Learn to Create Real World Web Applications using Go

by: Jonathan Calhoun

Web Development with Go

Learn to Create Real World Web Applications using Go

Jonathan Calhoun

Contents

About the author

Jon Calhoun is the author of the Web Development with Go, the book you are currently reading, and is also the founder of EasyPost (easypost.com), an API that helps companies integrate with shipping APIs. He is a former Google employee, Y Combinator alumn, and has a B.S. in Computer Science from the University of Central Florida.

Copyright and license

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.

Jordan does both website and graphic design and is available for hire. If you would like his contact information, please reach out to me - jon@calhoun.io - and I will make an introduction.

Where can I buy the full book?

Full copies of Web Development with Go can be purchased at usegolang.com.

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 start your career as a web developer, or to start building your online businesses.

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." This 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.

If you don't know how to program or how to use Go, don't worry! This book assumes almost no prior programming or Go experience, and I explain everything covered in this book in great detail as we go along. This level of detail has lead to feedback like:

I went through the sample chapters again (excited for the book!)

> last night and realized how approachable and PRODUCTIVE it was!

That said, there is a lot of material covered in this book. Not only will we be covering Go in great detail, but we will also be covering a basic amount of HTML, CSS, JavaScript, SQL, and we will even be using the command line. We will also be using Bootstrap, an HTML, CSS, and JS framework, which basically just means that someone else wrote a bunch of HTML, CSS, and JS that we can reuse to quickly make a pretty website.

Needless to say, that is a lot to take in on a first pass, so if you ever find yourself confused or overwhelmed, **don't stress yourself out!** It is perfectly normal, and it doesn't mean you are doomed and will never be a great web developer. Instead it might mean that you should take a step back and try to understand each piece of the puzzle on its own a bit before proceeding. I have provided two sections in this book designed to help anyone who feels overwhelmed or just wants to get a little more background information.

The first is Section 1.5, which lists several resources for learning about programming and Go. If you have never programmed before, or if you have programmed in another language and want an introduction to Go, that is the section to check out.

The second is Section 1.6, which references several books by Michael Hartl, and these all focus on teaching you enough knowledge of subjects like HTML, CSS, and JS to get by. You will not learn enough to be an expert at any of these tools, but you **do not need to be an expert** to get a ton of value out of this book. In fact, I am far from an expert in some of these topics myself, but instead learn more as the need arises.

If you are ever feeling uneasy, confused, or simply uncertain throughout this book, remember that you can always send me an email and I'll be happy to help out - jon@calhoun.io.

1.2 What if I am already familiar with web development / programming / Go?

If you are already familiar with web development, this book is still a great fit for you!

While I make every effort to go over the details of what I am doing for beginners, I also clearly label these sections so that someone familiar with the topic can either skim or skip that section and use it as a reference if they get confused down the road.

1.3 What are we building?

Web Development with Go takes an integrated approach to teaching web development. You won't be reading about theoretical web applications, but instead you will be following along and building a real web application that you can deploy and share with your friends.

The application we will be building is very similar to the popular photo gallery application, Pixieset. In Pixieset, users (who are typically photographers) can sign up for an account and create galleries. Inside of each gallery they upload images that they would like to be included in that gallery, and then finally when they are ready they publish the gallery and send a link to their client.

After a client receives a link to a Gallery they are able to share it with all of their friends and family, making it easy to share photos from their events, such as a wedding, without having to email large files to everyone.

Our application will start off incredibly simple, and will only have one page, but as we progress through the book we will slowly introduce new pages and improve our code. Rather than simply jumping right to "this is how you should build web applications" we will instead explore the naive way of building a page, and then explore why this might become problematic and search for solutions to those problems. With this approach we will eventually get to a scalable way to build web applications, and you will have an understanding of how and why we are building our pages the way we are (rather than wondering "but why didn't we do X?").

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. Instead I will be holding your hand and guiding you along the way so that you understand everything we are doing and can expand on what you have learned once you have completed the book.

1.4 How to install Go

If you haven't already, now is a great time to install Go. Rather than writing instructions on how to install Go in this book and watching them quickly get outdated, I am instead opting to reference the official installation guide. You can head over to https://golang.org/doc/install to get started

If you have **ANY** trouble getting Go installed please send me an email - jon@calhoun.io - with the subject "Install help!". In your email include any error messages you might have seen, along with info about what operating system you are using (Windows, Mac, or Linux). I will do my best to help you get setup so that you are prepared for success, and if I can't help I will try to find someone who can help.

1.5 Learning to program using Go

As I mentioned earlier, this book proceeds with the assumption that you have at least installed Go and have learned how to write a basic program using Go. It also assumes that you either have a very basic understanding of Go in general, or that you can pick it up as we go. For example, I won't be explaining what a slice is in Go, but if you are familiar with arrays in most other languages you will likely pick this up quickly.

If anything I mentioned in the last paragraph sounds foreign to you, you might want to check out one of the resources below before proceeding with this book. Do not try to read them all (that would be a waste of your time). Instead pick one or two resources and follow along with a few chapters or lessons until you are starting to feel comfortable with Go, and remember that you can always reference these lessons if you find something confusing in the book. You can also always email me - jon@calhoun.io if you get confused or stuck.

Again, I want to reiterate that these are **NOT** required reading, but instead are intended to help get you up to speed, and as a reference while reading this book. That said, here are those references.

1.5.1 Learn How To Code: Google's Go (golang) Programming Language

If you do not have any programming experience, this a great place to start. This course was designed for people with no programming experience, and will walk you through everything you need to successfully learn programming in general.

This is an udemy course created and instructed by Todd McLeod, a University Professor in Computer Science with over 15 years of teaching experience. The course has **over 30,000** (yes, you read that right, thirty thousand!) **students enrolled** and has a **4.5 out of 5 star rating** as of this writing.

The course costs $35, but if you email me - jon@calhoun.io - I will try to get you a discount code to the course to help offset the cost.

- **URL**: https://www.udemy.com/learn-how-to-code/
- **Author(s)**: Todd McLoed (@Todd_McLeod)
- **Price**: $35 (discounts may be available)

1.5.2 An Introduction to Programming in Go

An Introduction to Programming in Go is a nice free book written by Caleb Doxsey that covers getting started with Go, and most of the major pieces of the language.

The book covers a lot of content relatively quickly, and the high pace may not work for all beginners, but it does work for some. If you are new to programming and are looking for a free resource, this is where I recommend that you start. If you don't mind spending a bit, I instead recommend that you try out the udemy course listed in the previous section, as it is a better course for complete beginners.

- **URL**: https://www.golang-book.com/books/intro
- **Author(s)**: Caleb Doxsey (doxsey.net)
- **Price**: Free

1.5.3 Go By Example

Regardless of your existing programming experience, I highly recommend that you bookmark this resource.

I would not recommend reading this from start to finish, but instead I suggest that you use is as a reference manual when you are learning about something new, like maps.

- **URL**: https://gobyexample.com
- **Author(s)**: Mark McGranaghan (@mmcgrana)
- **Price**: Free

1.5.4 Additional resources

In addition to all of the resources listed above, you can find a list of resources (both paid and free) listed on the golang wiki here: https://github.com/golang/go/wiki/Learn

You can also check out the official Golang tour and docs made available by the Go team, but **if you do not have any programming experience, these are NOT a good place to start.**

- https://tour.golang.org

- https://golang.org/doc/

The official Go documentation is a great place to reference as you learn go, and in my screencasts you will see me frequently referencing the Go docs, but if you are just getting started with the language I do not recommend you spend your first few hours reading these docs.

