

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

Completed - book.usegolang.com/17.3

Changes - book.usegolang.com/17.3-diff

17.4 Persisting and prefilling form data

One of the easiest ways to frustrate your users is to allow them to fill out a form, hit submit, and then lose all of the data they just provided. Right now that is exactly what our sign up page does when we provide any invalid data.

On a related note, it isn't uncommon to want to link a user to a page with some of the form already filled in. For instance, when you purchased this course you were given a link used to create an account on the members site and that link had your name, email address, and likely your license key already filled in for you. This is also very handy when helping a user reset their password.

In this section we will look at how to do each of these.

Persisting form data

The first thing we want to handle is not losing any form data whenever a user has an error. To do this, we need to:

1. Update our templates to use any data we provide to it as the default value for a form input.

2. Update our controller methods to pass this data into our **Render** method so it is available for the template to use.

This is all much clearer with an example, so let's update the **Create** method on the users controller that is called whenever a user attempts to sign up for a new account.

We will start by updating the template, allowing it to use the data if it is provided.

```
$ atom views/users/new.gohtml
```

First fine the line that calls the **signupForm** template and verify that it passes in all the template data to the signupForm template. That is, make sure there is a dot (`.`) as the last argument.

```
{{template "signupForm" .}}
```

Next we need to find the `<input ...>` tags inside the **signupForm** template. We won't be persisting the password for security reasons, but if the name and email are provided we are going to automatically set a value for each of these inputs. We can do that by setting the the **value** attribute of each HTML input field.

```
 {{define "signupForm"}}
<form action="/signup" method="POST">
  <!-- ... -->
  <input type="text" name="name" class="form-control"
    id="name" placeholder="Your full name"
    value="{{.Name}}>
    <!-- Add this last part - the value="{{.Name}}" part -->
  <!-- ... -->
  <input type="email" name="email" class="form-control"
    id="email" placeholder="Email" value="{{.Email}}>
```

```
<!-- Add this last part - the value="{{.Email}}" part -->
<!-- ... -->
</form>
{{end}}
```

Now that our template will take advantage of the additional data we need to update our Create method on the users controller to provide any known data.

```
$ atom controllers/users.go
```

There are a number of ways to implement this, but I personally prefer to set the **Yield** field of our view data to be a pointer to the SignupForm that we parse.

```
func (u *Users) Create(w http.ResponseWriter, r *http.Request) {
    var vd views.Data
    var form SignupForm
    vd.Yield = &form
    if err := parseForm(r, &form); err != nil {
        // ...
    }
    // ...
}
```

*Note: If we do not use a pointer here the Yield field will be assigned a copy of the form, so when we then proceed to parse it the original **form** variable will get updated with new values but the **vd.Yield** field will not be updated. By using a pointer we ensure that any time we change the **form** variable we will also see those changes in the **vd.Yield** field.*

Now we will have all the form data available in our view as long as we pass that **vd** variable in as the data.

To verify that the updated code is working, restart your server and fill out the sign up form with a password that is too short. Your web application should render the sign up form again with an error message, but this time your name and email address fields should retain the information you originally typed in.

Prefilling form data via URL params

Note: This section assumes you completed the edits to the new user template covered in the previous section.

Prefilling form data is very similar to persisting it; in fact, we have already made all the changes necessary to the new user template (new.gohtml) to support this. All we need to do now is update our **New** method on the users controller to parse any URL parameters with a SignupForm and then provide that data to the NewView when we render it.

To support this we first need to tweak our **parseForm** function; we are going to split it into two functions that achieve the same end result, but each only do part of the work.

```
$ atom controllers/helpers.go
```

Replace the existing **parseForm** function with the updated one below, and add the **parseValues** function.

```
// Add this import
import "net/url"

func parseForm(r *http.Request, dst interface{}) error {
    if err := r.ParseForm(); err != nil {
        return err
    }
    return parseValues(r.PostForm, dst)
}

func parseValues(values url.Values, dst interface{}) error {
    dec := schema.NewDecoder()
    dec.IgnoreUnknownKeys(true)
    if err := dec.Decode(dst, values); err != nil {
        return err
    }
    return nil
}
```

Notice that our code didn't change much here. We just needed to split the code into two functions so that we can now leverage the `parseValues` function with URL parameter values.

To support this we are going to add the `parseURLParams` function.

```
func parseURLParams(r *http.Request, dst interface{}) error {
    if err := r.ParseForm(); err != nil {
        return err
    }
    return parseValues(r.Form, dst)
}
```

The only difference between our `parseURLParams` and `parseForm` functions are which source of data they use. The former, `parseURLParams`, uses the `r.Form` which can be filled in by URL query parameters. The latter, `parseForm`, uses the `r.PostForm`.

Now we can head back over to our users controller and update the `New` method, which we use to render the sign up view, to use our new `parseURLParams` function to check for any form data that needs prefilled.

```
$ atom controllers/users.go
```

```
func (u *Users) New(w http.ResponseWriter, r *http.Request) {
    var form SignupForm
    parseURLParams(r, &form)
    u.NewView.Render(w, r, form)
}
```

Note: We aren't creating a `views.Data` variable here because we know that the `Render` method will do that for us if the data we pass in isn't of that type.

We can now create URLs with data already provided and our form will automatically populate! We will need to make sure that data is URL encoded, but we

will explore how to do that in the next section where we will be implementing the password reset feature.

