

Let's Go!

LET'S GO!

A STEP-BY-STEP GUIDE TO CREATING FAST, SECURE AND
MAINTAINABLE WEB APPLICATIONS WITH GO



ALEX EDWARDS

2ND EDITION

Let's Go teaches you step-by-step how to create fast, secure and maintainable web applications using the fantastic programming language [Go](#).

The idea behind this book is to help you *learn by doing*. Together we'll walk through the start-to-finish build of a web application — from structuring your workspace, through to session management, authenticating users, securing your server and testing your application.

Building a complete web application in this way has several benefits. It helps put the things you're learning into context, it demonstrates how different parts of your codebase link together, and it forces us to work through the edge-cases and difficulties that come up when writing software in real-life. In essence, you'll learn more than you would by just reading Go's (great) documentation or standalone blog posts.

By the end of the book you'll have the understanding — and confidence — to build your own production-ready web applications with Go.

Although you can read this book cover-to-cover, it's designed specifically so you can follow along with the project build yourself.

Break out your text editor, and happy coding!

— Alex

Contents

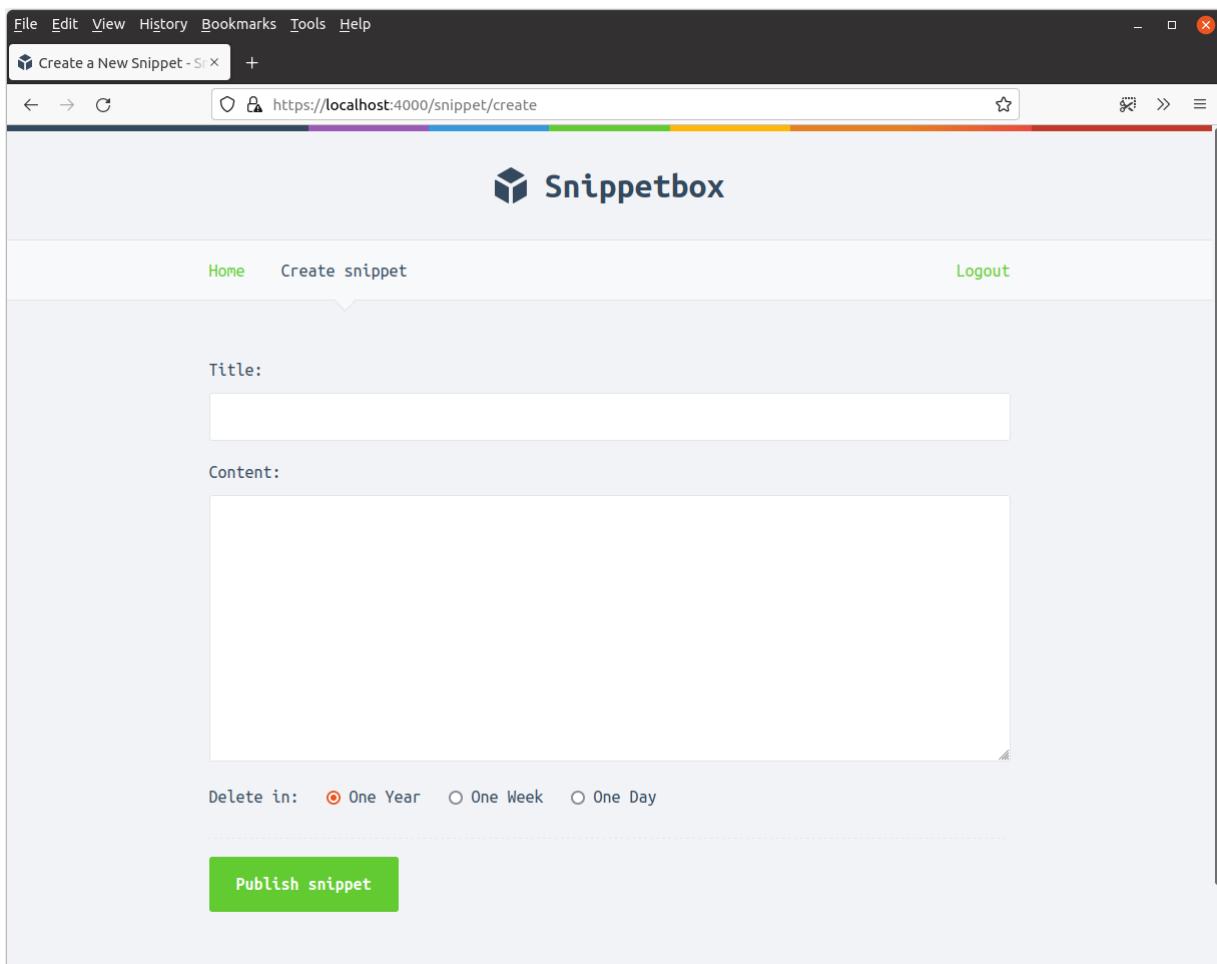
- **1. Introduction**
 - 1.1. Prerequisites
- **2. Foundations**
 - 2.1. Project setup and creating a module
 - 2.2. Web application basics
 - 2.3. Routing requests
 - 2.4. Wildcard route patterns
 - 2.5. Method-based routing
 - 2.6. Customizing responses
 - 2.7. Project structure and organization
 - 2.8. HTML templating and inheritance
 - 2.9. Serving static files
 - 2.10. The `http.Handler` interface
- **3. Configuration and error handling**
 - 3.1. Managing configuration settings
 - 3.2. Structured logging
 - 3.3. Dependency injection
 - 3.4. Centralized error handling
 - 3.5. Isolating the application routes
- **4. Database-driven responses**
 - 4.1. Setting up MySQL
 - 4.2. Installing a database driver
 - 4.3. Modules and reproducible builds
 - 4.4. Creating a database connection pool
 - 4.5. Designing a database model
 - 4.6. Executing SQL statements
 - 4.7. Single-record SQL queries
 - 4.8. Multiple-record SQL queries
 - 4.9. Transactions and other details
- **5. Dynamic HTML templates**
 - 5.1. Displaying dynamic data

- 5.2. Template actions and functions
- 5.3. Caching templates
- 5.4. Catching runtime errors
- 5.5. Common dynamic data
- 5.6. Custom template functions
- **6. Middleware**
 - 6.1. How middleware works
 - 6.2. Setting common headers
 - 6.3. Request logging
 - 6.4. Panic recovery
 - 6.5. Composable middleware chains
- **7. Processing forms**
 - 7.1. Setting up an HTML form
 - 7.2. Parsing form data
 - 7.3. Validating form data
 - 7.4. Displaying errors and repopulating fields
 - 7.5. Creating validation helpers
 - 7.6. Automatic form parsing
- **8. Stateful HTTP**
 - 8.1. Choosing a session manager
 - 8.2. Setting up the session manager
 - 8.3. Working with session data
- **9. Server and security improvements**
 - 9.1. The `http.Server` struct
 - 9.2. The server error log
 - 9.3. Generating a self-signed TLS certificate
 - 9.4. Running a HTTPS server
 - 9.5. Configuring HTTPS settings
 - 9.6. Connection timeouts
- **10. User authentication**
 - 10.1. Routes setup
 - 10.2. Creating a users model
 - 10.3. User signup and password encryption
 - 10.4. User login

- 10.5. User logout
- 10.6. User authorization
- 10.7. CSRF protection
- **11. Using request context**
 - 11.1. How request context works
 - 11.2. Request context for authentication/authorization
- **12. File embedding**
 - 12.1. Embedding static files
 - 12.2. Embedding HTML templates
- **13. Testing**
 - 13.1. Unit testing and sub-tests
 - 13.2. Testing HTTP handlers and middleware
 - 13.3. End-to-end testing
 - 13.4. Customizing how tests run
 - 13.5. Mocking dependencies
 - 13.6. Testing HTML forms
 - 13.7. Integration testing
 - 13.8. Profiling test coverage
- **14. Conclusion**
- **15. Further reading and useful links**

Introduction

In this book we'll be building a web application called Snippetbox, which lets people paste and share snippets of text — a bit like [Pastebin](#) or GitHub's [Gists](#). Towards the end of the build it will look a bit like this:



Our application will start off super simple, with just one web page. Then with each chapter we'll build it up step-by-step until a user can save and view snippets via the app. This will take us through topics like how to structure a project, routing requests, working with a database, processing forms and displaying dynamic data safely.

Then later in the book we'll add user accounts, and restrict the application so that only registered users can create snippets. This will take us through more advanced topics like configuring a HTTPS server, session management, user authentication and middleware.

Conventions

Code blocks in this book are shown with a silver background, like the example below. If a code block is particularly long, any parts that aren't relevant may be replaced with an ellipsis. To make it easy to follow along, most code blocks also have a title bar at the top indicating the name of the file that the code is in. Like this:

```
File: hello.go

package main

... // Indicates that some existing code has been omitted.

func sayHello() {
    fmt.Println("Hello world!")
}
```

Terminal (command line) instructions are shown with a black background and start with a dollar symbol. These commands should work on any Unix-based operating system, including Mac OSX and Linux. Sample output is shown in silver beneath the command, like so:

```
$ echo "Hello world!"
Hello world!
```

If you're using Windows, you should replace the command with the DOS equivalent or carry out the action via the normal Windows GUI.

Please note that the dates and timestamps shown in screenshots and the example output from commands are illustrative only. They do not necessarily align with each other, or progress chronologically throughout the book.

Some chapters in this book end with an *additional information* section. These sections contain information that isn't relevant to our application build, but is still important (or sometimes, just interesting) to know about. If you're very new to Go, you might want to skip these parts and circle back to them later.

Hint: If you're following along with the application build I recommend using the HTML version of this book instead of the PDF or EPUB. The HTML version works in all browsers, and the proper formatting of code blocks is retained if you want to copy-and-paste code directly from the book.

When using the HTML version, you can also use the left and right arrow keys on your keyboard to navigate between chapters.

About the author

Hey, I'm Alex Edwards, a full-stack web developer and author. I live near Innsbruck, Austria.

I've been working with Go for over 10 years, building production applications for myself and commercial clients, and helping people all around the world improve their Go skills.

You can see more of my writing on [my blog](#) (where I publish detailed tutorials), some of my open-source work on [GitHub](#), and you can also follow me on [Instagram](#) and [Twitter](#).

Copyright and disclaimer

Let's Go: Learn to build professional web applications with Go. Copyright © 2024 Alex Edwards.

Last updated 2024-04-05 09:15:11 UTC. Version 2.22.1.

The Go gopher was designed by [Renee French](#) and is used under the Creative Commons 3.0 Attributions license. Cover gopher adapted from vectors by [Egon Elbre](#).

The information provided within this book is for general informational purposes only. While the author and publisher have made every effort to ensure the accuracy of the information within this book was correct at time of publication there are no representations or warranties, express or implied, about the completeness, accuracy, reliability, suitability or availability with respect to the information, products, services, or related graphics contained in this book for any purpose. Any use of this information is at your own risk.

Prerequisites

Background knowledge

This book is designed for people who are new to Go, but you'll probably find it more enjoyable if you have a general understanding of Go's syntax first. If you find yourself struggling with the syntax, the [Little Book of Go](#) by Karl Seguin is a fantastic tutorial, or if you want something more interactive I recommend running through the [Tour of Go](#).

I've also assumed that you've got a (very) basic understanding of HTML/CSS and SQL, and some familiarity with using your terminal (or command line for Windows users). If you've built a web application in any other language before — whether it's Ruby, Python, PHP or C# — then this book should be a good fit for you.

Go 1.22

The information in this book is correct for the [latest major release of Go](#) (version 1.22), and you should install this if you'd like to code-along with the application build.

If you've already got Go installed, you can check the version number from your terminal by using the `go version` command. The output should look similar to this:

```
$ go version  
go version go1.22.0 linux/amd64
```

If you need to upgrade your version of Go — or install Go from scratch — then please go ahead and do that now. Detailed instructions for different operating systems can be found here:

- [Removing an old version of Go](#)
- [Installing Go on Mac OS X](#)
- [Installing Go on Windows](#)
- [Installing Go on Linux](#)

Other software

There are a few other bits of software that you should make sure are available on your computer if you want to follow along fully. They are:

- The [curl](#) tool for working with HTTP requests and responses from your terminal. On MacOS and Linux machines it should be pre-installed or available in your software repositories. Otherwise, you can download the latest version [from here](#).
- A web browser with good developer tools. I'll be using [Firefox](#) in this book, but Chromium, Chrome or Microsoft Edge will work too.
- Your favorite text editor 😊

Foundations

Alright, let's get started! In this first section of the book we're going to lay the groundwork for our project and explain the main principles that you need to know for the rest of the application build.

You'll learn how to:

- Set up a project directory which follows the Go conventions.
- Start a web server and listen for incoming HTTP requests.
- Route requests to different handlers based on the request path and method.
- Use wildcard segments in your routing patterns.
- Send different HTTP responses, headers and status codes to users.
- Structure your project in a sensible and scalable way.
- Render HTML pages and use template inheritance to keep your HTML markup free of duplicate boilerplate code.
- Serve static files like images, CSS and JavaScript from your application.

Project setup and creating a module

Before we write any code, you'll need to create a `snippetbox` directory on your computer to act as the top-level 'home' for this project. All the Go code we write throughout the book will live in here, along with other project-specific assets like HTML templates and CSS files.

So, if you're following along, open your terminal and create a new project directory called `snippetbox` anywhere on your computer. I'm going to locate my project directory under `$HOME/code`, but you can choose a different location if you wish.

```
$ mkdir -p $HOME/code/snippetbox
```

Creating a module

The next thing you need to do is decide on a module path for your project.

If you're not already familiar with [Go modules](#), you can think of a module path as basically being a canonical name or *identifier* for your project.

You can pick [almost any string](#) as your module path, but the important thing to focus on is *uniqueness*. To avoid potential import conflicts with other people's projects or the standard library in the future, you want to pick a module path that is globally unique and unlikely to be used by anything else. In the Go community, a common convention is to base your module paths on a URL that you own.

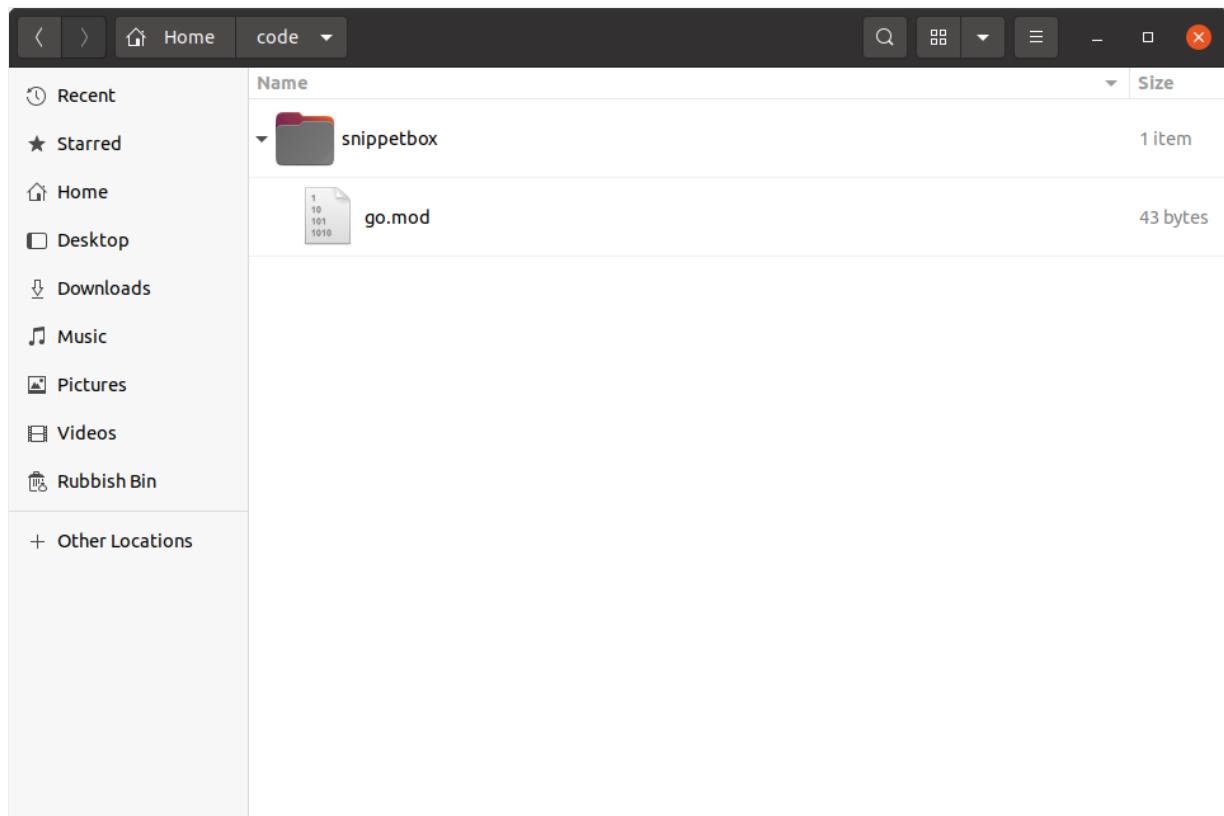
In my case, a clear, succinct and *unlikely-to-be-used-by-anything-else* module path for this project would be `snippetbox.alexewards.net`, and I'll use this throughout the rest of the book. If possible, you should swap this for something that's unique to you instead.