Instead, bookmark the docs and use them as a reference as you are using new packages in Go.

1.6 Learning HTML, CSS, JavaScript, Command Line, etc

If you are looking for resources to help get you started with these topics I recommend that you check out the Learn Enough to Be Dangerous series by Michael Hartl. Michael is mostly known for his Ruby on Rails Tutorial, but his Learn Enough to Be Dangerous series is a fantastic resource for people wanting to learn just enough about these topics to get started.

Once you have finished these courses you should by no means stop learning about them, but the goal now isn't to master CSS or HTML, but to instead learn enough to get by and create real web applications using these tools, and to expand on your knowledge as you need to. Believe it or not, this is exactly what every real world developer I know has done; They learned just enough to get by and then when there was a need they researched and learned more. Before they even realized it, there were experts with various tools, but had never actually sat down thinking "I am going to become a master of the command line!"

You can find all of these on the Learn Enough website, but here are links to the individual courses that are relevant to this book.

- Learn Enough Command Line to Be Dangerous
- Learn Enough HTML to Be Dangerous
- Learn Enough CSS & Layout to Be Dangerous

- Learn Enough JavaScript to Be Dangerous

Box 1.1. Do I need to know Git?

The short answer: No. Not right now.
While Git is a very helpful tool, and you might see me using it in the screencasts, it isn't a requirement of this book. I opted to not use Git in the first version of this book for two reasons: 1. I feared that adding yet another hurdle for potential developers to get over was unnecessary, and 2. While Git can be incredibly helpful, it can also be incredibly confusing when things go wrong. Instead, I opted to leave learning Git as an exercise for readers to pursue outside of this book.
If you are incredibly interested in source control, the Learn Enough series has a course on Git that should help get you started here: https://www.learnenough.com/git-tutorial

1.7 What editor / IDE / environment should I use?

I personally use Sublime Text, along with GoSublime and a few other custom packages. I plan to eventually make a screencast and blog post covering everything I use, along with instructions on how to set it all up, but unfortunately I haven't had time to do this yet.

I have also heard great things about Atom and the go-plus package for it, so you should also consider checking this out.

You can also use vim, emacs, Notepad++, or any other editor that you prefer.

The truth is, at this point it simply doesn't matter what editor you use. The important part is that you pick something and get started. Once you are more familiar with Go, you can start exploring different editors and you will have a better feel for what packages are helpful, but until then you simply cannot know what will or won't be helpful.

So just pick something and move on!

1.8 Conventions used in this book

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

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

Throughout this book you will see many examples of command-line commands. For simplicity, all of these commands will be prefixed with a Unix-style command line prompt, which is a dollar sign. For example:

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 `testing` line below is is meant to represent the output from the terminal, which is why it is not preceded with a dollar sign.

1.8.2 The `subl` command means "open with your text editor"

You might see me write code that opens up a text file using the `subl` command. This is what I personally use, because I use the Sublime Text text editor. Below is an example.

Listing 1.2: Using `subl` to open a file.

```
$ subl main.go
```

All this means is that I am opening the file `main.go` in my text editor. If you want to do this via command line and you use Atom, you would instead write `atom main.go`. If you use another text editor, you can just open it the same way you would any other file. You DO NOT need to open files via the command line, but this is simply something that I tend to do.

1.8.3 I use `...` to mean unchanged code at times

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 I will frequently use `...`, or sometimes a commented out version like `// ...`, to signify unchanged code.

It is important that **do not** replace existing code with `...` but instead know that this means to leave any other code unchanged in the file. If you are ever uncertain of what to delete or keep, I frequently follow up all of my snippets with a finished version of the file to try to avoid any confusion, so you can read ahead a bit for clarification.

Alternatively, you are always welcome to email me if a code sample in the book is confusing, and I will be happy to help out, as well as try to update the book to be less confusing.

1.8.4 All code samples are limited to 60 columns when possible

Both the **`epub`** and **`mobi`** formats are pretty fickle when it comes to code formatting, and they don't really provide a good way to force code to line wrap. As a result, I am limiting all of my code samples to be at most 60 characters per line (sometimes referred to as 60 columns because I am using a monospaced font for code samples). This should help make the code samples more readable in these formats, but this isn't always possible, so there will still be a few cases where this rule is broken.

If you are interested in a release of this book without the 60 column limit please reach out and let me know. If there is enough demand I might keep two separate versions of the code listings, as this isn't an issue with HTML code samples.

Also, when you are writing your own web applications you should know that you *DO NOT* need to limit your own code to 60 columns. I personally dislike doing this, so if you purchased any package that includes code or screencasts you might see me writing code with less line breaks. If any of it ever confuses you feel free to reach out for clarification - jon@calhoun.io.

Chapter 2

A basic web application

This section is omitted from the sample.

Chapter 3
Adding new pages

This section is omitted from the sample.

Chapter 4

A brief introduction to templates

This section is omitted from the sample.

Chapter 5

Understanding MVC

This section is omitted from the sample.

Chapter 6

Creating our first views

This section is omitted from the sample.

Chapter 7

Creating a sign up page

This section is omitted from the sample.

Chapter 8

An introduction to databases

This section is omitted from the sample.

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
$ subl 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 **??**.

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 manually 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.

9.2 Creating the `UserService` interface

In Section **??** 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 user another database, and if we were to use the `gorm.DB` object scattered 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.

On top of it being hard to test and change databases down the road, we would also have to write a lot of code multiple times. PostgreSQL is case sensitive, which means that if a user were to create an account with the email address JON@CALHOUN.IO it would be considered different than the email address jon@calhoun.io.

We don't want that to happen, so we are going to need to write some code that handles converting all email addresses to lowercase.

In a more general sense, we will likely end up wanting an entire layer of code that handles normalizing our data so that it is always in the same format. This might include converting all email addresses to lowercase, stripping extra spaces from names, or any number of other operations.

On top of a normalization layer, we will probably want to do some data validation before saving our data. We don't want users creating accounts with email addresses that don't even have an `@` sign in them, so we could write a validator to do a few sanity checks to make sure the email field looks valid.

To make all of this possible, we are going to spend some time creating an interface for interacting with our user resource, and we will be naming that interface the `UserServer`.

We will then provide implementations of the `UserService` to our controllers to use to create, update, and delete user resources, but because it is an interface it will give us the freedom to change the implementation behind the scenes without it negatively impacting our controllers.

We are going to continue working in `models/users.go` and we need to add the interface in Listing 9.3 to the file.

Listing 9.3: Declaring the `UserService`

```go
type UserService interface {
  ByID(id uint) *User
  ByEmail(email string) *User
  Create(user *User)
  Update(user *User)
  Delete(id uint)
}
```

9.2.1 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 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.

```go
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.

```go
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.

9.2.2 Examining the `UserService` methods

Now that you understand how interfaces work in Go, the `UserService` should start to make more sense.

Rather than providing a specific type to our controllers, we are going to provide them with an object that implements the `UserService` interface.

By doing this we limit the functionality of what can be done in our controllers, so in a way we are forcing our code to be encapsulated. If we only provide our users controller with a `UserService` implementation, it isn't possible for a developer to start writing code that should reside on the model layer because they won't have access to the `gorm.DB` object. That will only be available to our `UserService` implementation.