To verify your code is working, visit the following URL and see if the email address field is automatically populated on your signup form:

[localhost:3000/signup?email=michael%40dundermifflin.com]<http://localhost:3000/signup?email=michael%40dundermifflin.com>

Source code

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

Completed - book.usegolang.com/17.4

Changes - book.usegolang.com/17.4-diff

17.5 Resetting passwords

We want users to be able to reset their password if they forget it. To do this, we are going to implement the following workflow:

1. Visit a page and submit the email address associated with the account that they forgot the password to.
2. If we find a user with that email address, we will create a password reset token and email it to the account's email address. It is important that we don't display this token in the browser or send it to another email address otherwise we might let attackers get into someone else's account.
3. The user would then visit their email and find the token, along with a link to our password reset form.

4. When the user goes to the password reset form we will ideally prefill in the remember token for them using URL query parameters in the URL we give them. The user would then type in a new password and submit the form.
5. After submitting the password reset form our server will look up the token and verify it is valid. Our definition of valid for now will be that the token is less than 12 hours old, and it is one we can find in our database. Assuming the token is valid, we would then update the user's password with the new password provided via the reset form.
6. Finally, our server would log the user into that account and notify them that the password has been updated.

In order to implement this we need to add quite a bit of code to our application. We will need a place to store the password reset tokens, code to verify that they haven't expired, code to email a user their password reset token, and code to render and process the forgot password and reset password forms.

While this may feel like a lot at first, most of it will be very similar to code we have already learned about. Rendering forms, for example, is something we have been doing for some time now.

In order to make implementing this a little easier we will be breaking the process into roughly five steps covered in subsections.

17.5.1 Creating the database model

Our first step is to focus on creating everything related to the database table where we will store the password reset tokens.

While we could store these as a column in the `users` table we created earlier, this could become problematic long-term. For starters, we will have extra columns added to every single user in our database that frequently aren't being

used. It will also make validations, such as ensuring that all reset tokens are unique, harder to implement as we will need to ensure that even when a user isn't trying to reset their password that they still have a unique token.

Instead, we are going to store the password reset token in a new database table. We will be naming this table **pw_resets**, and it will store both the UserID that the reset token maps to, as well as a hashed value of the token. We are storing a hashed value of the token for reasons similar to why we stored a hashed value of the remember token - if our database is breached we don't want attackers to be able to recreate the original tokens, but we still need a way to validate that a token provided is one we created.

We will also have the normal columns from a **gorm.Model**, so we will know when the database entry was last created as well as its ID.

Putting that all together we can start to build our database model, which will go in a new source file.

```
$ atom models/pw_resets.go
```

We will start by declaring the package and adding the **pwReset** model. Notice that this model will NOT be exported, as we won't be using it directly outside of our models package. Instead, we are going to eventually expose the password reset functionality by adding methods to our UserService that mask all of the implementation details.

```
package models

import "github.com/jinzhu/gorm"

type pwReset struct {
    gorm.Model
    UserID   uint   `gorm:"not null"`
    Token    string `gorm:"-"`
    TokenHash string `gorm:"not null;unique_index"`
}
```

The Token and TokenHash fields here are going to be used in the same way we use the Remember and RememberHash fields on the User type. The former is used to store the raw value that is never stored in our database, while the latter is used to store the hashed value which will be persisted in our database.

We are also going to create an interface defining what functionality is available for our password reset table.

```
type pwResetDB interface {
    ByToken(token string) (*pwReset, error)
    Create(pwr *pwReset) error
    Delete(id uint) error
}
```

ByToken will be used to lookup a pwReset by its token. Create will be used to create a new password reset, and delete will delete them via their ID.

Next up is the GORM implementation of this interface.

```
type pwResetGorm struct {
    db *gorm.DB
}

func (pwrg *pwResetGorm) ByToken(tokenHash string) (*pwReset, error) {
    var pwr pwReset
    err := first(pwrg.db.Where("token_hash = ?", tokenHash), &pwr)
    if err != nil {
        return nil, err
    }
    return &pwr, nil
}

func (pwrg *pwResetGorm) Create(pwr *pwReset) error {
    return pwrg.db.Create(pwr).Error
}

func (pwrg *pwResetGorm) Delete(id uint) error {
    pwr := pwReset{Model: gorm.Model{ID: id}}
    return pwrg.db.Delete(&pwr).Error
}
```

Once again, this should look familiar to code we have written in the past, so we won't cover what it is doing in detail.

Next up is the validation layer. We will start with the pwResetValidator type.

```
import (
    "github.com/jinzhu/gorm"
    "lenslocked.com/hash"
    "lenslocked.com/rand"
)

func newPwResetValidator(db pwResetDB, hmac hash.HMAC) *pwResetValidator {
    return &pwResetValidator{
        pwResetDB: db,
        hmac:      hmac,
    }
}

type pwResetValidator struct {
    pwResetDB
    hmac hash.HMAC
}
```

We went ahead and added the HMAC here, because we know we will be hashing our tokens before storing them in the database.

Next up are the validation functions we will need.

```
func (pwrw *pwResetValidator) requireUserID(pwr *pwReset) error {
    if pwr.UserID <= 0 {
        return ErrUserIDRequired
    }
    return nil
}

func (pwrw *pwResetValidator) setTokenIfUnset(pwr *pwReset) error {
    if pwr.Token != "" {
        return nil
    }
    token, err := rand.RememberToken()
    if err != nil {
        return err
    }
    pwr.Token = token
    return nil
}
```

```

}

func (pwrv *pwResetValidator) hmacToken(pwr *pwReset) error {
    if pwr.Token == "" {
        return nil
    }
    pwr.TokenHash = pwrv.hmac.Hash(pwr.Token)
    return nil
}

```

Again, we are writing code similar to code we have written in the past. In fact, every validation function here is just a variation of a validation function we created on either the gallery or user models.