Once you've decided on a unique module path, the next thing you need to do is turn your project directory into a module.

Make sure that you're in the root of your project directory and then run the `go mod init` command — passing in your chosen module path as a parameter like so:

```
$ cd $HOME/code/snippetbox
$ go mod init snippetbox.alexewards.net
go: creating new go.mod: module snippetbox.alexewards.net
```

At this point your project directory should look a bit like the screenshot below. Notice the `go.mod` file which has been created?



At the moment there's not much going on in this file, and if you open it up in your text editor it should look like this (but preferably with your own unique module path instead):

```
File: go.mod
module snippetbox.alexewards.net
go 1.22.0
```

We'll talk about modules in more detail later in the book, but for now it's enough to know that when there is a valid `go.mod` file in the root of your project directory, your project *is a module*. Setting up your project as a module has a number of advantages — including making it much easier to manage third-party dependencies, [avoid supply-chain attacks](#), and ensure reproducible builds of your application in the future.

Hello world!

Before we continue, let's quickly check that everything is set up correctly. Go ahead and create a new `main.go` in your project directory containing the following code:

```
$ touch main.go
```

File: main.go

```
package main

import "fmt"

func main() {
    fmt.Println("Hello world!")
}
```

Save this file, then use the `go run .` command in your terminal to compile and execute the code in the current directory. All being well, you will see the following output:

```
$ go run .
Hello world!
```

Additional information

Module paths for downloadable packages

If you're creating a project which can be downloaded and used by other people and programs, then it's good practice for your module path to equal the location that the code can be downloaded from.

For instance, if your package is hosted at <https://github.com/foo/bar> then the module path for the project should be `github.com/foo/bar`.

Web application basics

Now that everything is set up correctly, let's make the first iteration of our web application. We'll begin with the three absolute essentials:

- The first thing we need is a handler. If you've previously built web applications using a MVC pattern, you can think of handlers as being a bit like controllers. They're responsible for executing your application logic and for writing HTTP response headers and bodies.
- The second component is a router (or servemux in Go terminology). This stores a mapping between the URL routing patterns for your application and the corresponding handlers. Usually you have one servemux for your application containing all your routes.
- The last thing we need is a web server. One of the great things about Go is that you can establish a web server and listen for incoming requests *as part of your application itself*. You don't need an external third-party server like Nginx, Apache or Caddy.

Let's put these components together in the `main.go` file to make a working application.

```
File: main.go

package main

import (
    "log"
    "net/http"
)

// Define a home handler function which writes a byte slice containing
// "Hello from Snippetbox" as the response body.
func home(w http.ResponseWriter, r *http.Request) {
    w.Write([]byte("Hello from Snippetbox"))
}

func main() {
    // Use the http.NewServeMux() function to initialize a new servemux, then
    // register the home function as the handler for the "/" URL pattern.
    mux := http.NewServeMux()
    mux.HandleFunc("/", home)

    // Print a log message to say that the server is starting.
    log.Println("starting server on :4000")

    // Use the http.ListenAndServe() function to start a new web server. We pass in
    // two parameters: the TCP network address to listen on (in this case ":4000")
    // and the servemux we just created. If http.ListenAndServe() returns an error
    // we use the log.Fatal() function to log the error message and exit. Note
    // that any error returned by http.ListenAndServe() is always non-nil.
    err := http.ListenAndServe(":4000", mux)
    log.Fatal(err)
}
```

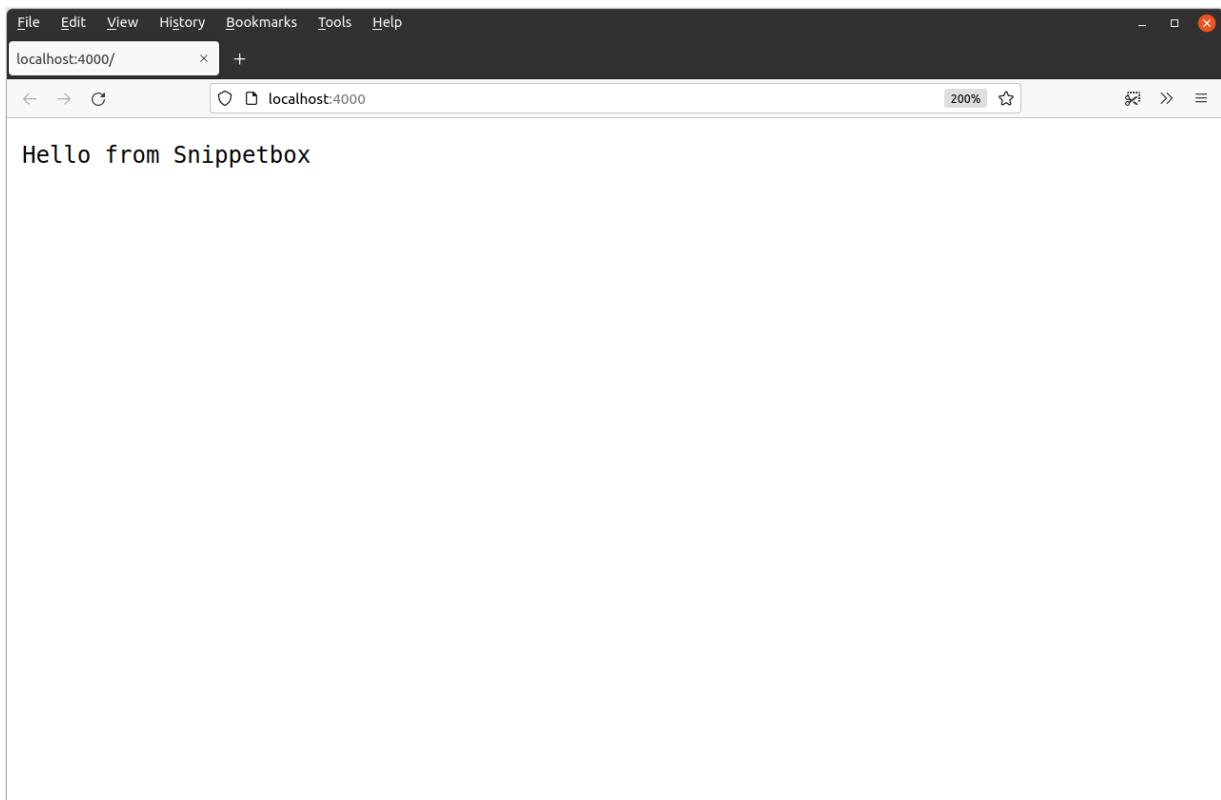
Note: The `home` handler function is just a regular Go function with two parameters. The `http.ResponseWriter` parameter provides methods for assembling a HTTP response and sending it to the user, and the `*http.Request` parameter is a pointer to a struct which holds information about the current request (like the HTTP method and the URL being requested). We'll talk more about these parameters and demonstrate how to use them as we progress through the book.

When you run this code, it will start a web server listening on port 4000 of your local machine. Each time the server receives a new HTTP request it will pass the request on to the servemux and — in turn — the servemux will check the URL path and dispatch the request to the matching handler.

Let's give this a whirl. Save your `main.go` file and then try running it from your terminal using the `go run` command.

```
$ cd $HOME/code/snippetbox
$ go run .
2024/03/18 11:29:23 starting server on :4000
```

While the server is running, open a web browser and try visiting <http://localhost:4000>. If everything has gone to plan you should see a page which looks a bit like this:



Important: Before we continue, I should explain that Go's servemux treats the route pattern `"/"` like a catch-all. So at the moment *all* HTTP requests to our server will be handled by the `home` function, regardless of their URL path. For instance, you can visit a different URL path like <http://localhost:4000/foo/bar> and you'll receive exactly the same response.

If you head back to your terminal window, you can stop the server by pressing `Ctrl+C` on your keyboard.

Additional information

Network addresses

The TCP network address that you pass to `http.ListenAndServe()` should be in the format `"host:port"`. If you omit the host (like we did with ":4000") then the server will listen on all

your computer's available network interfaces. Generally, you only need to specify a host in the address if your computer has multiple network interfaces and you want to listen on just one of them.

In other Go projects or documentation you might sometimes see network addresses written using named ports like "`:http`" or "`:http-alt`" instead of a number. If you use a named port then the `http.ListenAndServe()` function will attempt to look up the relevant port number from your `/etc/services` file when starting the server, returning an error if a match can't be found.

Using go run

During development the `go run` command is a convenient way to try out your code. It's essentially a shortcut that compiles your code, creates an executable binary in your `/tmp` directory, and then runs this binary in one step.

It accepts either a space-separated list of `.go` files, the path to a specific package (where the `.` character represents your current directory), or the full module path. For our application at the moment, the three following commands are all equivalent:

```
$ go run .
$ go run main.go
$ go run snippetbox.alexewards.net
```

Routing requests

Having a web application with just one route isn't very exciting... or useful! Let's add a couple more routes so that the application starts to shape up like this:

Route pattern	Handler	Action
/	home	Display the home page
/snippet/view	snippetView	Display a specific snippet
/snippet/create	snippetCreate	Display a form for creating a new snippet

Reopen the `main.go` file and update it as follows:

```
File: main.go

package main

import (
    "log"
    "net/http"
)

func home(w http.ResponseWriter, r *http.Request) {
    w.Write([]byte("Hello from Snippetbox"))
}

// Add a snippetView handler function.
func snippetView(w http.ResponseWriter, r *http.Request) {
    w.Write([]byte("Display a specific snippet..."))
}

// Add a snippetCreate handler function.
func snippetCreate(w http.ResponseWriter, r *http.Request) {
    w.Write([]byte("Display a form for creating a new snippet..."))
}

func main() {
    // Register the two new handler functions and corresponding route patterns with
    // the servemux, in exactly the same way that we did before.
    mux := http.NewServeMux()
    mux.HandleFunc("/", home)
    mux.HandleFunc("/snippet/view", snippetView)
    mux.HandleFunc("/snippet/create", snippetCreate)

    log.Println("starting server on :4000")

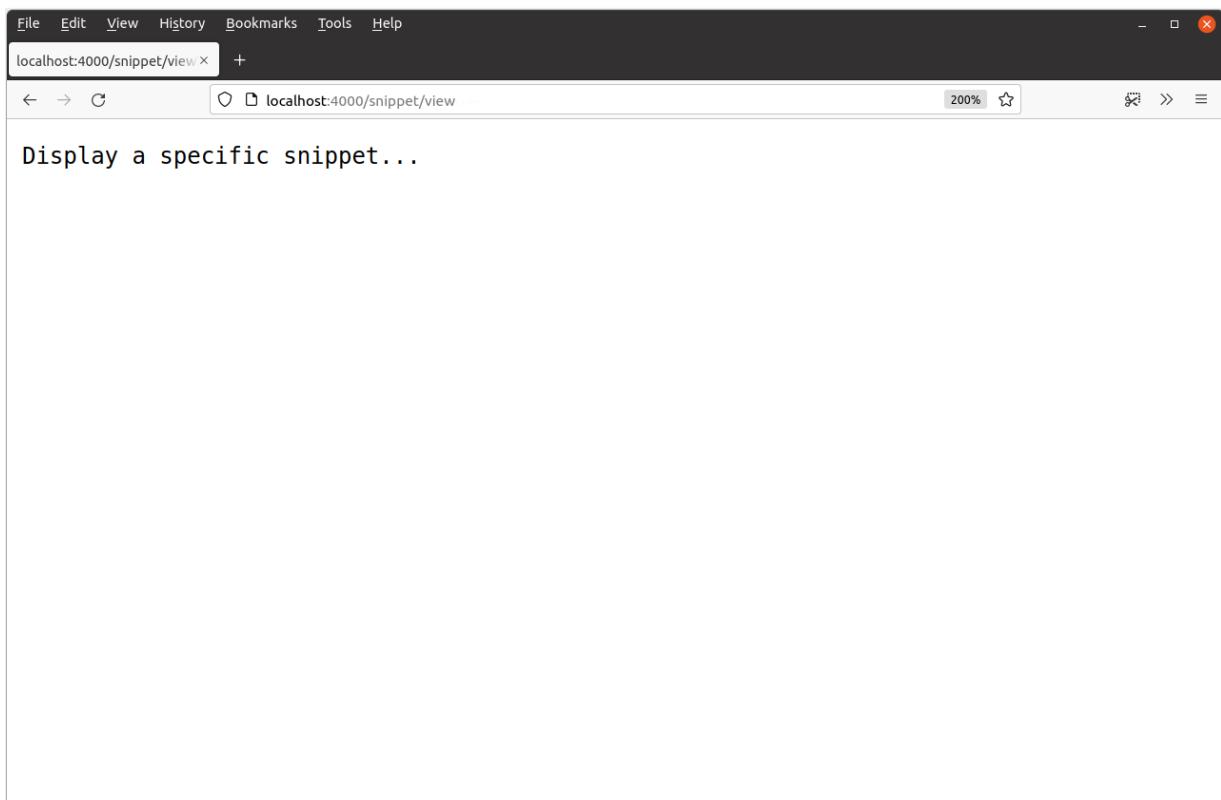
    err := http.ListenAndServe(":4000", mux)
    log.Fatal(err)
}
```

Make sure these changes are saved and then restart the web application:

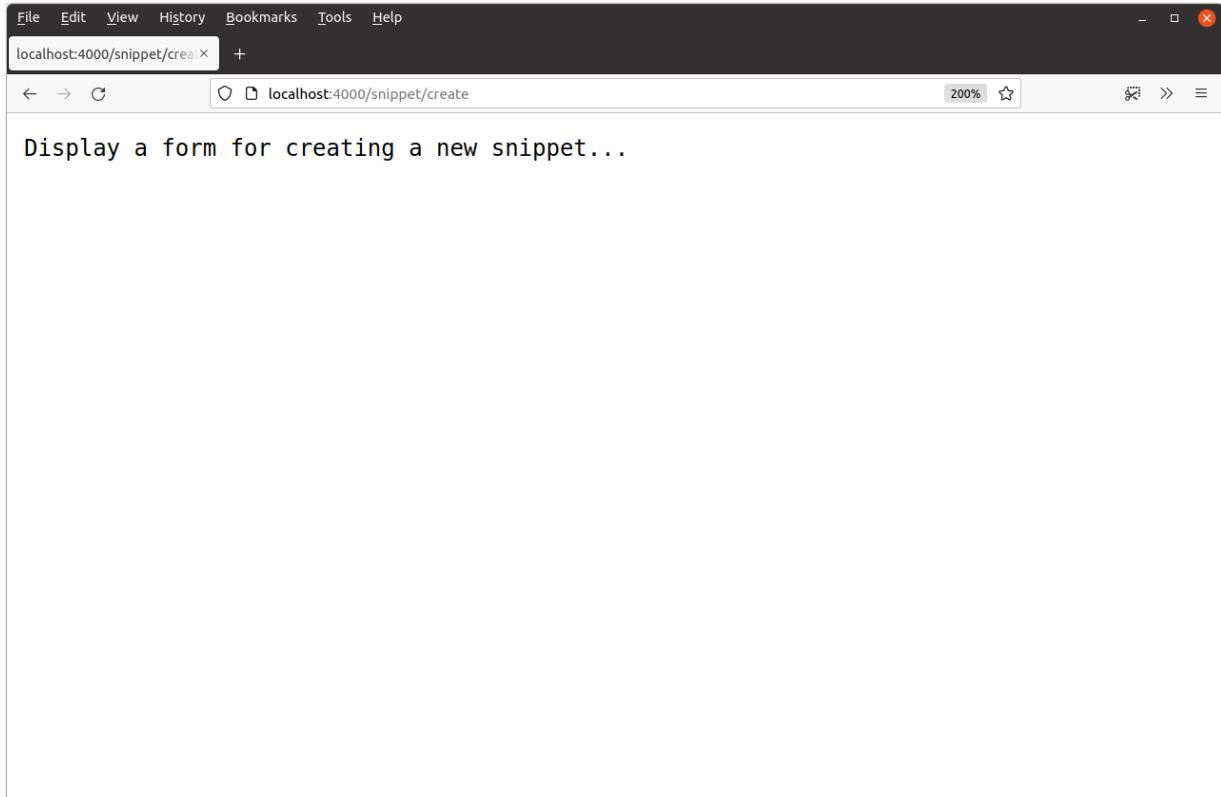
```
$ cd $HOME/code/snippetbox  
$ go run .  
2024/03/18 11:29:23 starting server on :4000
```

If you visit the following links in your web browser you should now get the appropriate response for each route:

- <http://localhost:4000/snippet/view>



- <http://localhost:4000/snippet/create>



Trailing slashes in route patterns

Now that the two new routes are up and running, let's talk a bit of theory.

It's important to know that Go's servemux has different matching rules depending on whether a route pattern ends with a trailing slash or not.

Our two new route patterns — "`/snippet/view`" and "`/snippet/create`" — don't end in a trailing slash. When a pattern doesn't have a trailing slash, it will only be matched (and the corresponding handler called) when the request URL path exactly matches the pattern in full.

When a route pattern ends with a trailing slash — like "`/"` or "`/static/`" — it is known as a *subtree path pattern*. Subtree path patterns are matched (and the corresponding handler called) whenever the *start* of a request URL path matches the subtree path. If it helps your understanding, you can think of subtree paths as acting a bit like they have a wildcard at the end, like "`/**`" or "`/static/**`".

This helps explain why the "`/"` route pattern acts like a catch-all. The pattern essentially means *match a single slash, followed by anything (or nothing at all)*.