What methods are we going to need in our users controller? We won't know for certain until we start using them, but for now it is a pretty safe assumption that we are going to need two retrieval methods - one to look up a user by id, and another to look up a user by their email address.

We are also likely going to need a method to create and update a user object. Each of these can take in the user object that needs created or updated and work from there.

Finally, we are going to need a way to delete a user resource. We don't really need the entire user account to delete it, but instead we just need to know the ID of that account.

9.3 Implementing the `UserService`

In Section 9.2 we talked about implementing a validation layer and a normalization layer, but that is a lot to bite off at once, so instead we are going to break this into a few pieces.

The first thing we are going to focus on is the layer that handles interacting with our database using GORM. This is useful because we can start creating users for our application right away. Even if we end up writing invalid data to our database, we are currently only working in a development environment, so it shouldn't be a big deal if we wipe our database and start over.

Once we have that layer written we will take another pass over our `UserService` and slowly start to introduce a few pieces to help us normalize and validate our data.

9.3.1 Creating the `UserGorm` type

The very first thing we are going to do is update our imports. GORM is going to be using PostgreSQL, so we want to eventually import a package that implements the Postgres driver. Open up `models/users.go` and add the import for the Postgres dialect.

Listing 9.4: Import `lib/pq`

```
import (
  "github.com/jinzhu/gorm"
  _ "github.com/jinzhu/gorm/dialects/postgres"
)
```

Next, we are going to create the `UserGorm` type. This is going to be our implementation of the `UserService` that writes to and reads from our database using GORM.

In order to read and write from our database, we are going to need to open a connection to our database like we did in Section **??**. Rather than doing this every time we need to interact with our database, we are instead going to store the `*gorm.DB` object as part of our `UserGorm` object so that we can reuse the open connection.

Listing 9.5: Creating the `UserGorm` type

```
type UserGorm struct {
  *gorm.DB
}
```

We are also going to want a function to help us construct the `UserGorm` type. While the `gorm.Open()` function takes in both a dialect and a connection string, we are only importing postgres dialect, so we can hard code that for now, leaving us with only one argument for our `NewUserGorm()` function - the connection string.

Listing 9.6: `NewUserGorm` function

```
func NewUserGorm(connectionInfo string) (*UserGorm, error) {
  db, err := gorm.Open("postgres", connectionInfo)
  if err != nil {
    return nil, err
  }
  return &UserGorm{db}, nil
}
```

Finally, we are going to write some stubs for each of the methods we need to implement. These won't actually do anything right now, but we will be filling them in shortly.

Listing 9.7: Stubbing the user service methods

```
func (ug *UserGorm) ByID(id uint) *User {
  return nil
}

func (ug *UserGorm) ByEmail(email string) *User {
  return nil
}

func (ug *UserGorm) Create(user *User) {
}

func (ug *UserGorm) Update(user *User) {
}

func (ug *UserGorm) Delete(id uint) {
}
```

And that's it! We are ready to start implementing each of these functions.

9.3.2 Creating an experimental main package to test with

Before we get started implementing each function, let's go ahead and create a file to test with. That way we can verify that each function is working as we intended as we write it.

Open up the `exp` directory we have been using in the past for experimental code, and then open up the `main.go` file there. That is, open up `exp/main.go`.

We are going to keep a little bit of the code here, but if you didn't save the file don't worry about it. The code we will be starting with is in Listing 9.8.

Specifically, we want to keep the all of the constants that have Postgres connection info, and we are going to want to keep the first line in `main()` where we create the PostgreSQL connection string.

We won't need the `lib/pq` import because the model package should handle that already, so you can delete that import and any others you aren't directly using.

Listing 9.8: Initial code for `exp/main.go`

```go
package main

import (
  "fmt"
)

const (
  host     = "localhost"
  port     = 5432
  user     = "postgres"
  password = "your-password"
  dbname   = "usegolang_dev"
)

func main() {
  psqlInfo := fmt.Sprintf("host=%s port=%d user=%s "+
    "password=%s dbname=%s sslmode=disable",
    host, port, user, password, dbname)
}
```

The first thing we are going to need to do to interact with our `UserGorm` type is to construct one, so we will start out with that code. If there is an error we should panic and figure out what it is before continuing.

If you don't have any tools helping you automatically manage imports, you will also need to import the `usegolang.com/models` package.

Listing 9.9: Construct a `UserGorm`

```
ug, err := models.NewUserGorm(psqlInfo)
if err != nil {
  panic(err)
}
```

After that we will want to initialize a `models.User` object so that we can use it to test our `Create()` method. This is pretty similar to what we were doing in Listing **??** in the last chapter.

Listing 9.10: Initialize a `User`

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

And finally, we are going to be starting with the `Create()` method so that we can create some data to interact with and test our other methods, so we will want to call that method inside of our experimental `main()` function.

Listing 9.11: Calling the create method

```
ug.Create(user)
```

At this point our code is technically complete, but we have an issue. We haven't written any code to manage our database, so unless you have already created a table in your database for the `User` type, it won't exist!

No worries, we can create a temporary method that will help us ensure that our database is reset. Head over to `models/users.go` and add the method in Listing 9.12. Then go back to your `exp/main.go` file and add a call to this function right after you construct the `UserGorm`.

This code will tell GORM to drop the users database table if it exists and then it will recreated it with the call to `AutoMigrate`. While this is handy for testing, it is important that you don't run this in production code as it will delete all of the users in your database!

Listing 9.12: Adding a destructive reset method

```
func (ug *UserGorm) DestructiveReset() {
  ug.DropTableIfExists(&User{})
  ug.AutoMigrate(&User{})
}
```

With everything said and done, your code should match Listing **??**, and if you run it you shouldn't get any errors. Make sure that is true before continuing.

Listing 9.13: Our experimental `main.go`

```
package main

import (
  "fmt"

  "usegolang.com/models"
)

const (
  host     = "localhost"
  port     = 5432
  user     = "postgres"
  password = "your-password"
  dbname   = "usegolang_dev"
)

func main() {
  psqlInfo := fmt.Sprintf("host=%s port=%d user=%s "+
    "password=%s dbname=%s sslmode=disable",
    host, port, user, password, dbname)
  ug, err := models.NewUserGorm(psqlInfo)
  if err != nil {
    panic(err)
  }
  ug.DestructiveReset()

  user := &models.User{
    Name:  "Jon Calhoun",
    Email: "jon@calhoun.io",
  }

  ug.Create(user)
}
```

9.3.3 Implementing the `Create()` method

Open up `models/users.go` if you don't already have it open. We are going to be working on the `Create()` method first because without it we won't have any data to lookup or create, which will make it tricky to actually see our code in action.

Section **??** introduced creating records with GORM, and our `Create()` method won't be any different. We simply need to call the `ug.DB.Create()` method to create our record, and then check to make sure there weren't any errors.

But wait. . . what do we do if we get an error? It looks like we were a little shortsighted when we created our `UserService`, but that should be easy enough to fix.

Update the `Create()` method definition in both the `UserService` and the implementation on the `UserGorm` type to return an error instead of nothing. That way if we get an error while creating a resource we can return it to the controller to handle.