Just like with our other models, we also will be using some code to make running these validations easier.

```

type pwResetValFn func(*pwReset) error

func runPwResetValFns(pwr *pwReset, fns ...pwResetValFn) error {
    for _, fn := range fns {
        if err := fn(pwr); err != nil {
            return err
        }
    }
    return nil
}

```

And finally we are ready to create the methods defined by the pwResetDB interface and add our validations to them.

```

func (pwrv *pwResetValidator) ByToken(token string) (*pwReset, error) {
    pwr := pwReset{Token: token}
    err := runPwResetValFns(&pwr, pwrv.hmacToken)
    if err != nil {
        return nil, err
    }
    return pwrv.pwResetDB.ByToken(pwr.TokenHash)
}

func (pwrv *pwResetValidator) Create(pwr *pwReset) error {
}

```

```
err := runPwResetValFns(pwr,
    pwrv.requireUserID,
    pwrv.setTokenIfUnset,
    pwrv.hmacToken,
)
if err != nil {
    return err
}
return pwrv.pwResetDB.Create(pwr)
```



```
func (pwrv *pwResetValidator) Delete(id uint) error {
    if id <= 0 {
        return ErrIDInvalid
    }
    return pwrv.pwResetDB.Delete(id)
}
```

That completes the code for the **pw_resets.go** source file. In the next section we will look at how we will be using this code inside our UserService to expose the functionality to initiate and complete a password reset.

Source code

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

Completed - book.usegolang.com/17.5.1

Changes - book.usegolang.com/17.5.1-diff

17.5.2 Updating the services

In this section we will focus on updating our model services to utilize the pwReset type we created in order to implement functions that allow us to initiate and complete the password reset process.

Rather than exposing the pwReset model and the pwResetDB directly, we are instead going to add the InitiateReset and CompleteReset methods to the UserService which will handle everything related to resetting a user's password. This includes reading and writing to the pwReset database table.

Let's start by adding the methods we ultimately want to implement to our UserService interface.

```
$ atom models/users.go
```

```
type UserService interface {
    Authenticate(email, password string) (*User, error)
    // InitiateReset will complete all the model-related tasks
    // to start the password reset process for the user with
    // the provided email address. Once completed, it will
    // return the token, or an error if there was one.
    InitiateReset(email string) (string, error)
    // CompleteReset will complete all the model-related tasks
    // to complete the password reset process for the user that
    // the token matches, including updating that user's pw.
    // If the token has expired, or if it is invalid for any
    // other reason the ErrTokenInvalid error will be returned.
    CompleteReset(token, newPw string) (*User, error)
    UserDB
}
```

We are adding the methods to the user service because ultimately they relate to the User resource. Users of our models package don't care about the details of how a password is reset, but instead only care that they can initiate and complete the process.

The first thing we need to do to implement these two functions is to add a pwResetDB to our **userService** type.

```
func NewUserService(db *gorm.DB, pepper, hmacKey string) UserService {
    ug := &userGorm{db}
    hmac := hash.NewHMAC(hmacKey)
    uv := newUserValidator(ug, hmac, pepper)
    return &userService{
```

```

    UserDB:    uv,
    pepper:    pepper,
    pwResetDB: newPwResetValidator(&pwResetGorm{db}, hmac),
}
}

type userService struct {
UserDB
pepper    string
pwResetDB pwResetDB
}

```

With that we can start implementing the initiate and complete methods. We will start with the `InitiateReset` method, which is intended to lookup a user by their email address, create a `pwReset`, and then return its corresponding token.

```

func (us *userService) InitiateReset(email string) (string, error) {
user, err := us.ByEmail(email)
if err != nil {
    return "", err
}
pwr := pwReset{
    UserID: user.ID,
}
if err := us.pwResetDB.Create(&pwr); err != nil {
    return "", err
}
return pwr.Token, nil
}

```

Next is the `CompleteReset` method, which has a little more logic involved because we need to verify that the `pwReset` hasn't expired. We will stub that part out for now and just write rest of the `CompleteReset` method first.

```

const (
// Add this error to our errors
ErrTokenInvalid modelError = "models: token provided is not valid"
)

func (us *userService) CompleteReset(token,
newPw string) (*User, error) {

```

```
pwr, err := us.pwResetDB.ByToken(token)
if err != nil {
    if err == ErrNotFound {
        return nil, ErrTokenInvalid
    }
    return nil, err
}
// TODO: If the pwReset is over 12 hours old, it is
// invalid and we should return the ErrTokenInvalid error

user, err := us.ByID(pwr.UserID)
if err != nil {
    return nil, err
}
user.Password = newPw
err = us.Update(user)
if err != nil {
    return nil, err
}
us.pwResetDB.Delete(pwr.ID)
return user, nil
}
```

Aside from the code checking if the token is too old, the rest of this code should look relatively familiar.

We first lookup the token and if one can't be found, we replace the ErrNotFound error with an ErrTokenInvalid error. We do this to ensure the end user sees a useful error message that tells them their token isn't valid.

After we eventually verify the token hasn't expired, we then lookup a user with the UserID field from our pwReset. Assuming there aren't any errors there, we update that user's password, delete the pwReset, and then return the user.

We wait until after the password has been updated to delete the reset token in case our server has an issue, permitting the end user to try again with the same token. Even though it is unlikely to happen often, it would be a pretty bad user experience to make the user request a new password reset token because our database had an issue, and it is easy to prevent by delaying the deletion.

Now we just need to figure out how to verify that the token hasn't expired. There are basically two ways to do this:

1. Change the ByToken method to only retrieve tokens that haven't expired.
2. Use the **time** package to verify the pwReset was created within the last 12 hours.