Restricting subtree paths

To prevent subtree path patterns from acting like they have a wildcard at the end, you can append the special character sequence `{$}` to the end of the pattern — like `"/{$}"` or `"/static/{$}"`.

So if you have the route pattern `"/{$}"`, it effectively means *match a single slash, followed by nothing else*. It will only match requests where the URL path is exactly `/`.

Let's use this in our application to stop our `home` handler acting as a catch all, like so:

```
File: main.go

package main

...

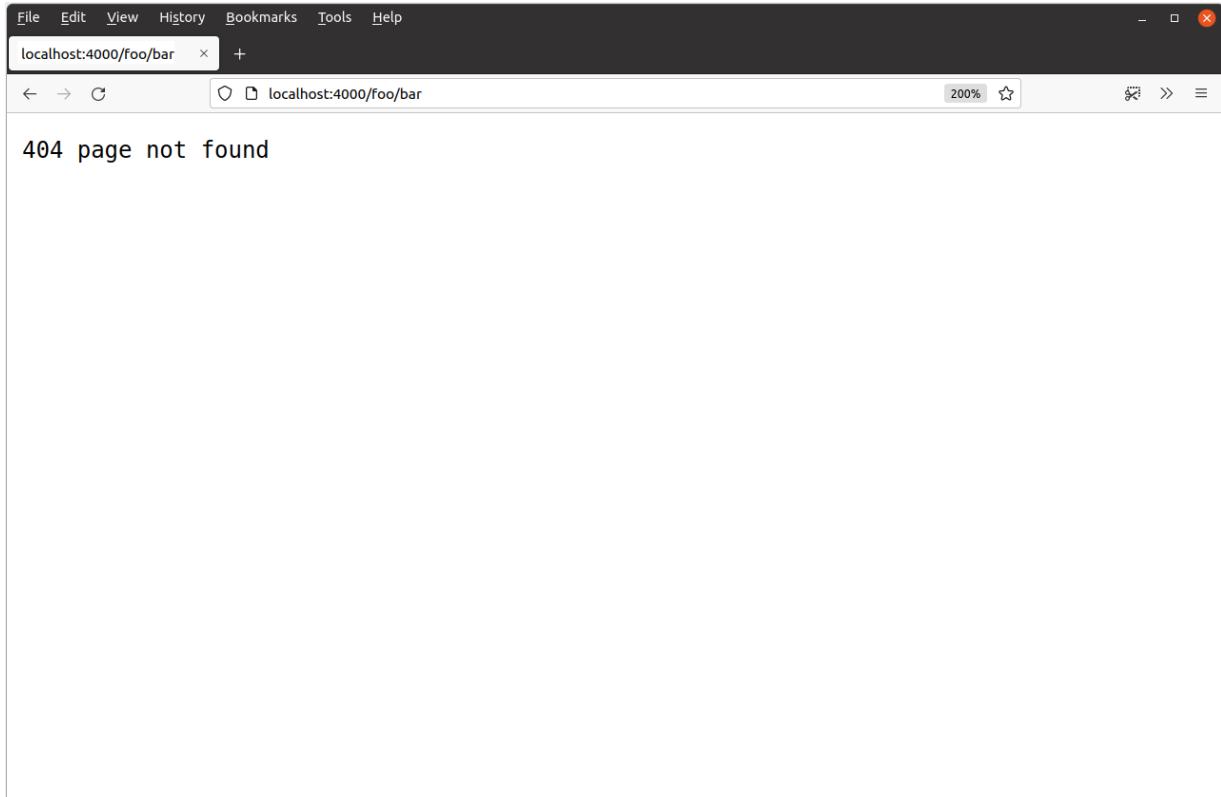
func main() {
    mux := http.NewServeMux()
    mux.HandleFunc("/", home) // Restrict this route to exact matches on / only.
    mux.HandleFunc("/snippet/view", snippetView)
    mux.HandleFunc("/snippet/create", snippetCreate)

    log.Println("starting server on :4000")

    err := http.ListenAndServe(":4000", mux)
    log.Fatal(err)
}
```

Note: It's only permitted to use `{$}` at the end of subtree path patterns (i.e. patterns that end with a trailing slash). Route patterns that don't have a trailing slash require a match on the entire request path anyway, so it doesn't make sense to include `{$}` at the end and trying to do so will result in a runtime panic.

Once you've made that change, restart the server and make a request for an unregistered URL path like `http://localhost:4000/foo/bar`. You should now get a `404` response which looks a bit like this:



Additional information

Additional servemux features

There are a couple of other servemux features worth pointing out:

- Request URL paths are automatically sanitized. If the request path contains any `.` or `..` elements or repeated slashes, the user will automatically be redirected to an equivalent clean URL. For example, if a user makes a request to `/foo/bar/..//baz` they will automatically be sent a [301 Permanent Redirect](#) to `/foo/baz` instead.
- If a subtree path has been registered and a request is received for that subtree path *without* a trailing slash, then the user will automatically be sent a [301 Permanent Redirect](#) to the subtree path with the slash added. For example, if you have registered the subtree path `/foo/`, then any request to `/foo` will be redirected to `/foo/`.

Host name matching

It's possible to include host names in your route patterns. This can be useful when you want

to redirect all HTTP requests to a canonical URL, or if your application is acting as the back end for multiple sites or services. For example:

```
mux := http.NewServeMux()
mux.HandleFunc("foo.example.org/", fooHandler)
mux.HandleFunc("bar.example.org/", barHandler)
mux.HandleFunc("/baz", bazHandler)
```

When it comes to pattern matching, any host-specific patterns will be checked first and if there is a match the request will be dispatched to the corresponding handler. Only when there *isn't* a host-specific match found will the non-host specific patterns also be checked.

The default servemux

If you've been working with Go for a while you might have come across the `http.Handle()` and `http.HandleFunc()` functions. These allow you to register routes *without* explicitly declaring a servemux, like this:

```
func main() {
    http.HandleFunc("/", home)
    http.HandleFunc("/snippet/view", snippetView)
    http.HandleFunc("/snippet/create", snippetCreate)

    log.Println("starting server on :4000")

    err := http.ListenAndServe(":4000", nil)
    log.Fatal(err)
}
```

Behind the scenes, these functions register their routes with something called the default servemux. This is just a regular servemux like we've already been using, but which is initialized automatically by Go and stored in the `http.DefaultServeMux` global variable.

If you pass `nil` as the second argument to `http.ListenAndServe()`, the server will use `http.DefaultServeMux` for routing.

Although this approach can make your code slightly shorter, I don't recommend it for two reasons:

- It's less explicit and feels more 'magic' than declaring and using your own locally-scoped servemux.
- Because `http.DefaultServeMux` is a global variable in the standard library, it means *any* Go code in your project can access it and potentially register a route. If you have a large project codebase (especially one that is being worked on by many people), that can

make it harder to ensure all route declarations for your application are easily discoverable in one central place.

It also means that any *third-party packages that your application imports* can register routes with `http.DefaultServeMux` too. If one of those third-party packages is compromised, they could potentially use `http.DefaultServeMux` to expose a malicious handler to the web. It's simple to avoid this risk by just not using `http.DefaultServeMux`.

So, for the sake of clarity, maintainability and security, it's generally a good idea to avoid `http.DefaultServeMux` and the corresponding helper functions. Use your own locally-scoped servemux instead, like we have been doing in this project so far.

Wildcard route patterns

It's also possible to define route patterns that contain *wildcard segments*. You can use these to create more flexible routing rules, and also to pass variables to your Go application via a request URL. If you've built web applications using frameworks in other languages before, the concepts in this chapter will probably feel familiar to you.

Let's step away from our application build for a moment to explain how it works.

Wildcard segments in a route pattern are denoted by an wildcard *identifier* inside `{}` brackets. Like this:

```
mux.HandleFunc("/products/{category}/item/{itemID}", exampleHandler)
```

In this example, the route pattern contains two wildcard segments. The first segment has the identifier `category` and the second has the identifier `itemID`.

The matching rules for route patterns containing wildcard segments are the same as we saw in the previous chapter, with the additional rule that the request path can contain *any* non-empty value for the wildcard segments. So, for example, the following requests would all match the route we defined above:

```
/products/hammocks/item/sku123456789  
/products/seasonal-plants/item/pdt-1234-wxyz  
/products/experimental_foods/item/quantum%20bananas
```

Important: When defining a route pattern, each path segment (the bit between forward slash characters) can only contain one wildcard and the wildcard needs to fill the *whole* path segment. Patterns like `"/products/c_{category}"`, `/date/{y}-{m}-{d}` or `/{slug}.html` are not valid.

Inside your handler, you can retrieve the corresponding value for a wildcard segment using its identifier and the `r.PathValue()` method. For example:

```

func exampleHandler(w http.ResponseWriter, r *http.Request) {
    category := r.PathValue("category")
    itemID := r.PathValue("itemID")

    ...
}

```

The `r.PathValue()` method always returns a `string` value, and it's important to remember that this can be *any value* that the user includes in the URL — so you should validate or sanity check the value before doing anything important with it.

Using wildcard segments in practice

OK, let's get back to our application and update it to include a new `{id}` wildcard segment in the `/snippet/view` route, so that our routes look like this:

Route pattern	Handler	Action
<code>/\${\$}</code>	<code>home</code>	Display the home page
<code>/snippet/view/{id}</code>	<code>snippetView</code>	Display a specific snippet
<code>/snippet/create</code>	<code>snippetCreate</code>	Display a form for creating a new snippet

Open up your `main.go` file, and make this change like so:

```

File: main.go

package main

...

func main() {
    mux := http.NewServeMux()
    mux.HandleFunc("/${$}", home)
    mux.HandleFunc("/snippet/view/{id}", snippetView) // Add the {id} wildcard segment
    mux.HandleFunc("/snippet/create", snippetCreate)

    log.Println("starting server on :4000")

    err := http.ListenAndServe(":4000", mux)
    log.Fatal(err)
}

```

Now let's head to our `snippetView` handler and update it to retrieve the `id` value from the request URL path. Later in the book, we'll use this `id` value to select a specific snippet from a database, but for now we'll just echo the `id` back to the user as part of the HTTP

response.

Because the `id` value is untrusted user input, we should validate it to make sure it's sane and sensible before we use it. For the purpose of our application we want to check that the `id` value contains a positive integer, which we can do by trying to convert the string value to an integer with the `strconv.Atoi()` function and then checking that the value is greater than zero.

Here's how:

```
File: main.go

package main

import (
    "fmt" // New import
    "log"
    "net/http"
    "strconv" // New import
)

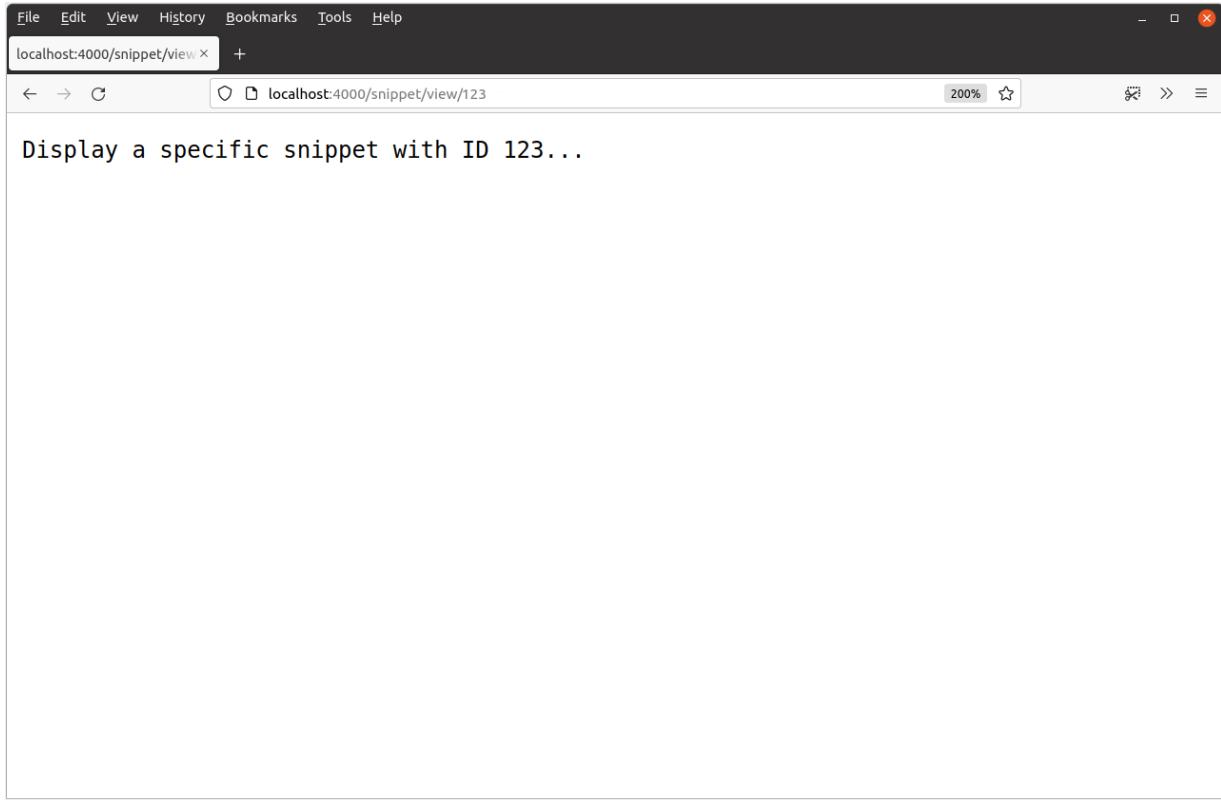
...

func snippetView(w http.ResponseWriter, r *http.Request) {
    // Extract the value of the id wildcard from the request using r.PathValue()
    // and try to convert it to an integer using the strconv.Atoi() function. If
    // it can't be converted to an integer, or the value is less than 1, we
    // return a 404 page not found response.
    id, err := strconv.Atoi(r.PathValue("id"))
    if err != nil || id < 1 {
        http.NotFound(w, r)
        return
    }

    // Use the fmt.Sprintf() function to interpolate the id value with a
    // message, then write it as the HTTP response.
    msg := fmt.Sprintf("Display a specific snippet with ID %d...", id)
    w.Write([]byte(msg))
}

...
```

Save your changes, restart the application, then open your browser and try visiting a URL like <http://localhost:4000/snippet/view/123>. You should see a response containing the echoed `id` wildcard value from the request URL, similar to this:



You might also like to try visiting some URLs which have invalid values for the `id` wildcard, or no wildcard value at all. For instance:

- <http://localhost:4000/snippet/view/>
- <http://localhost:4000/snippet/view/-1>
- <http://localhost:4000/snippet/view/foo>

For all these requests you should get a `404 page not found` response.

Additional information

Precedence and conflicts

When defining route patterns with wildcard segments, it's possible that some of your patterns will 'overlap'. For example, if you define routes with the patterns "`/post/edit`" and "`/post/{id}`" they overlap because an incoming HTTP request with the path `/post/edit` is a valid match for *both* patterns.

When route patterns overlap, Go's servemux needs to decide which pattern takes precedent so it can dispatch the request to the appropriate handler.

The rule for this is very neat and succinct: *the most specific route pattern wins*. Formally, Go defines a pattern as more specific than another if it matches only a subset of requests that the other pattern matches.

Continuing with the example above, the route pattern "`/post/edit`" only matches requests with the exact path `/post/edit`, whereas the pattern "`/post/{id}`" matches requests with the path `/post/edit`, `/post/123`, `/post/abc` and many more. Therefore "`/post/edit`" is the more specific route pattern and will take precedent.

While we're on this topic, there are a few other things worth mentioning:

- A nice side-effect of the *most specific pattern wins* rule is that you can register patterns in any order and it won't change how the servemux behaves.
- There is a potential edge case where you have two overlapping route patterns but neither one is obviously more specific than the other. For example, the patterns "`/post/new/{id}`" and "`/post/{author}/latest`" overlap because they both match the request path `/post/new/latest`, but it's not clear which one should take precedence. In this scenario, Go's servemux considers the patterns to *conflict*, and will panic at runtime when initializing the routes.
- Just because Go's servemux supports overlapping routes, it doesn't mean that you should use them! Having overlapping route patterns increases the risk of bugs and unintended behavior in your application, and if you have the freedom to design the URL structure for your application it's generally good practice to keep overlaps to a minimum or avoid them completely.

Subtree path patterns with wildcards

It's important to understand that the routing rules we described in the previous chapter still apply, even when you're using wildcard segments. In particular, if your route pattern ends in a trailing slash and has no `[$]` at the end, then it is treated as a *subtree path pattern* and it only requires the *start* of a request URL path to match.

So, if you have a subtree path pattern like "`/user/{id}/`" in your routes (note the trailing slash), this pattern will match requests like `/user/1/`, `/user/2/a`, `/user/2/a/b/c` and so on.

Again, if you don't want that behavior, stick a `[$]` at the end — like "`/user/{id}/{$}`".

Remainder wildcards

Wildcards in route patterns normally match a single, non-empty, segment of the request

path only. But there is one special case.

If a route pattern ends with a wildcard, and this final wildcard identifier ends in `...`, then the wildcard will match any and all remaining segments of a request path.

For example, if you declare a route pattern like `"/post/{path...}"` it will match requests like `/post/a`, `/post/a/b`, `/post/a/b/c` and so on — very much like a subtree path pattern does. But the difference is that you can access the entire wildcard part via the `r.PathValue()` method in your handlers. In this example, you could get the wildcard value for `{path...}` by calling `r.PathValue("path")`.