Listing 9.14: Updating and implementing `Create()`

```
type UserService interface {
  ByID(id uint) *User
  ByEmail(email string) *User
  Create(user *User) error
  Update(user *User)
  Delete(id uint)
}

// ...

func (ug *UserGorm) Create(user *User) error {
  return ug.DB.Create(user).Error
}
```

Head back over to our experimental code at `exp/main.go` and update our call to `Create()` to panic if there is an error, and otherwise have it print out the newly created user. Then go ahead and run the code.

Listing 9.15: Panic if an error occurs

```
if err := ug.Create(user); err != nil {
  panic(err)
}
fmt.Println(user)
```

If everything went according to plan, you should see output in your terminal that lists all of the user's attributes, including an `ID` assigned by the database, and matching `CreatedAt` and `UpdatedAt` dates.

Congrats! Your have successfully implemented the `Create()` method. Now we just need to finish up the rest methods using what we learned in Section **??**.

9.3.4 Updating `exp/main.go` to test all of our methods

We are going to go a little faster for the rest of these method because they are all reviewing information that we covered in Section **??**, so rather than update `exp/main.go` in each section we are instead going to just update it to use every method.

We are also going to update all of our methods to return an error so that we can return errors when they do occur, so let's start there so our code continues to compile. Open `models/users.go` and update it to reflect Listing **??**.

Listing 9.16: Updating our user service to return errors

```
package models

import (
  "github.com/jinzhu/gorm"
  _ "github.com/jinzhu/gorm/dialects/postgres"
)

type User struct {
  gorm.Model
  Name  string
  Email string `gorm:"not null;unique_index"`
}
```

```go
type UserService interface {
  ByID(id uint) *User
  ByEmail(email string) *User
  Create(user *User) error
  Update(user *User) error
  Delete(id uint) error
}

func NewUserGorm(connectionInfo string) (*UserGorm, error) {
  db, err := gorm.Open("postgres", connectionInfo)
  if err != nil {
    return nil, err
  }
  return &UserGorm{db}, nil
}

type UserGorm struct {
  *gorm.DB
}

func (ug *UserGorm) ByID(id uint) *User {
  return nil
}

func (ug *UserGorm) ByEmail(email string) *User {
  return nil
}

func (ug *UserGorm) Create(user *User) error {
  return ug.DB.Create(user).Error
}

func (ug *UserGorm) Update(user *User) error {
  return nil
}

func (ug *UserGorm) Delete(id uint) error {
  return nil
}

func (ug *UserGorm) DestructiveReset() {
  ug.DropTableIfExists(&User{})
  ug.AutoMigrate(&User{})
}
```

After that update `exp/main.go` to make a call to each of our methods and print out the results. We will be utilizing the first model that we create for this along with creating a few others.

Listing 9.17: Testing everything

```
package main

import (
  "fmt"

  "usegolang.com/models"
)

const (
  host     = "localhost"
  port     = 5432
  user     = "postgres"
  password = "your-password"
  dbname   = "usegolang_dev"
)

func main() {
  psqlInfo := fmt.Sprintf("host=%s port=%d user=%s "+
    "password=%s dbname=%s sslmode=disable",
    host, port, user, password, dbname)
  ug, err := models.NewUserGorm(psqlInfo)
  if err != nil {
    panic(err)
  }
  ug.DestructiveReset()

  user := &models.User{
    Name:  "Jon Calhoun",
    Email: "jon@calhoun.io",
  }

  fmt.Println("Creating a user...")
  if err := ug.Create(user); err != nil {
    panic(err)
  }
  fmt.Println("Created the user:", user)

  fmt.Println("Retrieving the user w/ ID:", user.ID)
  userByID := ug.ByID(user.ID)
  if userByID == nil {
    panic("No user found by ID!")
  }
  fmt.Println("Found the user:", userByID)

  fmt.Println("Updating the user...")
  user.Email = "jon@usegolang.com"
  if err := ug.Update(user); err != nil {
    panic(err)
  }
  fmt.Println("Updated the user:", user)

  fmt.Println("Retrieving the user w/ email:", user.Email)
  userByEmail := ug.ByEmail(user.Email)
  if userByEmail == nil {
    panic("No user found by email!")
  }
  fmt.Println("Found the user:", userByEmail)

  fmt.Println("Deleting the user...")
  if err := ug.Delete(user.ID); err != nil {
    panic(err)
  }
  // Verify that the user was deleted
  deletedUser := ug.ByID(user.ID)
  if deletedUser != nil {
    panic("User was found but should be deleted!")
  }
  fmt.Println("Deleted the user w/ ID:", user.ID)
}
```

There is a lot going on here, but the comments and print statements should make it pretty obvious what we are doing. If not, feel free to reach out - jon@calhoun.io.

If you run this code it **is supposed to panic**. That is because we haven't implemented several methods, so our code is complaining when it doesn't work.

Box 9.1. What we are doing is similar to TDD

What we are doing is very similar to test-driven development (aka TDD), which is a software development process where you write tests prior to writing your code and then verify that your code is passing your tests as you write it.
In our case, we aren't writing actual test cases, but our experimental program does serve the purpose of helping us determine if our implementations are correct as we write them.
Regardless of how you test your code, this should pretty clearly demonstrate some of the value touted by advocates of TDD. With test code we can quickly verify whether our code is correct as we develop it.

9.3.5 Implementing the `ByID()` method

The first method failing is the `ByID()` method, and that is because it never even tries to look up a user!

Box 9.2. Why doesn't `ByID()` return an error?

You might be wondering why we don't also return an error on this method, and the answer is pretty simple - when we don't find a user with the provided user we can simply return nil. If any errors do occur, it isn't an error that we expect, but that isn't always the case with methods like `Create()` and `Update()`.
For example, we might try to create a user with an email address that already exists and return an error when that happens. A duplicate email error is one that we expect our controller to handle, as it will need to inform the end user about the error so that they can change their email address.
With the `ByID()` method we don't have errors like this. If a user isn't found, we simply return nil, but if an error occurs it likely means that something is going wrong with our server. When those errors occur we don't need our controller handling them, but instead might want to panic our application and restart it.
At the end of the day, choosing to return errors on your `ByID()` method is completely up to you, but I am opting not to for now.

Implementing the `ByID()` method should only take a few lines of code that we have seen before, but we will be adding one new bit.

When we can't find a user we want to return nil instead of panicking, so when there is an error we need to check to see what type of error we got.

Luckily, GORM provides us with a list of the errors we can expect in their docs - https://godoc.org/github.com/jinzhu/gorm#pkg-variables. It looks like we just need to check to see if our error is an `gorm.ErrRecordNotFound` and if it is, return nil.

To achieve this we are going to use a switch statement. If you aren't familiar with them, I suggest you read up on them in effective go - https://golang.org/doc/effective_gc

Listing 9.18: Implmenting **ByID()**

```
func (ug *UserGorm) ByID(id uint) *User {
  ret := &User{}
  err := ug.DB.First(ret, id).Error
  switch err {
  case nil:
    return ret
  case gorm.ErrRecordNotFound:
    return nil
  default:
    panic(err)
  }
}
```

Run your experimental code and verify that it is now erroring when trying to look up a user by email.

Why are we panicking at runtime?

One thing worth mentioning here is that our **ByID()** function has a **panic()** call that could potentially occur while our web application is serving a web request.

In the past, all of our calls to panic occurred during setup. That is, if there was an error, it would halt the web server from ever starting and would terminate our entire program.