Both approaches have their own pros and cons. We are going to use the second approach so we can get a little practice using the **time** package². Specifically, we are going to use the **time.Now** function³ and the **Sub** method⁴ provided the **time.Time** type⁵.

In order for this all to make sense, the first thing we need to understand are **Durations**⁶, which are also part of the **time** package and are used to represent the elapsed time between two instants. That is, Durations are used to represent things like “1 hour” or “4 minutes 3 seconds” in code.

In order to verify if our pwReset is valid, we are going to calculate the duration between the current time - **time.Now()** - and when the pwReset was created. To do this use the **Sub** method, which stands for “subtract”.

```
duration := time.Now().Sub(pwr.CreatedAt)
```

In order to create a constant duration, like “12 hours”, we would multiply the number **12** by the duration “one hour”, which is provided by the time package as the constant **time.Hour**.

```
twelveHrs := 12 * time.Hour
```

Finally, we can compare these two durations using less than, greater than, and equals signs much like we would any other integer.

²<https://golang.org/pkg/time/>

³<https://golang.org/pkg/time/#Now>

⁴<https://golang.org/pkg/time/#Time.Sub>

⁵<https://golang.org/pkg/time/#Time>

⁶<https://golang.org/pkg/time/#Duration>

```
if duration > twelveHrs {
    // ... our duration is over 12 hours, meaning our pwReset
    // was created over 12 hours ago and is invalid.
}
```

Putting this all together, we can update our CompleteReset function with the source code below.

```
import (
    // Add this import
    "time"
)

func (us *userService) CompleteReset(token,
    newPw string) (*User, error) {
    pwr, err := us.pwResetDB.ByToken(token)
    if err != nil {
        if err == ErrNotFound {
            return nil, ErrTokenInvalid
        }
        return nil, err
    }
    if time.Now().Sub(pwr.CreatedAt) > (12 * time.Hour) {
        return nil, ErrTokenInvalid
    }
    user, err := us.ByID(pwr.UserID)
    if err != nil {
        return nil, err
    }
    user.Password = newPw
    err = us.Update(user)
    if err != nil {
        return nil, err
    }
    us.pwResetDB.Delete(pwr.ID)
    return user, nil
}
```

Finally, we need to update the **services.go** source file's DestructiveReset and AutoMigrate methods so that our pwReset table gets constructed when we call them.

```
$ atom models/services.go
```

```
func (s *Services) DestructiveReset() error {
    // Add the &pwReset{} below
    err := s.db.DropTableIfExists(&User{}, &Gallery{},
        &pwReset{}).Error
    // ...
}

func (s *Services) AutoMigrate() error {
    // Add the &pwReset{} below
    return s.db.AutoMigrate(&User{}, &Gallery{},
        &pwReset{}).Error
}
```

Now when we restart our application and it calls AutoMigrate this code will handle setting up the table in our SQL database.

Source code

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

Completed - book.usegolang.com/17.5.2

Changes - book.usegolang.com/17.5.2-diff

17.5.3 Forgotten password forms

The next thing we are going to do is create the forms used for our workflow. We will need two of these; one used for requesting a password reset token with an email address, and another for submitting that token along with a new password.

In this section we won't be creating the views that use these templates, but we will get the templates ready for when we do that in the next section.

We will start with the form for requesting a password reset token, and we will call this the `forgot_pw` template, which will be stored in the directory with all our other user views.

```
$ atom views/users/forgot_pw.gohtml
```

Our forgot password form will be very similar to the sign up form, except it will POST to another path and will only have an input for a user's email address.

```
 {{define "yield"}}
<div class="row">
  <div class="col-md-8 col-md-offset-2">
    <div class="panel panel-primary">
      <div class="panel-heading">
        <h3 class="panel-title">Forgot Your Password?</h3>
      </div>
      <div class="panel-body">
        {{template "forgotPwForm" .}}
      </div>
      <div class="panel-footer">
        <a href="/login">Remember your password?</a>
      </div>
    </div>
  </div>
{{end}}
```



```
 {{define "forgotPwForm"}}
<form action="/forgot" method="POST">
  {{csrfField}}
  <div class="form-group">
    <label for="email">Email address</label>
    <input type="email" name="email" class="form-control" id="email" placeholder="Email" value="{{.Email}}">
  </div>
  <button type="submit" class="btn btn-primary">Submit</button>
</form>
{{end}}
```

While creating this form we went ahead and added a link to the login page with the text, “Remember your password?” This is handy for users who accidentally navigate to this page or who suddenly remember their password.

We will be adding links like this to all of our authentication forms, and all will be placed inside a **panel-footer**, but the text and where they link will change based on what makes the most sense for that particular form. For instance, we will add a link to the forgot password form from the login page.

```
$ atom views/users/login.gohtml
```

Add the following footer below the **panel-body** section.

```
<div class="panel-footer">
  <a href="/forgot">Forgot your password?</a>
</div>
```

When a user is logging in it is likely that is when they will realize they forgot their password, so it is the most logical place to put a link to the forgot password page.