Method-based routing

Next, let's follow HTTP good practice and restrict our application so that it only responds to requests with an appropriate [HTTP method](#).

As we progress through our application build, our `home`, `snippetView` and `snippetCreate` handlers will merely retrieve information and display pages to the user, so it makes sense that these handlers should be restricted to acting on `GET` requests.

To restrict a route to a specific HTTP method, you can prefix the route pattern with the necessary HTTP method when declaring it, like so:

```
File: main.go

package main

...

func main() {
    mux := http.NewServeMux()
    // Prefix the route patterns with the required HTTP method (for now, we will
    // restrict all three routes to acting on GET requests).
    mux.HandleFunc("GET /${}", home)
    mux.HandleFunc("GET /snippet/view/{id}", snippetView)
    mux.HandleFunc("GET /snippet/create", snippetCreate)

    log.Println("starting server on :4000")

    err := http.ListenAndServe(":4000", mux)
    log.Fatal(err)
}
```

Note: The HTTP methods in route patterns are case sensitive and should always be written in uppercase, followed by *a single space character*. You can only include one HTTP method in each route pattern.

It's also worth mentioning that when you register a route pattern which uses the `GET` method, it will match both `GET` and `HEAD` requests. All other methods (like `POST`, `PUT` and `DELETE`) require an exact match.

Let's test out this change by using curl to make some requests to our application. If you're following along, start by making a regular `GET` request to `http://localhost:4000/`, like so:

```
$ curl -i localhost:4000/
HTTP/1.1 200 OK
Date: Wed, 18 Mar 2024 11:29:23 GMT
Content-Length: 21
Content-Type: text/plain; charset=utf-8

Hello from Snippetbox
```

The response here looks good. We can see that our route still works, and we get back a `200 OK` status and a `Hello from Snippetbox` response body, just like before.

You can also go ahead and try making a `HEAD` request for the same URL. You should see that this also works correctly, returning *just the HTTP response headers*, and no response body.

```
$ curl --head localhost:4000/
HTTP/1.1 200 OK
Date: Wed, 18 Mar 2024 11:29:23 GMT
Content-Length: 21
Content-Type: text/plain; charset=utf-8
```

In contrast, let's try making a `POST` request to `http://localhost:4000/`. The `POST` method isn't supported for this route, so you should get an error response similar to this:

```
$ curl -i -d "" localhost:4000/
HTTP/1.1 405 Method Not Allowed
Allow: GET, HEAD
Content-Type: text/plain; charset=utf-8
X-Content-Type-Options: nosniff
Date: Wed, 18 Mar 2024 11:29:23 GMT
Content-Length: 19

Method Not Allowed*
```

Note: The `curl -d` flag is used to declare any HTTP `POST` data that you want to include in the request body. In the command above we use `-d ""`, which means that the request body will be empty, but the request will still be sent with the HTTP method `POST` (rather than the default method of `GET`).

So that's looking really good. We can see that Go's servemux has automatically sent a `405 Method Not Allowed` response for us, including an `Allow` header which lists the HTTP methods that *are* supported for the request URL.

Adding a POST-only route and handler

Let's also add a new `snippetCreatePost` handler to our codebase, which we'll use later on to create and save a new snippet in a database. Because creating and saving a snippet is a non-idempotent action that will change the state of our server, we want make sure that this handler acts on `POST` requests only.

All in all, we'll want our fourth handler and route to look like this:

Route pattern	Handler	Action
<code>GET /{\$}</code>	<code>home</code>	Display the home page
<code>GET /snippet/view/{id}</code>	<code>snippetView</code>	Display a specific snippet
<code>GET /snippet/create</code>	<code>snippetCreate</code>	Display a form for creating a new snippet
<code>POST /snippet/create</code>	<code>snippetCreatePost</code>	Save a new snippet

Let's go ahead and add the necessary code to our `main.go` file, like so:

```
File: main.go

package main

...

// Add a snippetCreatePost handler function.
func snippetCreatePost(w http.ResponseWriter, r *http.Request) {
    w.Write([]byte("Save a new snippet..."))
}

func main() {
    mux := http.NewServeMux()
    mux.HandleFunc("GET /{$}", home)
    mux.HandleFunc("GET /snippet/view/{id}", snippetView)
    mux.HandleFunc("GET /snippet/create", snippetCreate)
    // Create the new route, which is restricted to POST requests only.
    mux.HandleFunc("POST /snippet/create", snippetCreatePost)

    log.Println("starting server on :4000")

    err := http.ListenAndServe(":4000", mux)
    log.Fatal(err)
}
```

Notice that it's totally OK to declare two (or more) separate routes that have different HTTP methods but otherwise have the same pattern, like we are doing here with "`GET /snippet/create`" and "`POST /snippet/create`".

If you restart the application and try making some requests with the URL path `/snippet/create`, you should now see different responses depending on the request

method that you use.

```
$ curl -i localhost:4000/snippet/create
HTTP/1.1 200 OK
Date: Wed, 18 Mar 2024 11:29:23 GMT
Content-Length: 50
Content-Type: text/plain; charset=utf-8

Display a form for creating a new snippet...

$ curl -i -d "" localhost:4000/snippet/create
HTTP/1.1 200 OK
Date: Wed, 18 Mar 2024 11:29:23 GMT
Content-Length: 21
Content-Type: text/plain; charset=utf-8

Save a new snippet...

$ curl -i -X DELETE localhost:4000/snippet/create
HTTP/1.1 405 Method Not Allowed
Allow: GET, HEAD, POST
Content-Type: text/plain; charset=utf-8
X-Content-Type-Options: nosniff
Date: Wed, 18 Mar 2024 11:29:23 GMT
Content-Length: 19

Method Not Allowed
```

Additional information

Method precedence

The *most specific pattern wins* rule also applies if you have route patterns that overlap because of a HTTP method.

It's important to be aware that a route pattern which doesn't include a method — like `/article/{id}` — will match incoming HTTP requests with *any method*. In contrast, a route like `POST /article/{id}` will only match requests which have the method `POST`. So if you declare the overlapping routes `/article/{id}` and `POST /article/{id}` in your application, then the `POST /article/{id}` route will take precedence.

Handler naming

I'd also like to emphasize that there is no right or wrong way to name your handlers in Go.

In this project, we'll follow a convention of postfixing the names of any handlers that deal with `POST` requests with the word 'Post'. Like so:

Route pattern	Handler	Action
GET /snippet/create	snippetCreate	Display a form for creating a new snippet
POST /snippet/create	snippetCreatePost	Create a new snippet

But in your own work it's not necessary to follow this pattern. For example, you could prefix handler names with the words 'get' and 'post' instead, like this:

Route pattern	Handler	Action
GET /snippet/create	getSnippetCreate	Display a form for creating a new snippet
POST /snippet/create	postSnippetCreate	Create a new snippet

Or even give the handlers completely different names. For example:

Route pattern	Handler	Action
GET /snippet/create	newSnippetForm	Display a form for creating a new snippet
POST /snippet/create	createSnippet	Create a new snippet

Basically, you have the freedom in Go to choose a naming convention for your handlers that works for you and fits with your brain.

Third-party routers

The wildcard and method-based routing functionality that we've been using in the past two chapters is very new to Go — it only became part of the standard library in Go 1.22. While this is a very welcome addition to the language and a big improvement, you might find that there are still times where the standard library routing functionality doesn't provide everything that you need.

For example, the following things are not currently supported:

- Sending custom `404 Not Found` and `405 Method Not Allowed` responses to the user (although there is an [open proposal](#) regarding this).
- Using regular expressions in your route patterns or wildcards.
- Matching multiple HTTP methods in a single route declaration.
- Automatic support for `OPTIONS` requests.

- Routing requests to handlers based on unusual things, like HTTP request headers.

If you need these features in your application, you'll need to use a third-party router package. The ones that I recommend are [httprouter](#), [chi](#), [flow](#) and [gorilla/mux](#), and you can find a comparison of them and guidance about which one to use in [this blog post](#).

Customizing responses

By default, every response that your handlers send has the [HTTP status code 200 OK](#) (which indicates to the user that their request was received and processed successfully), plus three automatic system-generated headers: a [Date](#) header, and the [Content-Length](#) and [Content-Type](#) of the response body. For example:

```
$ curl -i localhost:4000/
HTTP/1.1 200 OK
Date: Wed, 18 Mar 2024 11:29:23 GMT
Content-Length: 21
Content-Type: text/plain; charset=utf-8

Hello from Snippetbox
```

In this chapter we'll dive in to how to customize the response headers your handlers send, and also look at a couple of other ways that you can send plain text responses to users.

HTTP status codes

First of all, let's update our [snippetCreatePost](#) handler so that it sends a [201 Created](#) status code rather than [200 OK](#). To do this, you can use the [w.WriteHeader\(\)](#) method in your handlers like so:

```
File: main.go

package main

...

func snippetCreatePost(w http.ResponseWriter, r *http.Request) {
    // Use the w.WriteHeader() method to send a 201 status code.
    w.WriteHeader(201)

    // Then w.Write() method to write the response body as normal.
    w.Write([]byte("Save a new snippet..."))
}
```

(Yes, this is a bit silly because the handler isn't actually creating anything yet! But it nicely illustrates the pattern to set a custom status code.)

Although this change looks straightforward, there are a couple of nuances I should explain:

- It's only possible to call `w.WriteHeader()` once per response, and after the status code has been written it can't be changed. If you try to call `w.WriteHeader()` a second time Go will log a warning message.
- If you don't call `w.WriteHeader()` explicitly, then the first call to `w.Write()` will automatically send a `200` status code to the user. So, if you want to send a non-200 status code, you must call `w.WriteHeader()` before any call to `w.Write()`.

Restart the server, then use curl to make a `POST` request to

`http://localhost:4000/snippet/create` again. You should see that the HTTP response now has a `201 Created` status code similar to this:

```
$ curl -i -d "" http://localhost:4000/snippet/create
HTTP/1.1 201 Created
Date: Wed, 18 Mar 2024 11:29:23 GMT
Content-Length: 21
Content-Type: text/plain; charset=utf-8

Save a new snippet...
```

Status code constants

The `net/http` package provides [constants for HTTP status codes](#), which we can use instead of writing the status code number ourselves. Using these constants is good practice because it helps prevent mistakes due to typos, and it can also help make your code clearer and self-documenting — especially when dealing with less-commonly-used status codes.

Let's update our `snippetCreatePost` handler to use the constant `http.StatusCreated` instead of the integer `201`, like so:

```
File: main.go

package main

...

func snippetCreatePost(w http.ResponseWriter, r *http.Request) {
    w.WriteHeader(http.StatusCreated)

    w.Write([]byte("Save a new snippet..."))
}

...
```

Customizing headers

You can also customize the HTTP headers sent to a user by changing the *response header map*. Probably the most common thing you'll want to do is include an additional header in the map, which you can do using the `w.Header().Add()` method.

To demonstrate this, let's add a `Server: Go` header to the response that our `home` handler sends. If you're following along, go ahead and update the handler code like so:

```
File: main.go

package main

...

func home(w http.ResponseWriter, r *http.Request) {
    // Use the Header().Add() method to add a 'Server: Go' header to the
    // response header map. The first parameter is the header name, and
    // the second parameter is the header value.
    w.Header().Add("Server", "Go")

    w.Write([]byte("Hello from Snippetbox"))
}

...
```

Important: You must make sure that your response header map contains all the headers you want *before* you call `w.WriteHeader()` or `w.Write()`. Any changes you make to the response header map after calling `w.WriteHeader()` or `w.Write()` will have no effect on the headers that the user receives.

Let's try this out by using curl to make another request to `http://localhost:4000/`. This time you should see that the response now includes a new `Server: Go` header, like so:

```
$ curl -i http://localhost:4000
HTTP/1.1 200 OK
Server: Go
Date: Wed, 18 Mar 2024 11:29:23 GMT
Content-Length: 21
Content-Type: text/plain; charset=utf-8

Hello from Snippetbox
```

Writing response bodies

So far in this book we've been using `w.Write()` to send specific HTTP response bodies to a user. And while this is the simplest and most fundamental way to send a response, in practice it's far more common to pass your `http.ResponseWriter` value to *another function* that writes the response for you.

In fact, there are a lot of functions that you can use to write a response!

The key thing to understand is this... *because the `http.ResponseWriter` value in your handlers has a `Write()` method, it satisfies the `io.Writer` interface.*

If you're new to Go, then the [concept of interfaces](#) can be a bit confusing and I don't want to get too hung up on it right now. But at a practical level, it means that any functions where you see an `io.Writer` parameter, you can pass in your `http.ResponseWriter` value and whatever is being written will subsequently be sent as the body of the HTTP response.

That means you can use standard library functions like `io.WriteString()` and the `fmt.Fprint*()` family (all of which accept an `io.Writer` parameter) to write plain-text response bodies too.

```
// Instead of this...
w.Write([]byte("Hello world"))

// You can do this...
io.WriteString(w, "Hello world")
fmt.Fprint(w, "Hello world")
```

Let's leverage this, and update the code in our `snippetView` handler to use the `fmt.Fprintf()` function. This will allow us to interpolate the wildcard `id` value in our response body message *and* write the response in a single line, like so:

```
File: main.go

package main

...

func snippetView(w http.ResponseWriter, r *http.Request) {
    id, err := strconv.Atoi(r.PathValue("id"))
    if err != nil || id < 1 {
        http.NotFound(w, r)
        return
    }

    fmt.Fprintf(w, "Display a specific snippet with ID %d...", id)
}
```

Additional information

Content sniffing

In order to automatically set the `Content-Type` header, Go *content sniffs* the response body with the `http.DetectContentType()` function. If this function can't guess the content type, Go will fall back to setting the header `Content-Type: application/octet-stream` instead.

The `http.DetectContentType()` function generally works quite well, but a common gotcha for web developers is that it can't distinguish JSON from plain text. So, by default, JSON responses will be sent with a `Content-Type: text/plain; charset=utf-8` header. You can prevent this from happening by setting the correct header manually in your handler like so:

```
w.Header().Set("Content-Type", "application/json")
w.Write([]byte(`{"name": "Alex"}`))
```

Manipulating the header map

In this chapter we used `w.Header().Add()` to add a new header to the response header map. But there are also `Set()`, `Del()`, `Get()` and `Values()` methods that you can use to manipulate and read from the header map too.

```
// Set a new cache-control header. If an existing "Cache-Control" header exists
// it will be overwritten.
w.Header().Set("Cache-Control", "public, max-age=31536000")

// In contrast, the Add() method appends a new "Cache-Control" header and can
// be called multiple times.
w.Header().Add("Cache-Control", "public")
w.Header().Add("Cache-Control", "max-age=31536000")

// Delete all values for the "Cache-Control" header.
w.Header().Del("Cache-Control")

// Retrieve the first value for the "Cache-Control" header.
w.Header().Get("Cache-Control")

// Retrieve a slice of all values for the "Cache-Control" header.
w.Header().Values("Cache-Control")
```

Header canonicalization

When you're using the `Set()`, `Add()`, `Del()`, `Get()` and `Values()` methods on the header map, the header name will always be canonicalized using the `textproto.CanonicalMIMEHeaderKey()` function. This converts the first letter and any letter

following a hyphen to upper case, and the rest of the letters to lowercase. This has the practical implication that when calling these methods the header name is *case-insensitive*.

If you need to avoid this canonicalization behavior, you can edit the underlying header map directly. It has the type `map[string][]string` behind the scenes. For example:

```
w.Header()["X-XSS-Protection"] = []string{"1; mode=block"}
```

Note: If a HTTP/2 connection is being used, Go will *always* automatically convert the header names and values to lowercase for you when writing the response, as per [the HTTP/2 specifications](#).

Project structure and organization

Before we add any more code to our `main.go` file, it's a good time to think how to organize and structure this project.

It's important to explain upfront that there's no single right — or even recommended — way to structure web applications in Go. And that's both good and bad. It means that you have freedom and flexibility over how you organize your code, but it's also easy to get stuck down a rabbit-hole of uncertainty when trying to decide what the best structure should be.