An example of this is when we try to create a view using a template file. If the template doesn't exist or can't be parsed an error will be returned and our code panics with the error preventing our web server from ever starting up.

The **ByID()** use case is different because our program can start up, be running, and then after a few hours it might suddenly encounter an error when our database temporarily goes offline. If that happens, this code is very likely to panic. In short, we aren't following the Go proverb of "Don't just check errors, handle them gracefully."

Before we end up deploying our web application we are eventually going to start adding middleware. You don't need to know what middleware is right now, but what you do need to know is that we will also be adding some middleware intended to helper our web application recover from panics. It will do this by returning the 500 HTTP status code along with a page explaining that something went wrong while our server was attempting to process the request.

As our web application evolves it may make sense to add in better error handling here. For example, we could attempt to retry the query after waiting a few seconds. But for now, it is much simpler to just panic and let our recovery middleware tell the user that something went wrong. That way we don't invest too much time trying to handle every possible error path when some of them might not ever actually occur in production.

So in short, yes we are breaking one of the Go proverbs to a degree, and generally speaking I do not suggest doing this, but in this case I feel that it is worth doing until we have a real need to handle the error more gracefully.

9.3.6 Implementing the `Update()` method

At first our failure might appear to be coming from `ByEmail()`, but in reality there are two reasons why our code is failing.

The first is indeed that `ByEmail()` isn't implemented, but the second is that we haven't implemented the `Update()` method, so our code never actually updates the record in our database. As a result, the email address is never updated!

Updating records in GORM is pretty similar to creating them, but instead of calling a create method we instead call a save method.

Listing 9.19: Implmenting `Update()`

```
func (ug *UserGorm) Update(user *User) error {
  return ug.DB.Save(user).Error
}
```

9.3.7 Implementing the `ByEmail()` method

Now that we are actually updating our user's email address, the only thing currently causing our code to panic is the lack of an implementation for `ByEmail()`.

We covered how to query for records by a specific attribute in Listing **??** in the last chapter, so that code should look familiar. The rest of our code is going to be very similar to the `ByID()` method.

Listing 9.20: Implmenting `ByEmail()`

```go
func (ug *UserGorm) ByEmail(email string) *User {
  ret := &User{}
  err := ug.DB.Where("email = ?", email).First(ret).Error
  switch err {
  case nil:
    return ret
  case gorm.ErrRecordNotFound:
    return nil
  default:
    panic(err)
  }
}
```

Take a moment to stop and run your code again. It should now be panicking when it discovers that a deleted user still exists instead of when it tries to look up a user by their email address, but before we move on I want to take a minute to look at the code in `ByID()` and `ByEmail()`.

Doesn't this code look incredibly similar? In fact, we really only have one line that is different between the two.

When code looks nearly identical in two places that is usually a sign that it might be time to refactor the code into a reusable function of its own. Before moving on, let's take a moment to do that.

Listing 9.21: Refactoring with a `byQuery` method

```go
func (ug *UserGorm) ByID(id uint) *User {
  return ug.byQuery(ug.DB.Where("id = ?", id))
}

func (ug *UserGorm) ByEmail(email string) *User {
  return ug.byQuery(ug.DB.Where("email = ?", email))
}

func (ug *UserGorm) byQuery(query *gorm.DB) *User {
  ret := &User{}
  err := query.First(ret).Error
  switch err {
  case nil:
    return ret
  case gorm.ErrRecordNotFound:
    return nil
  default:
    panic(err)
  }
}
```

When we call `ug.DB.Where(...)` it returns a pointer to a `gorm.DB` object with the query we have built so far. By taking advantage of this fact, we are able to reduce both our `ByID()` and `ByEmail()` methods to just one line each, and then all of the shared code can be maintained in the `byQuery()` method which isn't accessibly to anything outside of the models package.

With our refactor complete, we are now ready to implement the final method - the delete method.

9.3.8 Implementing the `Delete()` method

In order to delete records with GORM you need to construct an instance of the type and then pass it into the Delete() method on the `gorm.DB` type.

Because of the requirement to create a `User`, it might be tempting to just have our own delete method take in a `User` pointer, but I am going to urge you not to do this.

If you read the GORM docs for deleting resources - http://jinzhu.me/gorm/crud.html#del - one of the things that is very important to notice is the line "if primary field's blank, GORM will delete all records for the model".

This means that if we were to accidentally forget to set the ID field on our `User` we pass in to be delete, there is a chance that **all** of our users could be delete.

Yikes! That definitely isn't what we want, so rather than letting users of the `UserService` make that mistake, we are instead going to force them to pass us an `ID` which we will then place in a user object that we construct, minimizing the number of potential failure points in our code to just one - our `Delete()` method.

Listing 9.22: Implmenting `Delete()`

```go
func (ug *UserGorm) Delete(id uint) error {
  user := &User{Model: gorm.Model{ID: id}}
  return ug.DB.Delete(user).Error
}
```

Delete was the last method we needed to implement, so if you run your experimental code again you shouldn't see any errors. We have successfully implemented the `UserService`!

9.4 Utilizing the user model in the users 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 plug everything together.

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 I still want to show you how you can connect everything together and go from sign up form to writing a record to the database.

9.4.1 Adding the user service to the users controller

Before we can use the `UserService` in our users controller we need to add it to the controller. We don't want to have to initialize a service every single request, so instead we are going to update our `controllers.Users` type so that it has a `UserService` field.

We will then update the `NewUsers()` function to accept a `UserService` as an argument so that our controller and use that for the field. Don't forget to also import the `models` package.

Listing 9.23: Adding the user service to the users controller

```go
package controllers

import (
  "fmt"
  "net/http"

  "usegolang.com/models"
  "usegolang.com/views"
)

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

type Users struct {
  NewView *views.View
  models.UserService
}

// ... nothing changes below here
```

If we try to run our code now we are going to get an error because we we are no longer constructing our users controller correctly in **`main.go`**, so now we are going to take some time to update that file.

NOTE - I am referring to our primary **`main.go`** file, not the experimental one. It should be located in your root directory.

The first thing we are going to need to do is set up some variables to store our database connection info. These should be located in your experimental code - **`exp/main.go`** - so copy the constants from there and add them to your **`main.go`** file.

Listing 9.24: Add your database info to **`main.go`**

```go
const (
  host     = "localhost"
  port     = 5432
  user     = "postgres"
  password = "your-password"
  dbname   = "usegolang_dev"
)
```

Box 9.3. 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.

After that we need to create a `UserGorm` instance that we can provide as our `UserService` implementation, and then pass that into the users controller when we construct it. Once again, be sure to import the models package!

Listing 9.25: Provide a `UserService` to the user controller

```
package main

import (
  "fmt"
  "net/http"

  "usegolang.com/controllers"
  "usegolang.com/models"

  "github.com/gorilla/mux"
)

const (
  host     = "localhost"
  port     = 5432
  user     = "postgres"
  password = "your-password"
  dbname   = "usegolang_dev"
)
```

```
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)
  ug, err := models.NewUserGorm(psqlInfo)
  if err != nil {
    panic(err)
  }

  staticC := controllers.NewStatic()
  usersC := controllers.NewUsers(ug)

  r := mux.NewRouter()
  r.Handle("/", staticC.Home).Methods("GET")
  r.Handle("/contact", staticC.Contact).Methods("GET")
  r.HandleFunc("/signup", usersC.New).Methods("GET")
  r.HandleFunc("/signup", usersC.Create).Methods("POST")
  http.ListenAndServe(":3000", r)
}
```

We now have a user service available in our controller, but before we can use it to create a user we need to update our view. Our sign up form doesn't currently ask for a user's name, so we need to update that first.