Similarly, when a user is creating an account they might realize they already have an account and need a link to the log in page.

```
$ atom views/users/new.gohtml
```

Once again, the HTML below will go after the **panel-body** section.

```
<div class="panel-footer">
  <a href="/login">Already have an account?</a>
</div>
```

Finally, we need to create the form for when a user is resetting their password and already has a reset token.

```
$ atom views/users/reset_pw.gohtml
```

```
{{define "yield"}}
<div class="row">
  <div class="col-md-8 col-md-offset-2">
    <div class="panel panel-primary">
      <div class="panel-heading">
        <h3 class="panel-title">Reset Your Password</h3>
      </div>
      <div class="panel-body">
        {{template "resetPwForm" .}}
      </div>
      <div class="panel-footer">
        <a href="/forgot">Need to request a new token?</a>
      </div>
    </div>
  </div>
{{end}}
```

```
{{define "resetPwForm"}}
<form action="/reset" method="POST">
  {{csrfField}}
  <div class="form-group">
    <label for="token">Reset Token</label>
    <input type="text" name="token" class="form-control"
      id="token" placeholder="You will receive this via email"
      value="{{.Token}}">
  </div>
  <div class="form-group">
    <label for="password">Password</label>
    <input type="password" name="password" class="form-control"
      id="password" placeholder="Password">
  </div>
  <button type="submit" class="btn btn-primary">Submit</button>
</form>
{{end}}
```

That's it for our view updates. We will discuss how to use the new views in the next section.

*Note: A few of our forms - like the sign up and log in forms - have different widths specified by the **col-md-8** css class before the panel. If you want, you can make these all the same by updating that portion of the HTML.*

Source code

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

Completed - book.usegolang.com/17.5.3

Changes - book.usegolang.com/17.5.3-diff

17.5.4 Controller actions and views

Next we need to create http handlers (aka controller actions in MVC) and views that will be used to render and process the reset password forms. We will start by declaring our views.

```
$ atom controllers/users.go
```

```
func NewUsers(us models.UserService, emailer *email.Client) *Users {
    return &Users{
        NewView:      views.NewView("bootstrap", "users/new"),
        LoginView:    views.NewView("bootstrap", "users/login"),
        ForgotPwView: views.NewView("bootstrap", "users/forgot_pw"),
        ResetPwView:  views.NewView("bootstrap", "users/reset_pw"),
        us:           us,
        emailer:     emailer,
    }
}

type Users struct {
    NewView      *views.View
    LoginView    *views.View
    ForgotPwView *views.View
    ResetPwView  *views.View
    us          models.UserService
    emailer     *email.Client
}
```

Now we need to add methods to our Users type for processing these forms,

and we will also need one for rendering the ResetPwView with prefilled data provided via URL query parameters.

Let's start with the code used to process the forgot password form. We will need to define the struct type used to represent the form data first. We are going to go ahead and share this type between our two forms to reduce some code.

```
// ResetPwForm is used to process the forgot password form
// and the reset password form.
type ResetPwForm struct {
    Email    string `schema:"email"`
    Token    string `schema:"token"`
    Password string `schema:"password"`
}
```

*Note: We add the **schema** tags so the gorilla/schema package can decode an http form and place values into a struct of this type.*

Next up is the function InitiateReset, which will be used to process the forgot password form. This form is only going to have a single field - the user's email address - so we can ignore the other fields in the ResetPwForm type.

```
// POST /forgot
func (u *Users) InitiateReset(w http.ResponseWriter, r *http.Request) {
    var vd views.Data
    var form ResetPwForm
    vd.Yield = &form
    if err := parseForm(r, &form); err != nil {
        vd.SetAlert(err)
        u.ForgotPwView.Render(w, r, vd)
        return
    }

    token, err := u.us.InitiateReset(form.Email)
    if err != nil {
        vd.SetAlert(err)
        u.ForgotPwView.Render(w, r, vd)
        return
    }

    // TODO: Email the user their password reset token. In the
    // meantime we need to "use" the token variable
```

```
_ = token

views.RedirectAlert(w, r, "/reset", http.StatusFound, views.Alert{
    Level:  views.AlertLvlSuccess,
    Message: "Instructions for resetting your password have been emailed to you.",
})
}
```

Our function will eventually need to send the user an email with their password reset token, but for now we have stubbed that part out with a TODO comment. We will handle that in the next section. In the meantime we need to add the `_ =` portion to trick the compiler into thinking we use the token variable, since we won't use it until we send the user an email.

Note: Using the `_ =` trick isn't a great idea all the time, as you can end up with many unused variables in your code. I am using it in this case because I know we will be using this variable in the next section of this chapter.

Aside from emailing the user, the rest of our code is creating a `views.Data` and `form` variable, parsing the form via the `parseForm` function, and using methods provided by our `UserService` to initiate the password reset.

The last few lines - the `RedirectAlert` function call - might be a little new because we just created that function in this chapter, but all we are doing is redirecting the user while also rendering an alert on the next page they visit.

Next we need to handle rendering the reset password form and prefilling it with any URL query parameters. This will enable us to send the user to the password reset form with their reset token already filled in for them.

To do this we are going to create a `ResetPw` handler that parses the URL query parameters and then renders the reset password form with that data. If no parameters are provided the fields will end up being empty, but otherwise they will have the data filled in automatically.

```
// ResetPw displays the reset password form and has a method
// so that we can prefill the form data with a token provided
// via the URL query params.
//
// GET /reset
func (u *Users) ResetPw(w http.ResponseWriter, r *http.Request) {
    var vd views.Data
    var form ResetPwForm
    vd.Yield = &form
    if err := parseURLParams(r, &form); err != nil {
        vd.SetAlert(err)
    }
    u.ResetPwView.Render(w, r, vd)
}
```

The final function we are going to add to our users controller will be used to process the reset password form.

```
// CompleteReset processed the reset password form
//
// POST /reset
func (u *Users) CompleteReset(w http.ResponseWriter, r *http.Request) {
    var vd views.Data
    var form ResetPwForm
    vd.Yield = &form
    if err := parseForm(r, &form); err != nil {
        vd.SetAlert(err)
        u.ResetPwView.Render(w, r, vd)
        return
    }

    user, err := u.us.CompleteReset(form.Token, form.Password)
    if err != nil {
        vd.SetAlert(err)
        u.ResetPwView.Render(w, r, vd)
        return
    }

    u.signIn(w, user)
    views.RedirectAlert(w, r, "/galleries", http.StatusFound, views.Alert{
        Level:  views.AlertLvlSuccess,
        Message: "Your password has been reset and you have been logged in!",
    })
}
```

In CompleteReset we first parse the form to get the token and new password.