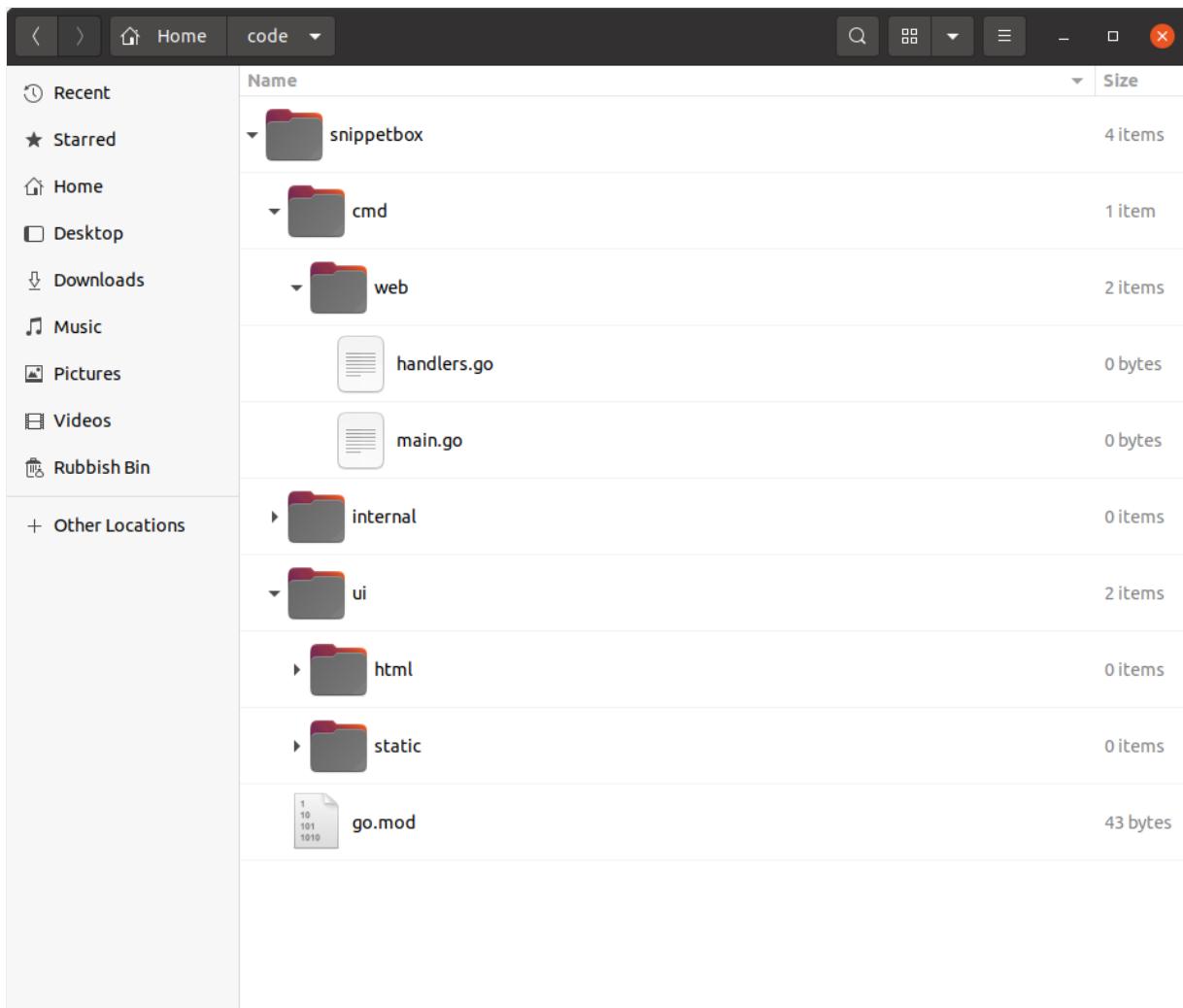
As you gain experience with Go, you'll get a feel for which patterns work well for you in different situations. But as a starting point, the best advice I can give you is *don't over-complicate things*. Try hard to add structure and complexity only when it's demonstrably needed.

For this project, we'll implement an outline structure that follows a [popular and tried-and-tested](#) approach. It's a solid starting point, and you should be able to reuse the general structure in a wide variety of projects.

If you're following along, make sure that you're in the root of your project repository and run the following commands:

```
$ cd $HOME/code/snippetbox
$ rm main.go
$ mkdir -p cmd/web internal ui/html ui/static
$ touch cmd/web/main.go
$ touch cmd/web/handlers.go
```

The structure of your project repository should now look like this:



Let's take a moment to discuss what each of these directories will be used for.

- The **cmd** directory will contain the *application-specific* code for the executable applications in the project. For now our project will have just one executable application — the web application — which will live under the **cmd/web** directory.
- The **internal** directory will contain the ancillary *non-application-specific* code used in the project. We'll use it to hold potentially reusable code like validation helpers and the SQL database models for the project.
- The **ui** directory will contain the *user-interface assets* used by the web application. Specifically, the **ui/html** directory will contain HTML templates, and the **ui/static** directory will contain static files (like CSS and images).

So why are we using this structure?

There are two big benefits:

1. It gives a clean separation between Go and non-Go assets. All the Go code we write will

live exclusively under the `cmd` and `internal` directories, leaving the project root free to hold non-Go assets like UI files, makefiles and module definitions (including our `go.mod` file).

2. It scales really nicely if you want to add another executable application to your project. For example, you might want to add a CLI (Command Line Interface) to automate some administrative tasks in the future. With this structure, you could create this CLI application under `cmd/cli` and it will be able to import and reuse all the code you've written under the `internal` directory.

Refactoring your existing code

Let's quickly port the code we've already written to this new structure.

```
File: cmd/web/main.go

package main

import (
    "log"
    "net/http"
)

func main() {
    mux := http.NewServeMux()
    mux.HandleFunc("GET /{}", home)
    mux.HandleFunc("GET /snippet/view/{id}", snippetView)
    mux.HandleFunc("GET /snippet/create", snippetCreate)
    mux.HandleFunc("POST /snippet/create", snippetCreatePost)

    log.Println("starting server on :4000")

    err := http.ListenAndServe(":4000", mux)
    log.Fatal(err)
}
```

```
File: cmd/web/handlers.go
```

```
package main

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

func home(w http.ResponseWriter, r *http.Request) {
    w.Header().Add("Server", "Go")
    w.Write([]byte("Hello from Snippetbox"))
}

func snippetView(w http.ResponseWriter, r *http.Request) {
    id, err := strconv.Atoi(r.PathValue("id"))
    if err != nil || id < 1 {
        http.NotFound(w, r)
        return
    }

    fmt.Fprintf(w, "Display a specific snippet with ID %d...", id)
}

func snippetCreate(w http.ResponseWriter, r *http.Request) {
    w.Write([]byte("Display a form for creating a new snippet..."))
}

func snippetCreatePost(w http.ResponseWriter, r *http.Request) {
    w.WriteHeader(http.StatusCreated)
    w.Write([]byte("Save a new snippet..."))
}
```

So now our web application consists of multiple `.go` files under the `cmd/web` directory. To run these, we can use the `go run` command like so:

```
$ cd $HOME/code/snippetbox
$ go run ./cmd/web
2024/03/18 11:29:23 starting server on :4000
```

Additional information

The internal directory

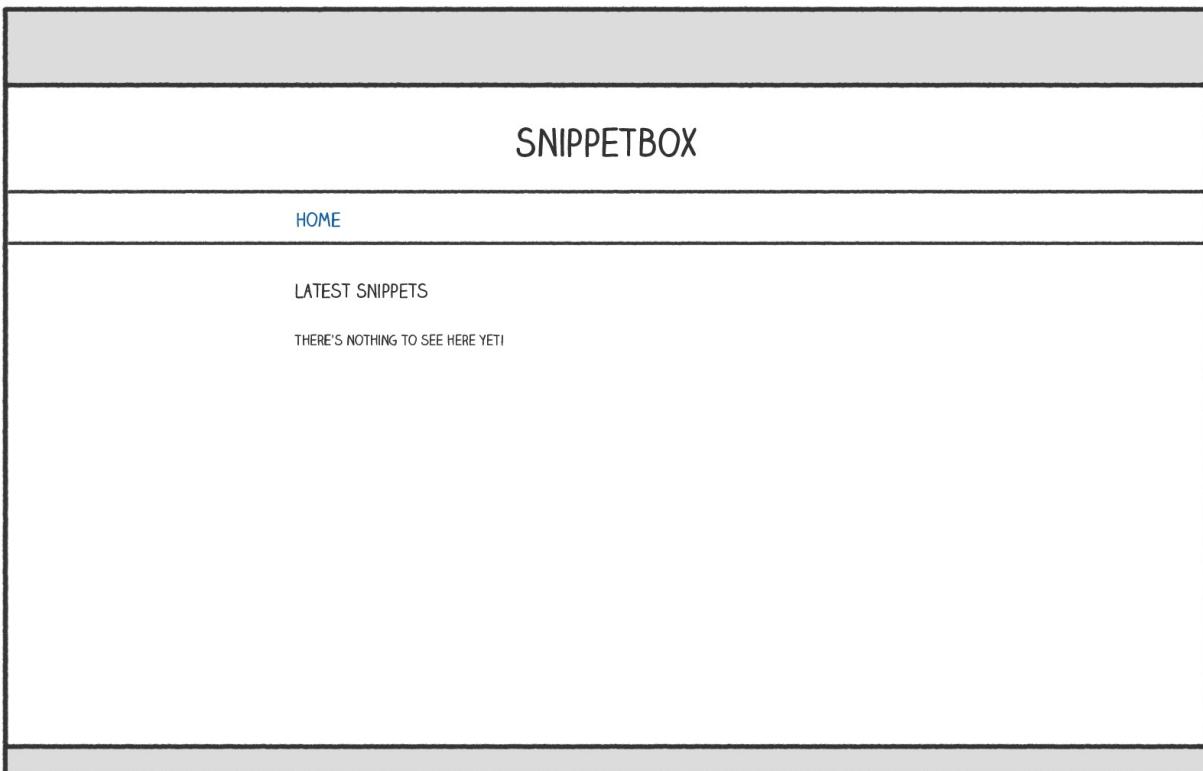
It's important to point out that the directory name `internal` carries a special meaning and behavior in Go: any packages which live under this directory can only be imported by code *inside the parent of the `internal` directory*. In our case, this means that any packages which live in `internal` can only be imported by code inside our `snippetbox` project directory.

Or, looking at it the other way, this means that any packages under `internal` *cannot be imported by code outside of our project.*

This is useful because it prevents other codebases from importing and relying on the (potentially unversioned and unsupported) packages in our `internal` directory — even if the project code is publicly available somewhere like GitHub.

HTML templating and inheritance

Let's inject a bit of life into the project and develop a proper home page for our Snippetbox web application. Over the next couple of chapters we'll work towards creating a page which looks like this:



Let's start by creating a template file at `ui/html/pages/home.tpl` to contain the HTML content for the home page. Like so:

```
$ mkdir ui/html/pages  
$ touch ui/html/pages/home.tpl
```

And add the following HTML markup:

```
File: ui/html/pages/home.tpl
```

```
<!doctype html>
<html lang='en'>
  <head>
    <meta charset='utf-8'>
    <title>Home - Snippetbox</title>
  </head>
  <body>
    <header>
      <h1><a href='/'>Snippetbox</a></h1>
    </header>
    <main>
      <h2>Latest Snippets</h2>
      <p>There's nothing to see here yet!</p>
    </main>
    <footer>Powered by <a href='https://golang.org/'>Go</a></footer>
  </body>
</html>
```

Note: The `.tpl` extension doesn't convey any special meaning or behavior here. I've only chosen this extension because it's a nice way of making it clear that the file contains a Go template when you're browsing a list of files. But, if you want, you could use the extension `.html` instead (which may make your text editor recognize the file as HTML for the purpose of syntax highlighting or autocompletion) — or you could even use a 'double extension' like `.tpl.html`. The choice is yours, but we'll stick to using `.tpl` for our templates throughout this book.

Now that we've created a template file containing the HTML markup for the home page, the next question is *how do we get our `home` handler to render it?*

For this we need to use Go's `html/template` package, which provides a family of functions for safely parsing and rendering HTML templates. We can use the functions in this package to parse the template file and then execute the template.

I'll demonstrate. Open the `cmd/web/handlers.go` file and add the following code:

```
File: cmd/web/handlers.go
```

```
package main

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

func home(w http.ResponseWriter, r *http.Request) {
    w.Header().Add("Server", "Go")

    // Use the template.ParseFiles() function to read the template file into a
    // template set. If there's an error, we log the detailed error message, use
    // the http.Error() function to send an Internal Server Error response to the
    // user, and then return from the handler so no subsequent code is executed.
    ts, err := template.ParseFiles("./ui/html/pages/home tmpl")
    if err != nil {
        log.Println(err.Error())
        http.Error(w, "Internal Server Error", http.StatusInternalServerError)
        return
    }

    // Then we use the Execute() method on the template set to write the
    // template content as the response body. The last parameter to Execute()
    // represents any dynamic data that we want to pass in, which for now we'll
    // leave as nil.
    err = ts.Execute(w, nil)
    if err != nil {
        log.Println(err.Error())
        http.Error(w, "Internal Server Error", http.StatusInternalServerError)
    }
}

...
```

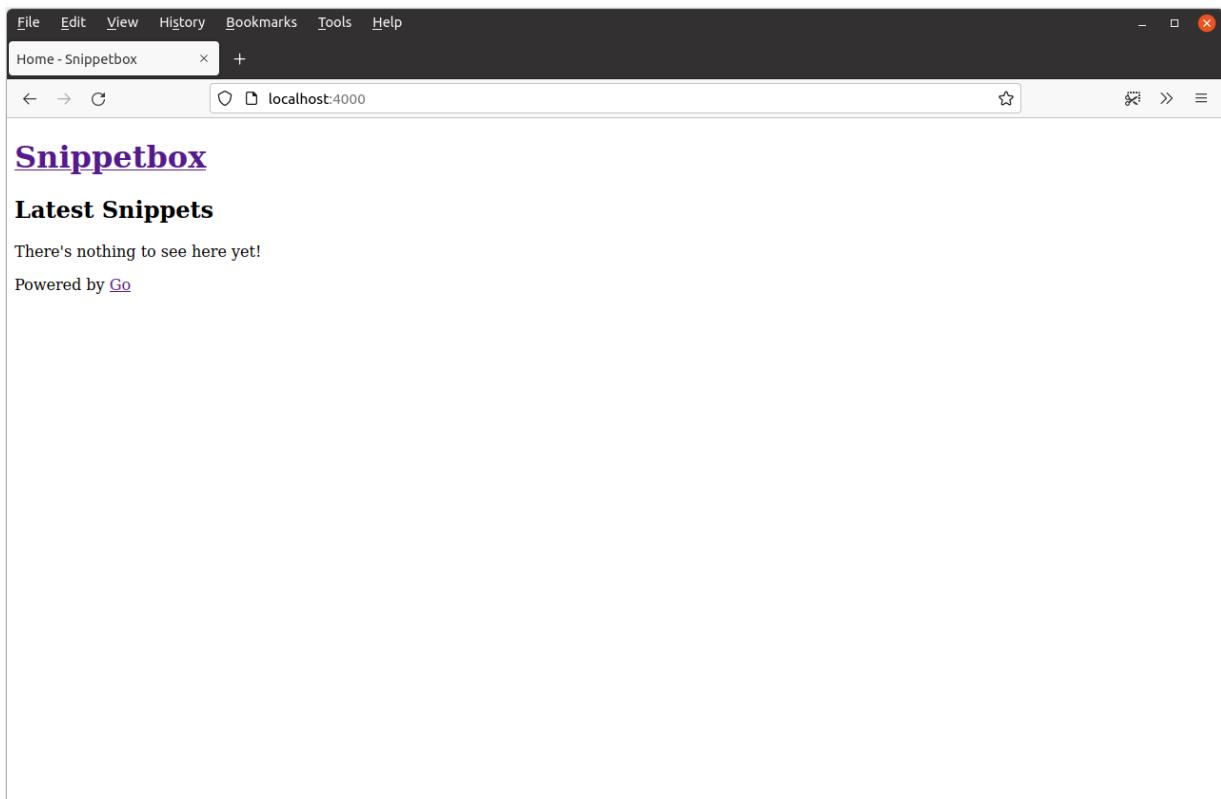
There are a couple of important things about this code to point out:

- The file path that you pass to the `template.ParseFiles()` function must either be relative to your current working directory, or an absolute path. In the code above I've made the path relative to the root of the project directory.
- If either the `template.ParseFiles()` or `ts.Execute()` functions return an error, we log the detailed error message and then use the `http.Error()` function to send a response to the user. `http.Error()` is a lightweight helper function which sends a plain text error message and a specific HTTP status code to the user (in our code we send the message "`Internal Server Error`" and the status code `500`, represented by the constant `http.StatusInternalServerError`). Effectively, this means that if there is an error, the user will see the message `Internal Server Error` in their browser, but the detailed error message will be recorded in the application log messages.

So, with that said, make sure you're in the root of your project directory and restart the application:

```
$ cd $HOME/code/snippetbox
$ go run ./cmd/web
2024/03/18 11:29:23 starting server on :4000
```

Then open <http://localhost:4000> in your web browser. You should find that the HTML homepage is shaping up nicely.



Template composition

As we add more pages to our web application, there will be some shared, boilerplate, HTML markup that we want to include on every page — like the header, navigation and metadata inside the `<head>` HTML element.

To prevent duplication and save typing, it's a good idea to create a *base* (or *master*) template which contains this shared content, which we can then compose with the page-specific markup for the individual pages.

Go ahead and create a new `ui/html/base tmpl` file...

```
$ touch ui/html/base.tpl
```

And add the following markup (which we want to appear in every page):

```
File: ui/html/base.tpl

{{define "base"}}
<!doctype html>
<html lang='en'>
  <head>
    <meta charset='utf-8'>
    <title>{{template "title" .}} - Snippetbox</title>
  </head>
  <body>
    <header>
      <h1><a href='/'>Snippetbox</a></h1>
    </header>
    <main>
      {{template "main" .}}
    </main>
    <footer>Powered by <a href='https://golang.org/'>Go</a></footer>
  </body>
</html>
{{end}}
```

Hopefully this feels familiar if you've used templating in other languages before. It's essentially just regular HTML with some extra *actions* in double curly braces.

We use the `{{define "base"}}...{{end}}` action as a wrapper to define a distinct named template called `base`, which contains the content we want to appear on every page.

Inside this we use the `{{template "title" .}}` and `{{template "main" .}}` actions to denote that we want to invoke other named templates (called `title` and `main`) at a particular location in the HTML.