9.4.2 Adding the name field the sign up form

In `views/users/new.gohtml` update the sign up form template so that it now asks the user for a name as well as their email address. We are going to keep the password field, but we won't be storing that value in our database until we learn how to properly handle hashing passwords.

Listing 9.26: Add the name field to the sign up form

```
{{define "signupForm"}}
<form action="/signup" method="POST">
  <div class="form-group">
    <label for="name">Name</label>
    <input type="name" name="name" class="form-control" id="name" placeholder="Your full name">
  </div>
  <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="Pass
  </div>
  <button type="submit" class="btn btn-primary">Sign Up</button>
</form>
{{end}}
```

After restarting your server and heading over to the sign up page you should have a form that matches Figure 9.1.

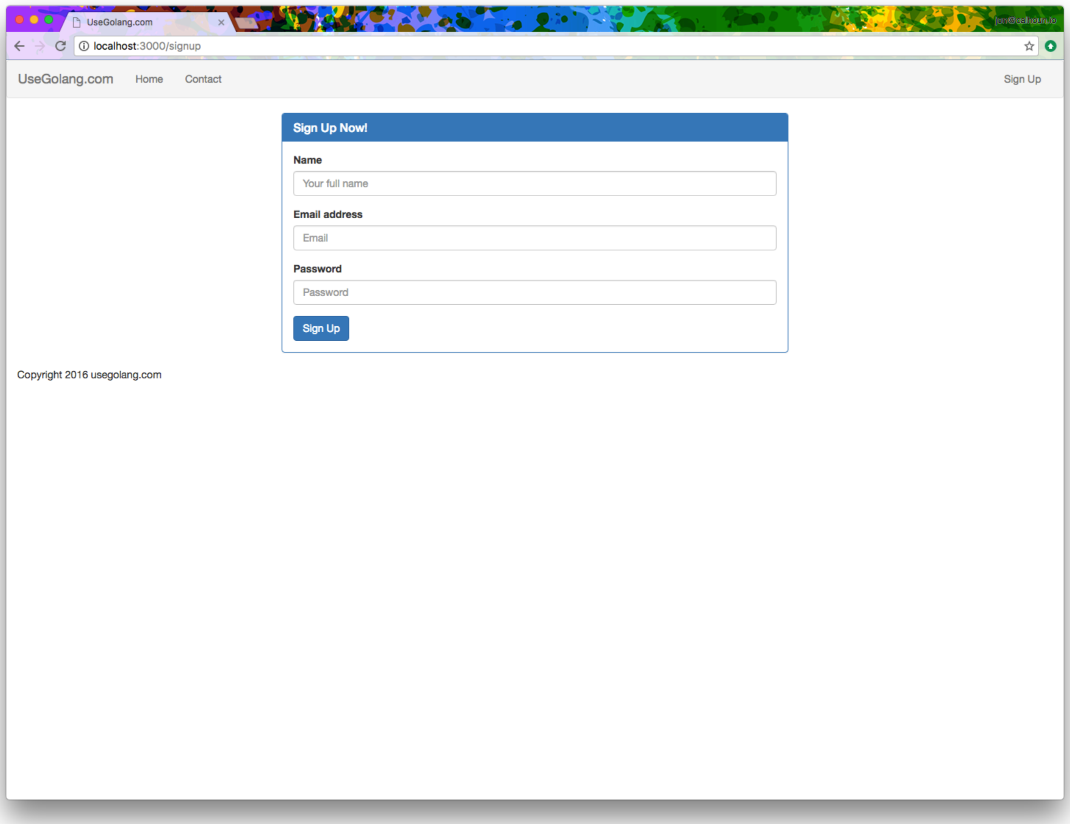

Figure 9.1: Screenshot of the sign up form

9.4.3 Parsing the sign up form and creating a user

Now that our sign up form has a name field, we need to update our create action on the users controller so that it parses the field and assigns it to a new user object.

After that we need to take the data and construct a user model and pass that into the user service's create method and verify that we didn't get any errors. For now we won't do anything with the error, but in the future we will display an error message on render our sign up form again.

Listing 9.27: Creating real user resources

```go
type SignupForm struct {
  Name     string `schema:"name"`
  Email    string `schema:"email"`
  Password string `schema:"password"`
}

// This 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) {
  form := SignupForm{}
  if err := parseForm(r, &form); err != nil {
    panic(err)
  }
  user := &models.User{
    Name:  form.Name,
    Email: form.Email,
  }
  if err := u.UserService.Create(user); err != nil {
    panic(err)
  }
  fmt.Fprintln(w, user)
}
```

Restart your server and try to create an account. Make sure you use an email address that isn't already registered though, otherwise the server will get an error from the user service's `Create()` method and panic.

9.4.4 Automigrating the user model

We aren't going to be making any changes to our user model now, but in the future we are likely going to add a few new fields to our user type. It would be nice if we our database automatically updated with these new fields when we introduced them, so we are going to add one more method to the `UserGorm` type - the `AutoMigrate()` method.

This is going to be similar to the `DestructiveReset()` method that we used in experimental, but it won't drop the table. It will only call `AutoMigrate()` on the `gorm.DB` field which will cause new fields to be added, but **will not delete or change existing fields**.

We will also need to update the `DestructiveReset()` method because the `AutoMigrate()` method will no longer be delegated to the `gorm.DB` object by default, but will instead use the method we define on the `UserGorm` type.

Listing 9.28: Adding an auto-migration method

```go
func (ug *UserGorm) DestructiveReset() {
  ug.DropTableIfExists(&User{})
  ug.AutoMigrate()
}

func (ug *UserGorm) AutoMigrate() {
  ug.DB.AutoMigrate(&User{})
}
```

Next, add this code to our `main()` function right after the declaration of the `UserGorm`.

Listing 9.29: Auto migrate the user table in `main()`

```go
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)
  ug, err := models.NewUserGorm(psqlInfo)
  if err != nil {
    panic(err)
  }
  ug.AutoMigrate()

  // ... this stays the same as it was
}
```

Your table should now be automatically migrating, so as you add new fields to your `models.User` type the database will get updated with the new columns.

Remember that this does not delete or update existing columns. If you need to do that you will need to drop the column (or the entire table) first, or write some custom SQL to handle the change.

9.4.5 How to stub the user service for testing

While we aren't going to be testing in this book, I wanted to take a moment to explain how the `UserService` helps make testing easier.

Remember back in Listing 9.25 where we passed the `models.UserGorm` object into the `controllers.NewUsers()` function to create a new controller? Well if we wanted to test without using GORM we would just need to change the type that we pass into our controller when setting up our test application.

As long as that type implements all of the methods in the `UserService` interface it would work without a problem, so for example we could use the code in Listing 9.30 to implement a user service for testing and we would be able to verify that the correct values were sent to the user service methods without actually using our real user service.