After we have that data we can call the CompleteReset method provided by the user service to allow it to do any model-related work that needs done. Once completed, the user service's CompleteReset method will return either a user with an updated password, or an error. In the case of an error we just render the reset password form again, but otherwise we know the user's password has been updated and we are going to sign them in rather than forcing them to type the password they just changed.

Lastly, we want to redirect the user to their galleries page with an alert message notifying them that they have successfully updated their password.

With our users controller updated, we now need to create routes for everything we just added.

```
$ atom main.go
```

Add the following routes to your main function.

```
r.Handle("/forgot", usersC.ForgotPwView).Methods("GET")
r.HandleFunc("/forgot", usersC.InitiateReset).Methods("POST")
r.HandleFunc("/reset", usersC.ResetPw).Methods("GET")
r.HandleFunc("/reset", usersC.CompleteReset).Methods("POST")
```

Note: The ForgotPwView uses the Handle method instead of the HandleFunc method because it is an http.Handler interface, not an http.HandlerFunc like the other three.

Source code

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

Completed - book.usegolang.com/17.5.4

Changes - book.usegolang.com/17.5.4-diff

17.5.5 Emailing users and building URLs

We have nearly completed the password reset feature; the final step is to add the code used to email users with their password reset token. By forcing users to read the email in order to proceed we are verifying that they have access to that account's email inbox.

Let's start by adding a few constants to our email package that will be used to build our password reset email.

```
$ atom email/mailgun.go
```

First the email subject.

```
const (
    resetSubject = "Instructions for resetting your password."
)
```

And then the template used to build the text version of the email. In our case we are going to be using the `fmt.Sprintf` function⁷ to fill in any dynamic data, but you could also opt to use a package like `text/template` or `html/template` depending on the situation.

```
const resetTextTmpl = `Hi there!

It appears that you have requested a password reset. If this was you, please follow the link below.

%s`
```

⁷<https://golang.org/pkg/fmt/#Sprintf>

```
If you are asked for a token, please use the following value:  
%s  
  
If you didn't request a password reset you can safely ignore this email and your account will n  
  
Best,  
LensLocked Support  
`
```

Now when we want to create an email we can replace the two `%s` characters with a reset URL and the password reset token. This isn't as safe as using a template since our variables aren't named and we have to know a lot more about the template to create a valid email, but given the simplicity of this template it shouldn't be an issue. Just remember to consider using a template package or something else if your email templates are getting more complicated.

Finally, we need a template for our HTML version of the email.

```
const resetHTMLTpl = `Hi there!  
  
It appears that you have requested a password reset. If this was you, please follow the link be  
  
<a href="%s">%s</a>  
  
If you are asked for a token, please use the following value:  
  
%s  
  
If you didn't request a password reset you can safely ignore this email and your account will n  
  
Best,  
LensLocked Support  
`
```

The HTML version is nearly identical to the text one, but we have placed line breaks everywhere we want them and have used an `<a ...>` tag for our link which takes two string variables, so we will need to pass in three variables when we build this email part.

Before we can write the code that uses these templates we need to discuss how to build URLs with encoded query parameters. While we could just send the user an email with a password reset token and a link to the reset page, many users have come to expect a little more. Most password reset emails will now include a link that has the token embedded into it via URL query parameters. Then when a user clicks it, the form will auto-populate the reset token field so the user never has to fill it out.

In order to do that with Go we are going to be using the `net/url` package⁸. Specifically, we will be using the `Values` type⁹ which was designed specifically for query parameters.

The simplest way to build the encoded query parameters is to construct a `url.Values` object then combine the output of its `Encode` method¹⁰ with the base URL you want to use.

```
v := url.Values{}
v.Set("token", "encode-me-!#@(*&")
resetUrl := "https://lenslocked.com/reset?" + v.Encode()
// resetUrl is ...
// https://lenslocked.com/reset?token=encode-me-%21%23%40%28%2A%26
```

With that knowledge, we can use the `url.Values` type to construct our reset password URL while adding a `ResetPw` method to our email client.

```
const resetBaseURL = "https://www.lenslocked.com/reset"

func (c *Client) ResetPw(toEmail, token string) error {
    v := url.Values{}
    v.Set("token", token)
    resetUrl := resetBaseURL + "?" + v.Encode()
    resetText := fmt.Sprintf(resetTextTmpl, resetUrl, token)
    message := mailgun.NewMessage(c.from, resetSubject, resetText, toEmail)
    resetHTML := fmt.Sprintf(resetHTMLTmpl, resetUrl, token)
    message.SetHtml(resetHTML)
```

⁸<https://golang.org/pkg/net/url/>

⁹<https://golang.org/pkg/net/url/#Values>

¹⁰<https://golang.org/pkg/net/url/#Values.Encode>

```
_ , _, err := c.mg.Send(message)
return err
}
```

With that we are ready to send users emails with their password reset token after they submit the forgot password form.

```
$ atom controllers/users.go
```

Find the “TODO” inside of the `InitiateReset` method. We added a `_ = token` line near it as well. Replace the TODO with the following code, which will use the `ResetPw` method we just created to email the user their reset token.

```
func (u *Users) InitiateReset(w http.ResponseWriter, r *http.Request) {
    // ... unchanged code

    err = u.emailer.ResetPw(form.Email, token)
    if err != nil {
        vd.SetAlert(err)
        u.ForgotPwView.Render(w, r, vd)
        return
    }

    // ... unchanged code
}
```

And that’s it. We have completed the password reset functionality, which was the last feature we will be adding in this book.