Note: If you're wondering, the dot at the end of the `{{template "title" .}}` action represents any dynamic data that you want to pass to the invoked template. We'll talk more about this later in the book.

Now let's go back to the `ui/html/pages/home.tpl` file and update it to define `title` and `main` named templates containing the specific content for the home page.

```
File: ui/html/pages/home tmpl

{{define "title"}}Home{{end}}

{{define "main"}}
<h2>Latest Snippets</h2>
<p>There's nothing to see here yet!</p>
{{end}}
```

Once that's done, the next step is to update the code in your `home` handler so that it parses *both* template files, like so:

```
File: cmd/web/handlers.go

package main

...

func home(w http.ResponseWriter, r *http.Request) {
    w.Header().Add("Server", "Go")

    // Initialize a slice containing the paths to the two files. It's important
    // to note that the file containing our base template must be the *first*
    // file in the slice.
    files := []string{
        "./ui/html/base tmpl",
        "./ui/html/pages/home tmpl",
    }

    // Use the template.ParseFiles() function to read the files and store the
    // templates in a template set. Notice that we use ... to pass the contents
    // of the files slice as variadic arguments.
    ts, err := template.ParseFiles(files...)
    if err != nil {
        log.Println(err.Error())
        http.Error(w, "Internal Server Error", http.StatusInternalServerError)
        return
    }

    // Use the ExecuteTemplate() method to write the content of the "base"
    // template as the response body.
    err = ts.ExecuteTemplate(w, "base", nil)
    if err != nil {
        log.Println(err.Error())
        http.Error(w, "Internal Server Error", http.StatusInternalServerError)
    }
}

...
```

So now, instead of containing HTML directly, our template set contains 3 named templates — `base`, `title` and `main`. We use the `ExecuteTemplate()` method to tell Go that we specifically want to respond using the content of the `base` template (which in turn invokes our `title` and `main` templates).

Feel free to restart the server and give this a try. You should find that it renders the same output as before (although there will be some extra whitespace in the HTML source where the actions are).

Embedding partials

For some applications you might want to break out certain bits of HTML into partials that can be reused in different pages or layouts. To illustrate, let's create a partial containing the primary navigation bar for our web application.

Create a new `ui/html/partials/nav.tpl` file containing a named template called "`nav`", like so:

```
$ mkdir ui/html/partials  
$ touch ui/html/partials/nav.tpl
```

```
File: ui/html/partials/nav.tpl  
  
{{define "nav"}}  
<nav>  
  <a href='/'>Home</a>  
</nav>  
{{end}}
```

Then update the `base` template so that it invokes the navigation partial using the `{{template "nav" .}}` action:

```
File: ui/html/base.tmpl
```

```
{{define "base"}}
<!doctype html>
<html lang='en'>
  <head>
    <meta charset='utf-8'>
    <title>{{template "title" .}} - Snippetbox</title>
  </head>
  <body>
    <header>
      <h1><a href='/'>Snippetbox</a></h1>
    </header>
    <!-- Invoke the navigation template -->
    {{template "nav" .}}
    <main>
      {{template "main" .}}
    </main>
    <footer>Powered by <a href='https://golang.org/'>Go</a></footer>
  </body>
</html>
{{end}}
```

Finally, we need to update the `home` handler to include the new `ui/html/partials/nav.tmpl` file when parsing the template files:

```
File: cmd/web/handlers.go
```

```
package main

...

func home(w http.ResponseWriter, r *http.Request) {
    w.Header().Add("Server", "Go")

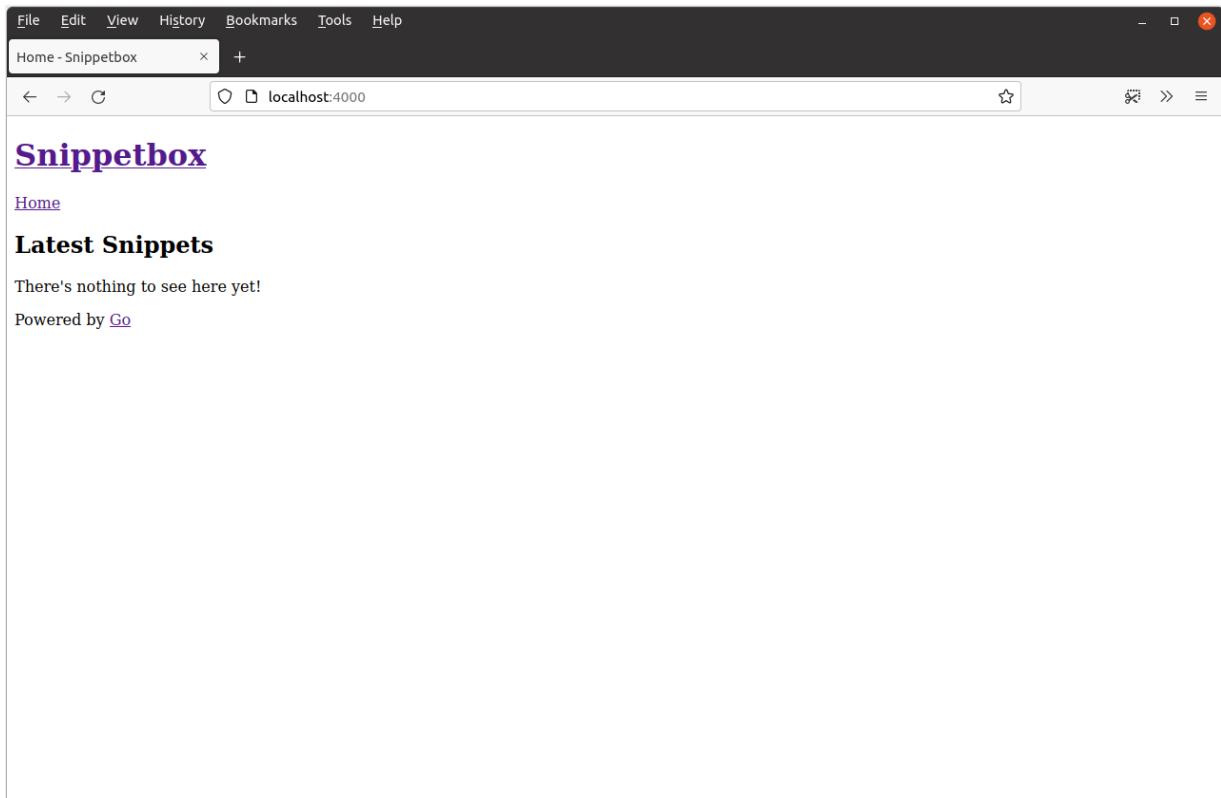
    // Include the navigation partial in the template files.
    files := []string{
        "./ui/html/base.tmpl",
        "./ui/html/partials/nav.tmpl",
        "./ui/html/pages/home.tmpl",
    }

    ts, err := template.ParseFiles(files...)
    if err != nil {
        log.Println(err.Error())
        http.Error(w, "Internal Server Error", http.StatusInternalServerError)
        return
    }

    err = ts.ExecuteTemplate(w, "base", nil)
    if err != nil {
        log.Println(err.Error())
        http.Error(w, "Internal Server Error", http.StatusInternalServerError)
    }
}

...
```

Once you restart the server, the `base` template should now invoke the `nav` template and your home page should look like this:



Additional information

The block action

In the code above we've used the `{{template}}` action to invoke one template from another. But Go also provides a `{{block}}...{{end}}` action which you can use instead. This acts like the `{{template}}` action, except it allows you to specify some default content if the template being invoked *doesn't exist in the current template set*.

In the context of a web application, this is useful when you want to provide some default content (such as a sidebar) which individual pages can override on a case-by-case basis if they need to.

Syntactically you use it like this:

```
 {{define "base"}}
  <h1>An example template</h1>
  {{block "sidebar" .}}
    <p>My default sidebar content</p>
  {{end}}
{{end}}
```

But — if you want — you don't *need* to include any default content between the `{{block}}` and `{{end}}` actions. In that case, the invoked template acts like it's 'optional'. If the template exists in the template set, then it will be rendered. But if it doesn't, then nothing will be displayed.

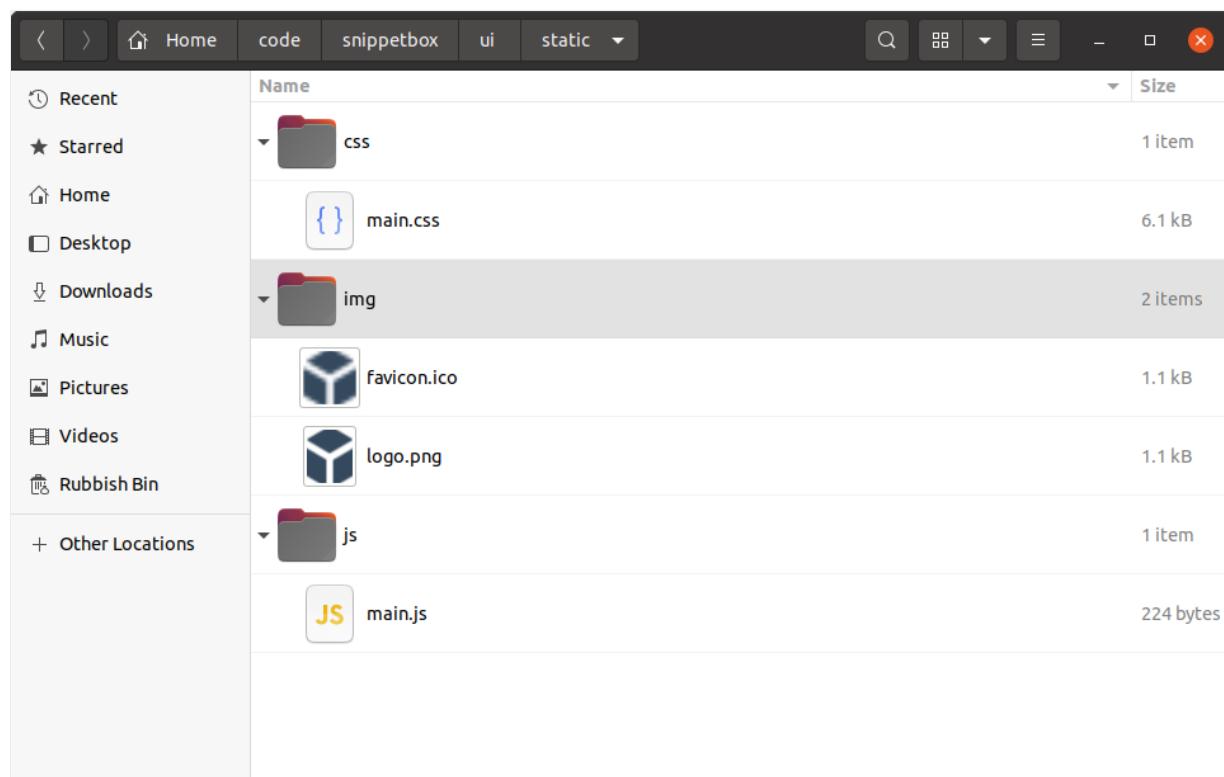
Serving static files

Now let's improve the look and feel of the home page by adding some static CSS and image files to our project, along with a tiny bit of JavaScript to highlight the active navigation item.

If you're following along, you can grab the necessary files and extract them into the `ui/static` folder that we made earlier with the following commands:

```
$ cd $HOME/code/snippetbox
$ curl https://www.alexewards.net/static/sb-v2.tar.gz | tar -xvz -C ./ui/static/
```

The contents of your `ui/static` directory should now look like this:



The `http.FileServer` handler

Go's `net/http` package ships with a built-in `http.FileServer` handler which you can use to serve files over HTTP from a specific directory. Let's add a new route to our application so that all `GET` requests which begin with `"/static/"` are handled using this, like so:

Route pattern	Handler	Action
GET /	home	Display the home page
GET /snippet/view/{id}	snippetView	Display a specific snippet
GET /snippet/create	snippetCreate	Display a form for creating a new snippet
POST /snippet/create	snippetCreatePost	Save a new snippet
GET /static/	http.FileServer	Serve a specific static file

Remember: The pattern "`GET /static/`" is a subtree path pattern, so it acts a bit like there is a wildcard at the end.

To create a new `http.FileServer` handler, we need to use the `http.FileServer()` function like this:

```
fileServer := http.FileServer(http.Dir("./ui/static/"))
```

When this handler receives a request for a file, it will remove the leading slash from the request URL path and then search the `./ui/static` directory for the corresponding file to send to the user.

So, for this to work correctly, we must strip the leading `"/static"` from the URL path *before* passing it to `http.FileServer`. Otherwise it will be looking for a file which doesn't exist and the user will receive a `404 page not found` response. Fortunately Go includes a `http.StripPrefix()` helper specifically for this task.

Open your `main.go` file and add the following code, so that the file ends up looking like this:

```
File: cmd/web/main.go
```

```
package main

import (
    "log"
    "net/http"
)

func main() {
    mux := http.NewServeMux()

    // Create a file server which serves files out of the "./ui/static" directory.
    // Note that the path given to the http.Dir function is relative to the project
    // directory root.
    fileServer := http.FileServer(http.Dir("./ui/static"))

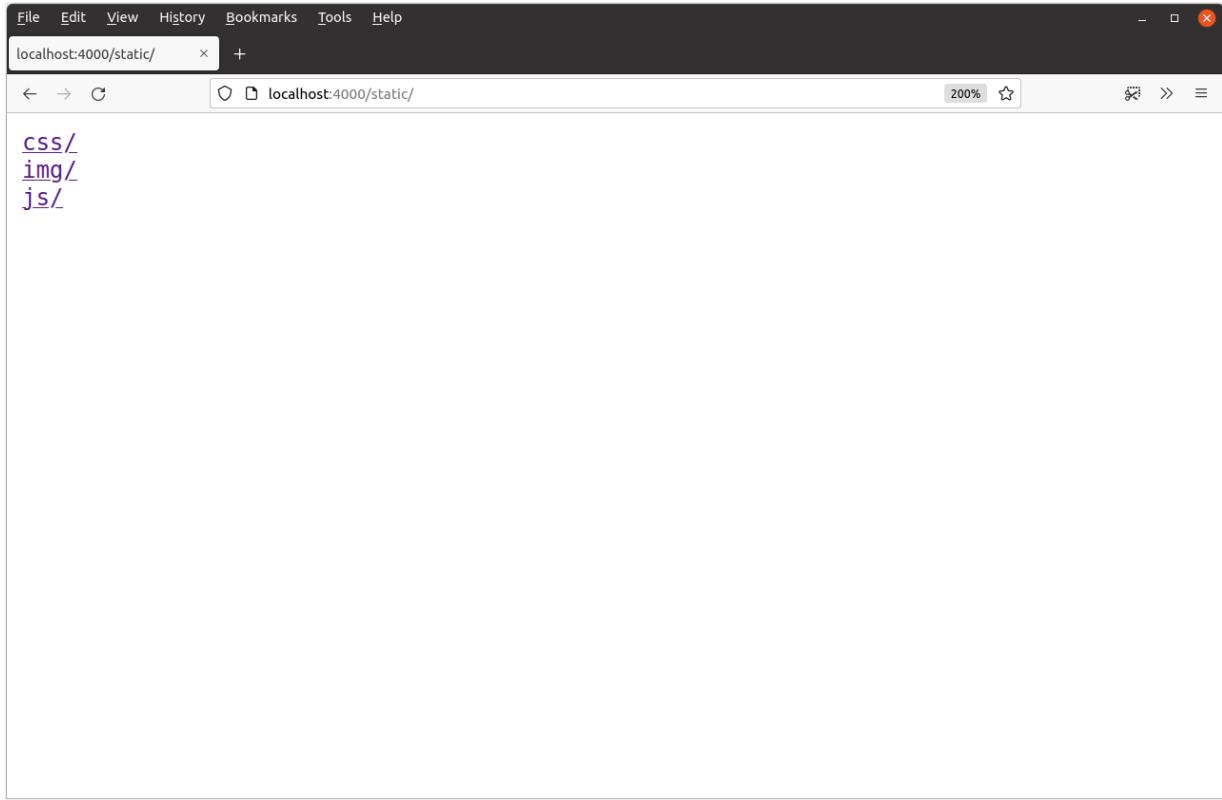
    // Use the mux.Handle() function to register the file server as the handler for
    // all URL paths that start with "/static/". For matching paths, we strip the
    // "/static" prefix before the request reaches the file server.
    mux.Handle("GET /static/", http.StripPrefix("/static", fileServer))

    // Register the other application routes as normal..
    mux.HandleFunc("GET /${}", home)
    mux.HandleFunc("GET /snippet/view/{id}", snippetView)
    mux.HandleFunc("GET /snippet/create", snippetCreate)
    mux.HandleFunc("POST /snippet/create", snippetCreatePost)

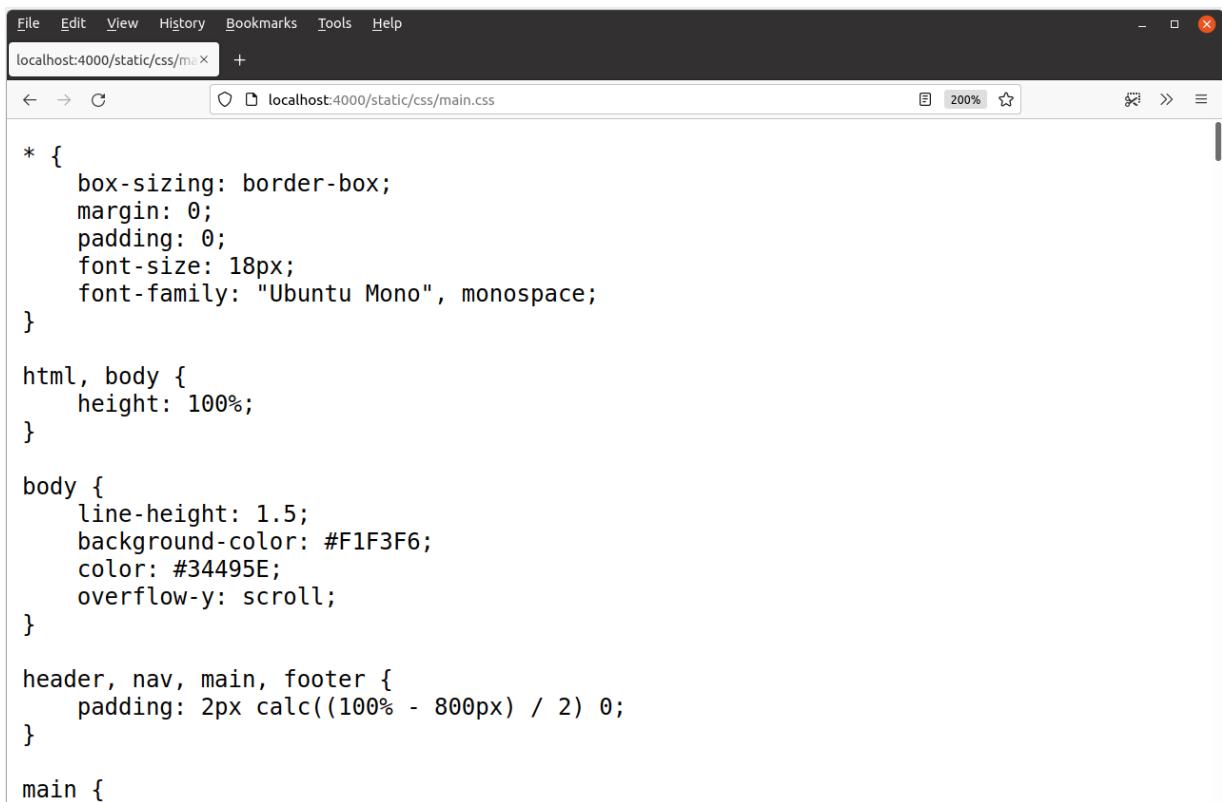
    log.Println("starting server on :4000")

    err := http.ListenAndServe(":4000", mux)
    log.Fatal(err)
}
```