Listing 9.30: A fake user service for testing

```
// Do not actually code this
// It is for illustration purposes only
type TestUserService struct {
  Received map[string]interface{}
}
```

```
func (tus *TestUserService) ByID(id uint) *models.User {
  tus.Received["id"] = id
  return nil
}

func (tus *TestUserService) ByEmail(email string) *models.User {
  tus.Received["email"] = email
  return nil
}

func (tus *TestUserService) Create(user *models.User) error {
  tus.Received["user"] = user
  return nil
}

func (tus *TestUserService) Update(user *models.User) error {
  tus.Received["user"] = user
  return nil
}

func (tus *TestUserService) Delete(id uint) error {
  tus.Received["user"] = user
  return nil
}
```

Verifying argument values is a breeze because we simply need to check the `Received` field on the `TestUserService` object that we pass in during testing, and we don't have to worry about resetting a database or anything else to test our controller.

Then, if we wanted to test our `UserService` implementation, we could write tests that only test the `UserGorm` object without needing to create a controller or even visit a single web page, which vastly simplifies the tests.

On top of simplifying tests, it also becomes much clearer where a bug is if one pops up. Before we wouldn't really know if it was in the controller or the model, but any failures now would pop up in the correct layer's tests because the tests only work with code from that layer. Everything else is stubbed out with test interfaces.

Neat, right?

9.5 Exercises

The exercises here are meant to help you think a bit about what we did and how you could adapt the code for a different resource.

1. What all would you need to create for a galleries model layer?

2. What methods do you think you would need on a `GalleriesService`?

3. Try coding up a type like the test one in Listing 9.30. Instead of storing the arguments in a map, print them out to the terminal and then pass that interface to our users controller instead of the real one. Then go try to create an account and verify that it prints out information to the terminal.

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 you 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 he New York Times with a headline like "Data breach could put your accounts at risk."

10.1 Why not use another package or service?

Before continuing on and building our own authentication system, I want to first address a fairly common question. That question 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 do something on your web application to make an otherwise secure authentication system insecure.

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 might require passwords to be reset monthly and never be repeated.

There is no shortage of random criteria that can be added to your authentication system, so it is extremely unlikely that you will be able to take an off-the-shelf solution and just plug it into your web application. That means customization will be necessary, and 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

As I said before, building a proper authentication system isn't hard, but instead there are several simpler smalls pieces that need to be taken into consideration and put together to make a wholly secure system.

This isn't specific to Go, but instead is just a generally good idea.

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`, and then you should redirect all of your users to the `https` version.

We won't actually get to this step until later in the book when we prepare for production and deploy our application, but I wanted to add a section discussing it to the authentication chapter because it would feel incomplete without it.

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 users 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 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. Encrypting functions aren't 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.

Update the `models.User` type

Now that you understand how we are going to be storying passwords, let's go ahead and look at some code. The first thing we are going to do is add the `Password` and `PasswordHash` fields to our user in `models/users.go`.

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"`
}
```

The first field, `Password`, will be used to enter in the password that a user provides, but we **will not** save it to the database. That is why we added the `gorm:"-"` struct tag; This tells GORM to not save this field in the database.

The second field, `PasswordHash`, will get stored in our database, and will store a hashed password that is safe to store in our database.

Set the hashed password field in our user service

We have a field to store our hashed password, but our code doesn't actually put anything into that field yet, so the next thing we are going to is update our code so that the password gets hashed and stored in that field.

The first thing we need to do is `go get` the bcrypt package.

Listing 10.3: Installing bcrypt

```
$ go get -u golang.org/x/crypto/bcrypt
```

After that we can open up `models/users.go` and update the `Create()` method on the `UserGorm` type with the code in Listing 10.5. You will also need to make sure you add in the imports from Listing **??**.

Listing 10.4: bcrypt imports

```
import (
  "github.com/jinzhu/gorm"
  _ "github.com/jinzhu/gorm/dialects/postgres"
  "golang.org/x/crypto/bcrypt"
)
```

Listing 10.5: Hashing the password with bcrypt

```
func (ug *UserGorm) 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 ug.DB.Create(user).Error
}
```

There is a lot going on here, so let's take a moment to review the code in Listing 10.5.

The first thing we are doing is calling `bcrypt.GenerateFromPassword()`. This function requires a byte slice as the first argument, so we need to convert our password string into a byte array. That is the `[]byte(user.Password)` part.

The bcrypt function also requires a cost parameter, but what is that? Well, when you hash using bcrypt it can adjust how much work is required to hash the password. This is used to make it harder for malicious parties to use rainbow tables to try to figure out passwords stored in our database.

We will discuss those in a bit more detail when we talk about adding salting our password, but for now all you need to know is that larger costs will cause our hashing function to take more time to calculate a final value, and lower values will calculate values faster. We want to use a value large enough that attackers can't hash millions of passwords with ease, but low enough that our user isn't waiting a long time on our server. The `bcrypt.DefaultCost` should be a nice happy medium for us to use.

The `bcrypt.GenerateFromPassword()` function then returns a byte slice, and an error. We return the error if we get one, otherwise we want to store a string in our database, so we convert the byte slice returned into a string with `string(byteSlice)`, and then set the value to the `user.PasswordHash` field.

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.Password` field to an empty string, I prefer to clear our a user provided password as soon as possible to minimize any chances of it being leaked in logs or somewhere else in our code.

Update our create action on the users controller

The password hash won't do us much good without a password being parsed from the sign up form. Open up `controllers/users.go` and update the `Create()` method there to set the password with the value provided in the form.

Listing 10.6: Use the sign up form password

```
func (u *Users) Create(w http.ResponseWriter, r *http.Request) {
  form := SignupForm{}
  if err := parseForm(r, &form); err != nil {
    panic(err)
  }
  user := &models.User{
    Name:     form.Name,
    Email:    form.Email,
    Password: form.Password,
  }
  if err := u.UserService.Create(user); err != nil {
    panic(err)
  }
  fmt.Fprintln(w, user)
}
```

Now when we call the `UserService.Create()` method it will have a password to hash and assign to the PasswordHash field. Even if someone were to look at the data in your database, all they would see is what looks like a gibberish string in the `PasswordHash` field.

10.3.3 Salt and pepper the password

We have one last thing left to do before we begin permanently 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.

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.

This might seem like a waste of time at first; Earlier I told you the it is impossible to go from a hashed value back to the original value, so why would we need to add fake data to our original password before hashing it?

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
aksdfj
```

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-aksdfj
```

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.

Implementing a salt

Luckily, we don't actually need to do anything to implement a salt in our program because the `bcrypt` encryption algorithm already does this for us. The resulting hash value after running `bcrypt.GenerateFromPassword()` has the salt stored in it. In fact, it also has both the cost argument used along with the version of bcrypt used to hash the password.

I only mentioned salts in this book because it is important to understand what they are, what they are used for, and to be aware that it is something you need in your code if your chosen hashing function doesn't have this built into it.

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 I definitely recommend adding it to your applications.

Implementing a pepper

Open up `models/users.go` and add the package variable `userPwPepper`. You can pick whatever value you want for your string.

Listing 10.7: Adding a pepper variable

```go
var userPwPepper = "secret-random-string"
```

Next we need to add the pepper to our user provided password. Head over to `UserGorm.Create()` in the same file and update it so that the pepper is appended to the password before being hashed.

Listing 10.8: Add a pepper to the password when creating users