Source code

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

Completed - book.usegolang.com/17.5.5

Changes - book.usegolang.com/17.5.5-diff

17.6 Where do I go next?

First, I just want to say congratulations! What we built in this course wasn't small by any means, and you should be proud of yourself for completing the book and your application!

While we won't be covering anymore features in this book, that doesn't mean you need to quit. A great way to continue learning is to continue adding new features to your web application, build new apps, or look for open source projects you can contribute to. The important thing is to keep practicing and writing code. That is the *only* way to get better.

If you are looking for advice on what to build next, the most common advice I give is to try to build another web application. The key here is to pick something that isn't too challenging, but is within your skillset. A few examples of applications that should fit that bill are:

1. A simplified Twitter clone.
2. A simplified Pinterest clone.
3. A task manager similar to Asana or Trello, but without the fancy JS.

The key here is to keep it simple. Figure out the bare minimum features you need to create your application and start there, then add on once you complete that part.

While building your application, try to see how much you can complete without referencing this book or other tutorials. Documentation for the standard library and other packages is fine - we all need to reference those from time to time

- but try to avoid anything that tells you what code to write and make those decisions on your own.

If you do get stuck, try to use this book as a guide. For instance, if you are building a twitter clone you might follow along with the chapters where we build the gallery resource and instead build your own “tweet” resource. The book will still give you a feeling for what code you need to complete, but you will need to write that code on your own. After building a few applications this way you will soon find that you don’t need to reference the book very often when writing code, but to get there it takes practice.

Happy coding!

Source code

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

Completed - book.usegolang.com/final

Chapter 18

Appendix

The appendix contains some write-ups about subjects that may be useful when reading the book, but are slightly outside the scope of the book. They are included in the appendix so that when the book is printed the size can be reduced a bit, while still including the material in a digital format for those who would like to reference it.

18.1 Using interfaces in Go

The **UserService** might not make sense just yet, but before proceeding, we are going to spend a few minutes talking about interfaces and how you interact with them in Go.

If you are already familiar with interfaces feel free to skip this section. We aren't going to keep any of the code from this section, and it is intended entirely for educational purposes.

Interfaces in Go are a way of accepting arguments to a function while not being too picky about what the actual data is. For example, let's imagine we wanted to write a function that looks at two objects and returns true if the first

one is “denser” than the second one. We’ll call it `IsItDenser()`.

Now let’s imagine that we are writing this function for a geological application, and we have a `Rock` type that we want to compare.

```
type Rock struct {
    Mass  int
    Volume int
}

func (r Rock) Density() int {
    return r.Mass / r.Volume
}
```

Knowing that we have this rock type, we would probably write our `IsItDenser()` function to take advantage of it and to use the `Density()` method.

```
func IsItDenser(a, b Rock) bool {
    return a.Density() > b.Density()
}
```

At that point we would continue along with our code without any issues, until suddenly we find out that someone introduced a `Geode` type to our application, and we now need to be able to compare both geodes and regular rocks.

```
type Geode struct {
    Rock
}
```

At this point our code might start complaining when we try to pass a `Geode` into the `IsItDenser()` function, because a `Geode` isn’t a `Rock`. It might have a `Rock` object embedded in it, but it isn’t a `Rock`. So how do we fix our code?

This is where interfaces shine; Interfaces provide us with a way of writing our `IsItDenser()` function so that it doesn’t care what type of object you pass in, as long as it implements the methods required by the interface type.

To make this work in our rock and geode example, we would first start off by creating an interface type.

```
type Dense interface {
    Density() int
}
```

Both the **Geode** and the **Rock** type have a **Density()** method, so they are both implementations of the **Dense** interface by default. No extra code is necessary to implement an interface in Go.

After creating our interface we would go and update our **IsItDenser()** function to accept parameters of the **Dense** type instead of the **Rock** type.

With those changes we would then be able to compare both rocks and geodes in our **IsItDenser()** function without having to worry about the type of each. The only thing our **IsItDenser()** function would care about is that the arguments passed in both implement the **Dense** interface.

How that interface gets implemented is irrelevant; In this example the **Density()** method is actually delegated to the **Rock** object that is nested inside of the **Geode** object, but it doesn't have to be. We could just as easily write a **Density()** method on the **Geode** type and our code would still work.

```
package main

import "fmt"

type Rock struct {
    Mass  int
    Volume int
}

func (r Rock) Density() int {
    return r.Mass / r.Volume
}

func IsItDenser(a, b Dense) bool {
    return a.Density() > b.Density()
}
```

```
type Geode struct {
    Rock
}

func (g Geode) Density() int {
    return 100
}

type Dense interface {
    Density() int
}

func main() {
    r := Rock{10, 1}
    g := Geode{Rock{2, 1}}
    // Returns true because Geode's Density method always
    // returns 100
    fmt.Println(IsItDenser(g, r))
}
```

By creating a **UserService** this enables us to do something similar in our code. All of our controllers will only care about calling the methods defined by the **UserService** interface, which means that at any point in time we could write a new implementation of the **UserService** that uses another database, or maybe even one that doesn't use any database at all for testing without setting up a full database.