Once that's complete, restart the application and open <http://localhost:4000/static/> in your browser. You should see a navigable directory listing of the `ui/static` folder which looks like this:



Feel free to have a play around and browse through the directory listing to view individual files. For example, if you navigate to <http://localhost:4000/static/css/main.css> you should see the CSS file appear in your browser like so:



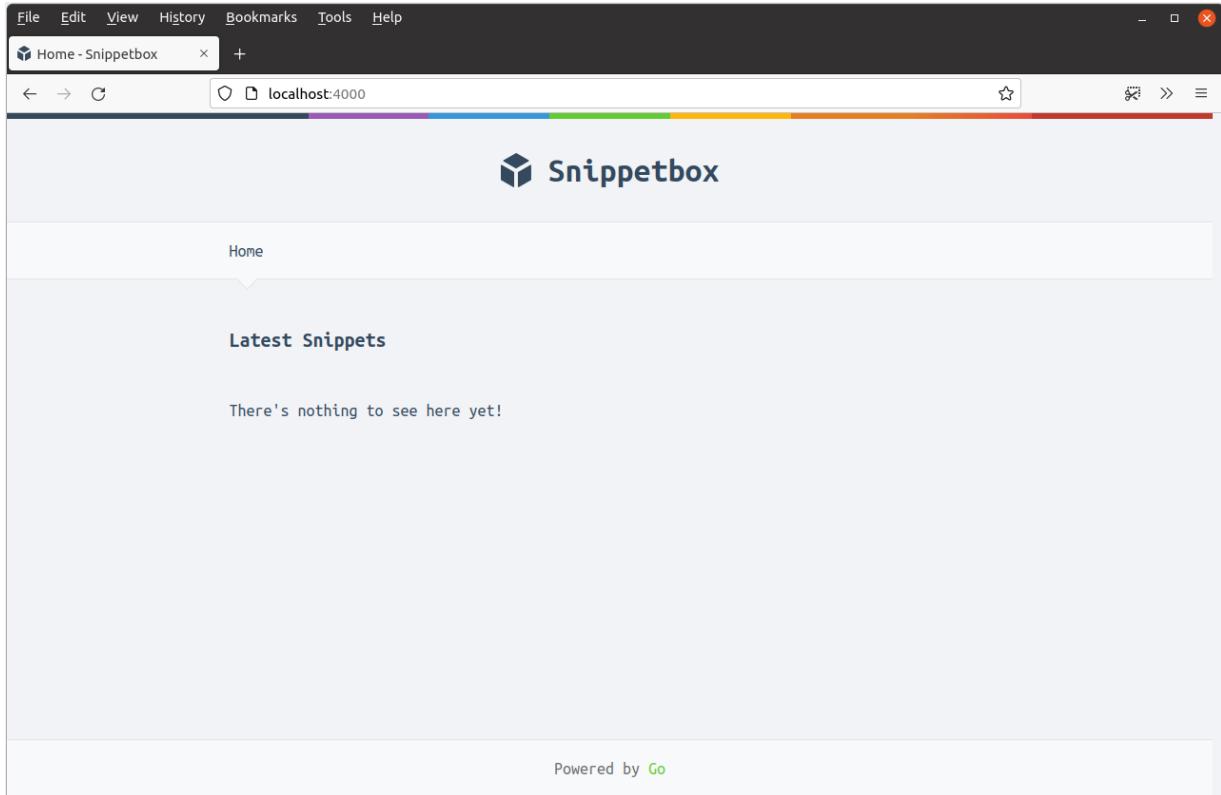
Using the static files

With the file server working properly, we can now update the `ui/html/base tmpl` file to make use of the static files:

```
File: ui/html/base tmpl

{{define "base"}}
<!doctype html>
<html lang='en'>
  <head>
    <meta charset='utf-8'>
    <title>{{template "title" .}} - Snippetbox</title>
    <!-- Link to the CSS stylesheet and favicon -->
    <link rel='stylesheet' href='/static/css/main.css'>
    <link rel='shortcut icon' href='/static/img/favicon.ico' type='image/x-icon'>
    <!-- Also link to some fonts hosted by Google -->
    <link rel='stylesheet' href='https://fonts.googleapis.com/css?family=Ubuntu+Mono:400,700'>
  </head>
  <body>
    <header>
      <h1><a href='/'>Snippetbox</a></h1>
    </header>
    {{template "nav" .}}
    <main>
      {{template "main" .}}
    </main>
    <footer>Powered by <a href='https://golang.org/'>Go</a></footer>
    <!-- And include the JavaScript file -->
    <script src='/static/js/main.js' type='text/javascript'></script>
  </body>
</html>
{{end}}
```

Make sure you save the changes, then restart the server and visit `http://localhost:4000`. Your home page should now look like this:



Additional information

File server features and functions

Go's `http.FileServer` handler has a few really nice features that are worth mentioning:

- It sanitizes all request paths by running them through the `path.Clean()` function before searching for a file. This removes any `.` and `..` elements from the URL path, which helps to stop directory traversal attacks. This feature is particularly useful if you're using the fileserver in conjunction with a router that doesn't automatically sanitize URL paths.
- `Range requests` are fully supported. This is great if your application is serving large files and you want to support resumable downloads. You can see this functionality in action if you use curl to request bytes 100-199 of the `logo.png` file, like so:

```
$ curl -i -H "Range: bytes=100-199" --output - http://localhost:4000/static/img/logo.png
HTTP/1.1 206 Partial Content
Accept-Ranges: bytes
Content-Length: 100
Content-Range: bytes 100-199/1075
Content-Type: image/png
Last-Modified: Wed, 18 Mar 2024 11:29:23 GMT
Date: Wed, 18 Mar 2024 11:29:23 GMT
[binary data]
```

- The `Last-Modified` and `If-Modified-Since` headers are transparently supported. If a file hasn't changed since the user last requested it, then `http.FileServer` will send a `304 Not Modified` status code instead of the file itself. This helps reduce latency and processing overhead for both the client and server.
- The `Content-Type` is automatically set from the file extension using the `mime.TypeByExtension()` function. You can add your own custom extensions and content types using the `mime.AddExtensionType()` function if necessary.

Performance

In this chapter we set up the file server so that it serves files out of the `./ui/static` directory on your hard disk.

But it's important to note that `http.FileServer` probably won't be reading these files from disk once the application is up-and-running. Both `Windows` and `Unix-based` operating systems cache recently-used files in RAM, so (for frequently-served files at least) it's likely that `http.FileServer` will be serving them from RAM rather than making the `relatively slow` round-trip to your hard disk.

Serving single files

Sometimes you might want to serve a single file from within a handler. For this there's the `http.ServeFile()` function, which you can use like so:

```
func downloadHandler(w http.ResponseWriter, r *http.Request) {
    http.ServeFile(w, r, "./ui/static/file.zip")
}
```

Warning: `http.ServeFile()` does not automatically sanitize the file path. If you're constructing a file path from untrusted user input, to avoid directory traversal attacks you *must* sanitize the input with `filepath.Clean()` before using it.

Disabling directory listings

If you want to disable directory listings there are a few different approaches you can take.

The simplest way? Add a blank `index.html` file to the specific directory that you want to disable listings for. This will then be served instead of the directory listing, and the user will get a `200 OK` response with no body. If you want to do this for all directories under `./ui/static` you can use the command:

```
$ find ./ui/static -type d -exec touch {}/index.html \;
```

A more complicated (but arguably better) solution is to create a custom implementation of `http.FileSystem`, and have it return an `os.ErrNotExist` error for any directories. A full explanation and sample code can be found in [this blog post](#).

The http.Handler interface

Before we go any further there's a little theory that we should cover. It's a bit complicated, so if you find this chapter hard-going don't worry. Carry on with the application build and circle back to it later once you're more familiar with Go.

In the previous chapters I've thrown around the term handler without explaining what it truly means. Strictly speaking, what we mean by handler is *an object which satisfies the `http.Handler` interface*:

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

In simple terms, this basically means that to be a handler an object *must* have a `ServeHTTP()` method with the exact signature:

```
ServeHTTP(http.ResponseWriter, *http.Request)
```

So in its simplest form a handler might look something like this:

```
type home struct {}

func (h *home) ServeHTTP(w http.ResponseWriter, r *http.Request) {
    w.Write([]byte("This is my home page"))
}
```

Here we have an object (in this case it's an empty `home` struct, but it could equally be a string or function or anything else), and we've implemented a method with the signature `ServeHTTP(http.ResponseWriter, *http.Request)` on it. That's all we need to make a handler.

You could then register this with a servemux using the `Handle` method like so:

```
mux := http.NewServeMux()
mux.Handle("/", &home{})
```

When this servemux receives a HTTP request for `"/"`, it will then call the `ServeHTTP()`

method of the `home` struct — which in turn writes the HTTP response.

Handler functions

Now, creating an object just so we can implement a `ServeHTTP()` method on it is long-winded and a bit confusing. Which is why in practice it's far more common to write your handlers as a normal function (like we have been so far in this book). For example:

```
func home(w http.ResponseWriter, r *http.Request) {
    w.Write([]byte("This is my home page"))
}
```

But this `home` function is just a normal function; it doesn't have a `ServeHTTP()` method. So in itself it *isn't* a handler.

Instead we can *transform* it into a handler using the `http.HandlerFunc()` adapter, like so:

```
mux := http.NewServeMux()
mux.Handle("/", http.HandlerFunc(home))
```

The `http.HandlerFunc()` adapter works by automatically adding a `ServeHTTP()` method to the `home` function. When executed, this `ServeHTTP()` method then simply *calls the code inside of the original `home` function*. It's a roundabout but convenient way of coercing a normal function into satisfying the `http.Handler` interface.

Throughout this project so far we've been using the `HandleFunc()` method to register our handler functions with the servemux. This is just some syntactic sugar that transforms a function to a handler and registers it in one step, instead of having to do it manually. The example immediately above is functionality equivalent to this:

```
mux := http.NewServeMux()
mux.HandleFunc("/", home)
```

Chaining handlers

The eagle-eyed of you might have noticed something interesting right at the start of this project. The `http.ListenAndServe()` function takes a `http.Handler` object as the second parameter:

```
func ListenAndServe(addr string, handler Handler) error
```

... but we've been passing in a servemux.

We were able to do this because the servemux also has a [ServeHTTP\(\)](#) method, meaning that it too satisfies the [http.Handler](#) interface.

For me it simplifies things to think of the servemux as just being a *special kind of handler*, which instead of providing a response itself passes the request on to a second handler. This isn't as much of a leap as it might first sound. Chaining handlers together is a very common idiom in Go, and something that we'll do a lot of later in this project.

In fact, what exactly is happening is this: When our server receives a new HTTP request, it calls the servemux's [ServeHTTP\(\)](#) method. This looks up the relevant handler based on the request method and URL path, and in turn calls that handler's [ServeHTTP\(\)](#) method. You can think of a Go web application as a *chain of ServeHTTP() methods being called one after another*.

Requests are handled concurrently

There is one more thing that's really important to point out: *all incoming HTTP requests are served in their own goroutine*. For busy servers, this means it's very likely that the code in or called by your handlers will be running concurrently. While this helps make Go blazingly fast, the downside is that you need to be aware of (and protect against) [race conditions](#) when accessing shared resources from your handlers.

Configuration and error handling

In this section of the book we're going to do some housekeeping. We won't add much new functionality to our application, but instead focus on improvements that will make it easier to develop and manage.

You'll learn how to:

- Pass configuration settings for your application at runtime in an easy and idiomatic way using [command-line flags](#).
- Create a [custom logger](#) for writing structured and levelled log entries, and use it throughout your application.
- Make [dependencies](#) available to your handlers in a way that's extensible, type-safe, and doesn't get in the way when it comes to writing tests.
- [Centralize error handling](#) so that you don't need to repeat yourself when writing code.

Managing configuration settings

Our web application's `main.go` file currently contains a couple of hard-coded configuration settings:

- The network address for the server to listen on (currently ":4000")
- The file path for the static files directory (currently `".ui/static"`)

Having these hard-coded isn't ideal. There's no separation between our configuration settings and code, and we can't change the settings at runtime (which is important if you need different settings for development, testing and production environments).

In this chapter we'll start to improve that, beginning by making the network address for our server configurable at runtime.

Command-line flags

In Go, a common and idiomatic way to manage configuration settings is to use command-line flags when starting an application. For example:

```
$ go run ./cmd/web -addr=:80
```

The easiest way to accept and parse a command-line flag in your application is with a line of code like this:

```
addr := flag.String("addr", ":4000", "HTTP network address")
```

This essentially defines a new command-line flag with the name `addr`, a default value of ":4000" and some short help text explaining what the flag controls. The value of the flag will be stored in the `addr` variable at runtime.

Let's use this in our application and swap out the hard-coded network address in favor of a command-line flag instead:

```

File: cmd/web/main.go

package main

import (
    "flag" // New import
    "log"
    "net/http"
)

func main() {
    // Define a new command-line flag with the name 'addr', a default value of ":4000"
    // and some short help text explaining what the flag controls. The value of the
    // flag will be stored in the addr variable at runtime.
    addr := flag.String("addr", ":4000", "HTTP network address")

    // Importantly, we use the flag.Parse() function to parse the command-line flag.
    // This reads in the command-line flag value and assigns it to the addr
    // variable. You need to call this *before* you use the addr variable
    // otherwise it will always contain the default value of ":4000". If any errors are
    // encountered during parsing the application will be terminated.
    flag.Parse()

    mux := http.NewServeMux()

    fileServer := http.FileServer(http.Dir("./ui/static/"))
    mux.Handle("GET /static/", http.StripPrefix("/static", fileServer))

    mux.HandleFunc("GET /${}", home)
    mux.HandleFunc("GET /snippet/view/{id}", snippetView)
    mux.HandleFunc("GET /snippet/create", snippetCreate)
    mux.HandleFunc("POST /snippet/create", snippetCreatePost)

    // The value returned from the flag.String() function is a pointer to the flag
    // value, not the value itself. So in this code, that means the addr variable
    // is actually a pointer, and we need to dereference it (i.e. prefix it with
    // the * symbol) before using it. Note that we're using the log.Printf()
    // function to interpolate the address with the log message.
    log.Printf("starting server on %s", *addr)

    // And we pass the dereferenced addr pointer to http.ListenAndServe() too.
    err := http.ListenAndServe(*addr, mux)
    log.Fatal(err)
}

```

Save this file and try using the `-addr` flag when you start the application. You should find that the server now listens on whatever address you specify, like so:

```

$ go run ./cmd/web -addr=:9999
2024/03/18 11:29:23 starting server on :9999

```

Note: Ports 0-1023 are restricted and (typically) can only be used by services which have root privileges. If you try to use one of these ports you should get a `bind: permission denied` error message on start-up.