```
func (ug *UserGorm) Create(user *User) error {
  hashedBytes, err := bcrypt.GenerateFromPassword(
    []byte(user.Password+userPwPepper),
    bcrypt.DefaultCost)
  if err != nil {
    return err
  }
  user.PasswordHash = string(hashedBytes)
  user.Password = ""
  return ug.DB.Create(user).Error
}
```

With a pepper in place we are done with the code necessary to hash a user's password! If you want to give it a test run, restart your applications and sign up for an account.

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 such a well known duo.

10.4 Authenticating returning users

Having user accounts and hashed passwords don't do us much good if we can't authenticate a user when they return to our website, so for the rest of the chapter we are going to focus on creating a log in form that accepts an email address and passwords and verifies whether or not they are correct.

Eventually we will need to use sessions or cookies to log a user in and remember them as they browse different pages on our application, but we will leave that for next chapter. In this section we are going to focus purely on the code to accept the form input and tell us if the information was correct, starting with the input form template.

10.4.1 Create the login template

Our log in form is going to be be nearly identical to our sign up form with two major distinctions. The first is that it is going to POST to the `/login` path instead of the `/signup` path, and the second is that it won't have a name field.

Create the file `views/users/login.gohtml` and add the code in Listing 10.9.

Listing 10.9: Creating the login template

```
{{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">Welcome Back!</h3>
      </div>
      <div class="panel-body">
        {{template "loginForm"}}
      </div>
    </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}}
```

10.4.2 Create the login action

If it wasn't clear from the template location, we are going to be adding the login action to the users controller.

Technically speaking, logging in creates a session, so many developers will choose to create a sessions controller and place actions for logging in and logging out there, but I personally prefer to put these actions under the users controller because it isn't really possible to create a session in our application without an associated user.

Head over to your users controller at `controllers/users.go`. In this file we are going to add a `Login()` method to the `Users` type, and we will also be adding a `LoginView` field to the `Users` type, and initializing this field in the `NewUsers()` function.

Listing 10.10: Adding the `LoginView` to the users controller

```
func NewUsers(us models.UserService) *Users {
  return &Users{
    NewView:     views.NewView("bootstrap", "users/new"),
    LoginView:   views.NewView("bootstrap", "users/login"),
    UserService: us,
  }
}

type Users struct {
  NewView   *views.View
  LoginView *views.View
  models.UserService
}
```

Listing 10.11: Adding the login action to the users controller

```go
type LoginForm struct {
  Email    string `schema:"email"`
  Password string `schema:"password"`
}

func (u *Users) Login(w http.ResponseWriter, r *http.Request) {
  form := LoginForm{}
  if err := parseForm(r, &form); err != nil {
    panic(err)
  }
  // Do something to see if the user is valid
}
```

This code is once again very similar to our sign up code (the `New()` method), but after we parse our form we want to call some kind of method to determine if the provided information is valid. We will be calling this method `Authenticate()` and will be writing it in the next secion, but first let's take moment to add the `Login()` action to our router.

Listing 10.12: Adding the login routes

```go
func main() {
  // ... nothing changes here
  r.Handle("/login", usersC.LoginView).Methods("GET")
  r.HandleFunc("/login", usersC.Login).Methods("POST")
  // ... nothing changes here
}
```

10.4.3 Add the `Authenticate()` method to the `UserService`

In this section we will be working in `models/users.go` to add the `Authenticate()` method to the user service.

The goal of this method is to take in an email address and a password and then verify if it matches a user in our database. If it does match, we will return that user object, and if it doesn't match for any reason we will return `nil`.

In either case, whatever code calls this method will be expected to use the results to create a session or do whatever else is necessary. Our `UserService` is solely responsible for validating the info and returning a user if one matches.

First up is the `UserService`. We need to add the `Authenticate()` method to the interface definition.

Listing 10.13: Add the `Authenticate()` method to interface

```
type UserService interface {
  ByID(id uint) *User
  ByEmail(email string) *User
  Authenticate(email, password string) *User
  Create(user *User) error
  Update(user *User) error
  Delete(id uint) error
}
```

Next, we need to implement this new method with our user service implementation, the `UserGorm` type.

As we just saw, the `Authenticate()` method is going to take in an email address and a password and return a user. That means the first thing we need to do is take the given email address and try to find a user with that email address. If we find one we will check to see if we were given the correct password for that user, but if we can't find a user with that email address we can immediately return `nil`. Not finding a user is a clear sign that the provided info isn't correct.

If we find a user, our next step in authenticating them is to compare the hashed password of that user with the password that was passed into our `Authenticate()` method. `bcrypt` provides us with a function to compare a hash and a password, so we are going to leverage it.

The `CompareHashAndPassword()` function will return an error if the passwords don't match for any reason; Otherwise it will return nil. That means that our code should return nil if there was any error returned at all, but if we don't get any errors we are safe to return the user we originally looked up with the email address.

Listing 10.14: Implementing the `Authenticate()` method

```
func (ug *UserGorm) Authenticate(email, password string) *User {
  foundUser := ug.ByEmail(email)
  if foundUser == nil {
    // No user found with that email address
    return nil
  }

  err := bcrypt.CompareHashAndPassword(
    []byte(foundUser.PasswordHash),
    []byte(password+userPwPepper))
  if err != nil {
    // Invalid password
    return nil
  }

  return foundUser
}
```

10.4.4 Connecting all the pieces of our login code

The final thing left to do is to connect all of these pieces together and test out our login flow.

Head over to `controllers/users.go` and finish coding up the `Login()` method. We need to call the `UserService.Authenticate()` method using the data that we parsed from the form and stored in the `LoginForm`. Then once we have a user we will simply print that user object out to the screen to verify that it is correct.

Listing 10.15: Finishing the login action

```
func (u *Users) Login(w http.ResponseWriter, r *http.Request) {
  form := LoginForm{}
  if err := parseForm(r, &form); err != nil {
    panic(err)
  }
  user := u.UserService.Authenticate(form.Email, form.Password)
  fmt.Fprintln(w, user)
}
```

We already added routes to our application in Listing 10.12, but what we didn't do is update our navigation. Without doing that, visitors won't have a way to get to our login page.

Open up `views/layouts/navbar.gohtml` and add a link to the `/login` page in the `navbar-right` section. The final code is shown in Listing 10.16.

Listing 10.16: Adding the login page to our navigation

```
{{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=
        <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="/">UseGolang.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">
        <li><a href="/login">Log In</a></li>
        <li><a href="/signup">Sign Up</a></li>
      </ul>
    </div>
  </div>
</nav>
{{end}}
```

Finally, restart your server and test it out. You will need to create an account if you haven't already, and then go to the login page to try to log in with the same email address and password.

If you type in the correct email address and password combination you should see a long string printed out to your screen that has the email address, a few timestamps, and the hashed password of the user you just logged in as.

If you type in an incorrect email address or password you should just get back `<nil>`, which means that no user was returned by the `UserService.Authenticate` method call.

10.5 Exercises

I mentioned this earlier, but our code currently doesn't actually handle creating user sessions. That means that no matter how many times a user logs in, our application won't remember that user. In order to remember a user, we need to implement some sort of session.

Next chapter we are going to focus on creating sessions and using them to limit user's access to only pages they are allowed to see, but first let's take a moment to review some of the things we learned about building a secure authentication system by doing some exercises and answering a few questions.

1. What is a hash function?

2. What purpose does a salt and pepper serve? What makes these two different?

3. I didn't cover this in this chapter, but do a quick Google search on timing attacks. The CompareHashAndPassword function in the **`bcrypt`** package helps protect us from these, but it is good to be aware of what they are.