18.2 Refactoring with gorename

A common way to perform a refactor in Go is to use the **gorename** tool. If you are using an IDE you might be able to use a built-in refactor tool, but since many of us aren't I find it helpful to get a quick overview of how gorename works.

First we need to grab the tool.

```
$ go get golang.org/x/tools/cmd/gorename
```

gorename needs two pieces of information to work - the variable, type, or function that you will be renaming, along with the new name you want to use. With that information it will parse all of your Go code attempting to update comments and references to that code with the changed name. This can be really handy when refactoring some code that is used by many different packages you have coded in the past.

To give a real example of this, we are going to rename the ErrInvalidID and ErrInvalidPassword variables that get refactored in [Section 12.7](#).

The first variable we will look at is ErrInvalidID. We want to rename this to ErrIDInvalid, so we would run the following in our console.

```
$ gorename -from '"lenslocked.com/models".ErrInvalidID' -to 'ErrIDInvalid'
```

When doing this you typically need to define the entire package in the from portion, but this isn't necessary in the to section. It is also typically a good idea to wrap the package in double quotes, and to wrap the entire string in a single quote just to ensure there aren't any unexpected issues.

Box 18.1. **gorename** errors

As we discussed earlier, gorename processes *all* of the code in your GOPATH, so if there is an issue with any of this code it is possible you will see an error when using the rename tool. The error message might be long, but it will typically end with a message like:

```
gorename: couldn't load packages due to errors: some/package/name
```

If you see this, I suggest just moving that broken code to another directory temporarily so you can proceed. Hopefully gorenname will be updated to work in this case in the future.

Assuming you haven't already refactored, running the command should yield an output similar the one below.

```
Renamed 2 occurrences in 1 file in 1 package.
```

This response means that gorenname updated the code in two places in a single source file. Our comment right above the variable should also get updated, as gorenname tries to update comments in these obvious locations.

Next we need to do the same thing for our ErrInvalidPassword error.

```
$ gorenname -from '"lenslocked.com/models".ErrInvalidPassword' \
-to 'ErrPasswordIncorrect'
```

Note: The backslash (\) is there because this command runs over to a second line, and the backslash tells our terminal we have more input to provide.

After running this you should see the following output:

```
Renamed 3 occurrences in 2 files in 2 packages.
```

And that is how you refactor using gorenname!

18.3 Handling Bootstrap issues

Bootstrap updated to the version 4 beta in the process of this course being created, so if you head over to the Bootstrap docs it is very likely that you might accidentally be looking at docs for v4 beta and not for the version used in this course - v3.3.7. As a result, my first suggestion is to check out the docs for version 3.3 first - <https://getbootstrap.com/docs/3.3/>

While I do eventually intend to update this course to use version 4, I am holding off on doing so right away because it is in your best interest to learn version 3 right now. Not only is version 4 in beta, but most of the themes and other resources you will find online at this time are intended for version 3. For example, bootswatch.com, a site that offers free themes for Bootstrap, is tuned specifically for version 3, as are most themes sold on websites like ThemeForest.

Once version 4 has started to pick up proper traction I will provide updates to the course, but in the meantime rest assured that version 3 is what you really ought to be learning, and v4's changes will be easy to learn once you understand v3.

18.4 Private model errors

TODO: Write about the code below. For now it is provided as an example.

Step 1 - Create an errors source file and create the privateError type.

```
$ atom models/errors.go
```

```
package models

type privateError string

func (e privateError) Error() string {
```

```

    return string(e)
}

```

Step 2 - Move our existing modeError type over to the errors source file.

```

import "strings"

type modelError string

func (e modelError) Error() string {
    return string(e)
}

func (e modelError) Public() string {
    s := strings.Replace(string(e), "models: ", "", 1)
    split := strings.Split(s, " ")
    split[0] = strings.Title(split[0])
    return strings.Join(split, " ")
}

```

Step 3 - Move our existing errors over to the errors source file and convert the ones that shouldn't be public into private errors.

```

const (
    // ErrNotFound is returned when a resource cannot be found
    // in the database.
    ErrNotFound modelError = "models: resource not found"
    // ErrPasswordIncorrect is returned when an invalid password
    // is used when attempting to authenticate a user.
    ErrPasswordIncorrect modelError = "models: incorrect password provided"
    // ErrEmailRequired is returned when an email address is
    // not provided when creating a user
    ErrEmailRequired modelError = "models: email address is required"
    // ErrEmailInvalid is returned when an email address provided
    // does not match any of our requirements
    ErrEmailInvalid modelError = "models: email address is not valid"
    // ErrEmailTaken is returned when an update or create is attempted
    // with an email address that is already in use.
    ErrEmailTaken modelError = "models: email address is already taken"
    // ErrPasswordRequired is returned when a create is attempted
    // without a user password provided.
    ErrPasswordRequired modelError = "models: password is required"
    // ErrPasswordTooShort is returned when an update or create is
)

```

```
// attempted with a user password that is less than 8 characters.
ErrPasswordTooShort modelError = "models: password must be at least 8 characters long"
ErrTitleRequired    modelError = "models: title is required"

// ErrIDInvalid is returned when an invalid ID is provided
// to a method like Delete.
ErrIDInvalid privateError = "models: ID provided was invalid"
// ErrRememberRequired is returned when a create or update
// is attempted without a user remember token hash
ErrRememberRequired privateError = "models: remember token is required"
// ErrRememberTooShort is returned when a remember token is
// not at least 32 bytes
ErrRememberTooShort privateError = "models: remember token must be at least 32 bytes"
ErrUserIDRequired   privateError = "models: user ID is required"
)
```