Default values

Command-line flags are completely optional. For instance, if you run the application with no `-addr` flag the server will fall back to listening on address ":4000" (which is the default value we specified).

```
$ go run ./cmd/web
2024/03/18 11:29:23 starting server on :4000
```

There are no rules about what to use as the default values for your command-line flags. I like to use defaults which make sense for my development environment, because it saves me time and typing when I'm building an application. But YMMV. You might prefer the safer approach of setting defaults for your production environment instead.

Type conversions

In the code above we've used the `flag.String()` function to define the command-line flag. This has the benefit of converting whatever value the user provides at runtime to a `string` type. If the value *can't* be converted to a `string` then the application will print an error message and exit.

Go also has a range of other functions including `flag.Int()`, `flag.Bool()`, `flag.Float64()` and `flag.Duration()` for defining flags. These work in exactly the same way as `flag.String()`, except they automatically convert the command-line flag value to the appropriate type.

Automated help

Another great feature is that you can use the `-help` flag to list all the available command-line flags for an application and their accompanying help text. Give it a try:

```
$ go run ./cmd/web -help
Usage of /tmp/go-build3672328037/b001/exe/web:
-addr string
    HTTP network address (default ":4000")
```

So, all in all, this is starting to look really good. We've introduced an idiomatic way of managing configuration settings for our application at runtime, and thanks to the `-help` flag, we've also got an explicit and documented interface between our application and its

operating configuration.

Additional information

Environment variables

If you've built and deployed web applications before, then you're probably thinking *what about environment variables?* Surely it's *good-practice* to store configuration settings there?

If you want, you *can* store your configuration settings in environment variables and access them directly from your application by using the `os.Getenv()` function like so:

```
addr := os.Getenv("SNIPPETBOX_ADDR")
```

But this has some drawbacks compared to using command-line flags. You can't specify a default setting (the return value from `os.Getenv()` is the empty string if the environment variable doesn't exist), you don't get the `-help` functionality that you do with command-line flags, and the return value from `os.Getenv()` is *always* a string — you don't get automatic type conversions like you do with `flag.Int()`, `flag.Bool()` and the other command line flag functions.

Instead, you can get the best of both worlds by passing the environment variable as a command-line flag when starting the application. For example:

```
$ export SNIPPETBOX_ADDR=:9999
$ go run ./cmd/web -addr=$SNIPPETBOX_ADDR
2024/03/18 11:29:23 starting server on :9999
```

Boolean flags

For flags defined with `flag.Bool()`, omitting a value when starting the application is the same as writing `-flag=true`. The following two commands are equivalent:

```
$ go run ./example -flag=true
$ go run ./example -flag
```

You must explicitly use `-flag=false` if you want to set a boolean flag value to false.

Pre-existing variables

It's possible to parse command-line flag values into the memory addresses of pre-existing variables, using `flag.StringVar()`, `flag.IntVar()`, `flag.BoolVar()`, and similar functions for other types.

These functions are particularly useful if you want to store all your configuration settings in a single struct. As a rough example:

```
type config struct {
    addr      string
    staticDir string
}

...

var cfg config

flag.StringVar(&cfg.addr, "addr", ":4000", "HTTP network address")
flag.StringVar(&cfg.staticDir, "static-dir", "./ui/static", "Path to static assets")

flag.Parse()
```

Structured logging

At the moment we’re outputting log entries from our code using the `log.Printf()` and `log.Fatal()` functions. A good example of this is the “*starting server...*” log entry that we print just before our server starts:

```
log.Printf("starting server on %s", *addr)
```

Both the `log.Printf()` and `log.Fatal()` functions output log entries using Go’s standard logger from the `log` package, which — by default — prefixes a message with the local date and time and writes out to the standard error stream (which should display in your terminal window).

```
$ go run ./cmd/web/
2024/03/18 11:29:23 starting server on :4000
```

For many applications, using the standard logger will be *good enough*, and there’s no need to do anything more complex.

But for applications which do a lot of logging, you may want to make the log entries easier to filter and work with. For example, you might want to distinguish between different *severities* of log entries (like informational and error entries), or to enforce a consistent structure for log entries so that they are easy for external programs or services to parse.

To support this, the Go standard library includes the `log/slog` package which lets you create custom *structured loggers* that output log entries in a set format. Each log entry includes the following things:

- A timestamp with millisecond precision.
- The severity level of the log entry (`Debug`, `Info`, `Warn` or `Error`).
- The log message (an arbitrary `string` value).
- Optionally, any number of key-value pairs (known as *attributes*) containing additional information.

Creating a structured logger

The code for creating a structured logger with the `log/slog` package can be a little bit confusing the first time you see it.

The key thing to understand is that all structured loggers have a *structured logging handler* associated with them (not to be confused with a HTTP handler), and it's actually this handler that controls how log entries are formatted and where they are written to.

The code for creating a logger looks like this:

```
loggerHandler := slog.NewTextHandler(os.Stdout, &slog.HandlerOptions{...})  
logger := slog.New(loggerHandler)
```

In the first line of code we first use the `slog.NewTextHandler()` function to create the structured logging handler. This function accepts two arguments:

- The first argument is the write destination for the log entries. In the example above we've set it to `os.Stdout`, which means it will write log entries to the standard output stream.
- The second argument is a pointer to a `slog.HandlerOptions` struct, which you can use to customize the behavior of the handler. We'll take a look at some of the available customizations at the end of this chapter. If you're happy with the defaults and don't want to change anything, you can pass `nil` as the second argument instead.

Then in the second line of code, we actually create the structured logger by passing the handler to the `slog.New()` function.

In practice, it's more common to do all this in a single line of code:

```
logger := slog.New(slog.NewTextHandler(os.Stdout, &slog.HandlerOptions{...}))
```

Using a structured logger

Once you've created a structured logger, you can then write a log entry at a specific severity level by calling the `Debug()`, `Info()`, `Warn()` or `Error()` methods on the logger. As an example, the following line of code:

```
logger.Info("request received")
```

Would result in a log entry that looks like this:

```
time=2024-03-18T11:29:23.000+00:00 level=INFO msg="request received"
```

The `Debug()`, `Info()`, `Warn()` or `Error()` methods are *variadic methods* which accept an arbitrary number of additional attributes (key-value pairs). Like so:

```
logger.Info("request received", "method", "GET", "path", "/")
```

In this example, we've added two extra attributes to the log entry: the key `"method"` and value `"GET"`, and the key `"path"` and value `"/"`. Attribute keys must always be strings, but the values can be of any type. In this example, the log entry will look like this:

```
time=2024-03-18T11:29:23.000+00:00 level=INFO msg="request received" method=GET path=/
```

Note: If your attribute keys, values, or log message contain `"` or `=` characters or any whitespace, they will be wrapped in double quotes in the log output. We can see this behavior in the example above, where the log message `msg="request received"` is quoted.

Adding structured logging to our application

OK, let's go ahead and update our `main.go` file to use a structured logger instead of Go's standard logger. Like so:

File: cmd/web/main.go

```
package main

import (
    "flag"
    "log/slog" // New import
    "net/http"
    "os" // New import
)

func main() {
    addr := flag.String("addr", ":4000", "HTTP network address")
    flag.Parse()

    // Use the slog.New() function to initialize a new structured logger, which
    // writes to the standard out stream and uses the default settings.
    logger := slog.New(slog.NewTextHandler(os.Stdout, nil))

    mux := http.NewServeMux()

    fileServer := http.FileServer(http.Dir("./ui/static/"))
    mux.Handle("GET /static/", http.StripPrefix("/static", fileServer))

    mux.HandleFunc("GET /${}", home)
    mux.HandleFunc("GET /snippet/view/{id}", snippetView)
    mux.HandleFunc("GET /snippet/create", snippetCreate)
    mux.HandleFunc("POST /snippet/create", snippetCreatePost)

    // Use the Info() method to log the starting server message at Info severity
    // (along with the listen address as an attribute).
    logger.Info("starting server", "addr", *addr)

    err := http.ListenAndServe(*addr, mux)
    // And we also use the Error() method to log any error message returned by
    // http.ListenAndServe() at Error severity (with no additional attributes),
    // and then call os.Exit(1) to terminate the application with exit code 1.
    logger.Error(err.Error())
    os.Exit(1)
}
```

Important: There is no structured logging equivalent to the `log.Fatal()` function that we can use to handle an error returned by `http.ListenAndServe()`. Instead, the closest we can get is logging a message at the `Error` severity level and then manually calling `os.Exit(1)` to terminate the application with the `exit code 1`, like we are in the code above.

Alright... let's try this out!

Go ahead and run the application, then open *another* terminal window and try to run it a second time. This should generate an error because the network address our server wants to listen on (`:4000`) is already in use.

The log output in your second terminal should look a bit like this:

```
$ go run ./cmd/web
time=2024-03-18T11:29:23.000+00:00 level=INFO msg="starting server" addr=:4000
time=2024-03-18T11:29:23.000+00:00 level=ERROR msg="listen tcp :4000: bind: address already in use"
exit status 1
```

This is looking pretty good. We can see that the two log entries contain different information, but are formatted in the same overall way.

The first log entry has the severity `level=INFO` and message `msg="starting server"`, along with the additional `addr=:4000` attribute. In contrast, we see that the second log entry has the severity `level=ERROR`, the `msg` value contains the content of the error message, and there are no additional attributes.

Additional information

Safer attributes

Let's say that you accidentally write some code where you forget to include either the key or value for an attribute. For example:

```
logger.Info("starting server", "addr") // Oops, the value for "addr" is missing
```

When this happens, the log entry will still be written but the attribute will have the key `!BADKEY`, like so:

```
time=2024-03-18T11:29:23.000+00:00 level=INFO msg="starting server" !BADKEY=addr
```

To avoid this happening and catch any problems at compile-time, you can use the `slog.Any()` function to create an attribute pair instead:

```
logger.Info("starting server", slog.Any("addr", ":4000"))
```

Or you can go even further and introduce some additional type safety by using the `slog.String()`, `slog.Int()`, `slog.Bool()`, `slog.Time()` and `slog.Duration()` functions to create attributes with a specific type of value.

```
logger.Info("starting server", slog.String("addr", ":4000"))
```

Whether you want to use these functions or not is up to you. The [log/slog](#) package is relatively new to Go (introduced in Go 1.21), and there isn't much in the way of established best-practices or conventions around using it yet. But the trade-off is straightforward... using functions like `slog.String()` to create attributes is more verbose, but safer in sense that it reduces the risk of bugs in your application.

JSON formatted logs

The `slog.NewTextHandler()` function that we've used in this chapter creates a handler that writes plaintext log entries. But it's possible to create a handler that writes log entries as *JSON objects* instead using the `slog.NewJSONHandler()` function. Like so:

```
logger := slog.New(slog.NewJSONHandler(os.Stdout, nil))
```

When using the JSON handler, the log output will look similar to this:

```
{"time": "2024-03-18T11:29:23.0000000+00:00", "level": "INFO", "msg": "starting server", "addr": ":4000"}  
{"time": "2024-03-18T11:29:23.0000000+00:00", "level": "ERROR", "msg": "listen tcp :4000: bind: address already in use"}
```

Minimum log level

As we've mentioned a couple of times, the [log/slog](#) package supports four severity levels: `Debug`, `Info`, `Warn` and `Error` *in that order*. `Debug` is the least severe level, and `Error` is the most severe.

By default, the minimum log level for a structured logger is `Info`. That means that any log entries with a severity *less* than `Info` — i.e. `Debug` level entries — will be silently discarded.

You can use the `slog.HandlerOptions` struct to override this and set the minimum level to `Debug` (or any other level) if you want:

```
logger := slog.New(slog.NewTextHandler(os.Stdout, &slog.HandlerOptions{  
    Level: slog.LevelDebug,  
}))
```

Caller location

You can also customize the handler so that it includes the filename and line number of the

calling source code in the log entries, like so:

```
logger := slog.New(slog.NewTextHandler(os.Stdout, &slog.HandlerOptions{
    AddSource: true,
}))
```

The log entries will look similar to this, with the caller location recorded under the `source` key:

```
time=2024-03-18T11:29:23.000+00:00 level=INFO source=/home/alex/code/snippetbox/cmd/web/main.go:32 msg="starting server" addr=
```

Decoupled logging

In this chapter we've set up our structured logger to write entries to `os.Stdout` — the standard out stream.

The big benefit of writing log entries to `os.Stdout` is that your application and logging are decoupled. Your application itself isn't concerned with the routing or storage of the logs, and that can make it easier to manage the logs differently depending on the environment.

During development, it's easy to view the log output because the standard out stream is displayed in the terminal.

In staging or production environments, you can redirect the stream to a final destination for viewing and archival. This destination could be on-disk files, or a logging service such as Splunk. Either way, the final destination of the logs can be managed by your execution environment independently of the application.

For example, we could redirect the standard out stream to a on-disk file when starting the application like so:

```
$ go run ./cmd/web >>/tmp/web.log
```

Note: Using the double arrow `>>` will append to an existing file, instead of truncating it when starting the application.

Concurrent logging

Custom loggers created by `slog.New()` are concurrency-safe. You can share a single logger

and use it across multiple goroutines and in your HTTP handlers without needing to worry about race conditions.

That said, if you have *multiple* structured loggers writing to the same destination then you need to be careful and ensure that the destination's underlying `Write()` method is also safe for concurrent use.

Dependency injection

If you open up your `handlers.go` file you'll notice that the `home` handler function is still writing error messages using Go's standard logger, not the structured logger that we now want to be using.

```
func home(w http.ResponseWriter, r *http.Request) {
    ...

    ts, err := template.ParseFiles(files...)
    if err != nil {
        log.Println(err.Error()) // This isn't using our new structured logger.
        http.Error(w, "Internal Server Error", http.StatusInternalServerError)
        return
    }

    err = ts.ExecuteTemplate(w, "base", nil)
    if err != nil {
        log.Println(err.Error()) // This isn't using our new structured logger.
        http.Error(w, "Internal Server Error", http.StatusInternalServerError)
    }
}
```

This raises a good question: *how can we make our new structured logger available to our `home` function from `main()`?*

And this question generalizes further. Most web applications will have multiple dependencies that their handlers need to access, such as a database connection pool, centralized error handlers, and template caches. What we really want to answer is: *how can we make any dependency available to our handlers?*

There are a few different ways to do this, the simplest being to just put the dependencies in global variables. But in general, it is good practice to *inject dependencies* into your handlers. It makes your code more explicit, less error-prone, and easier to unit test than if you use global variables.

For applications where all your handlers are in the same package, like ours, a neat way to inject dependencies is to put them into a custom `application` struct, and then define your handler functions as methods against `application`.

I'll demonstrate.

First open your `main.go` file and create a new `application` struct like so:

```

File: cmd/web/main.go

package main

import (
    "flag"
    "log/slog"
    "net/http"
    "os"
)

// Define an application struct to hold the application-wide dependencies for the
// web application. For now we'll only include the structured logger, but we'll
// add more to this as the build progresses.
type application struct {
    logger *slog.Logger
}

func main() {
    ...
}

```

And then in the `handlers.go` file, we want to update the handler functions so that they become *methods against the application struct* and use the structured logger that it contains.

```

File: cmd/web/handlers.go

package main

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

// Change the signature of the home handler so it is defined as a method against
// *application.
func (app *application) home(w http.ResponseWriter, r *http.Request) {
    w.Header().Add("Server", "Go")

    files := []string{
        "./ui/html/base tmpl",
        "./ui/html/partials/nav tmpl",
        "./ui/html/pages/home tmpl",
    }

    ts, err := template.ParseFiles(files...)
    if err != nil {
        // Because the home handler is now a method against the application
        // struct it can access its fields, including the structured logger. We'll
        // use this to create a log entry at Error level containing the error
        // message, also including the request method and URI as attributes to
        // assist with debugging.
        app.logger.Error(err.Error(), "method", r.Method, "uri", r.URL.RequestURI())
        http.Error(w, "Internal Server Error", http.StatusInternalServerError)
        return
    }
}

```

```

err = ts.ExecuteTemplate(w, "base", nil)
if err != nil {
    // And we also need to update the code here to use the structured logger
    // too.
    app.logger.Error(err.Error(), "method", r.Method, "uri", r.URL.RequestURI())
    http.Error(w, "Internal Server Error", http.StatusInternalServerError)
}

// Change the signature of the snippetView handler so it is defined as a method
// against *application.
func (app *application) snippetView(w http.ResponseWriter, r *http.Request) {
    id, err := strconv.Atoi(r.PathValue("id"))
    if err != nil || id < 1 {
        http.NotFound(w, r)
        return
    }

    fmt.Fprintf(w, "Display a specific snippet with ID %d...", id)
}

// Change the signature of the snippetCreate handler so it is defined as a method
// against *application.
func (app *application) snippetCreate(w http.ResponseWriter, r *http.Request) {
    w.Write([]byte("Display a form for creating a new snippet..."))
}

// Change the signature of the snippetCreatePost handler so it is defined as a method
// against *application.
func (app *application) snippetCreatePost(w http.ResponseWriter, r *http.Request) {
    w.WriteHeader(http.StatusCreated)
    w.Write([]byte("Save a new snippet..."))
}

```

And finally let's wire things together in our `main.go` file:

```
File: cmd/web/main.go
```

```
package main

import (
    "flag"
    "log/slog"
    "net/http"
    "os"
)

type application struct {
    logger *slog.Logger
}

func main() {
    addr := flag.String("addr", ":4000", "HTTP network address")
    flag.Parse()

    logger := slog.New(slog.NewTextHandler(os.Stdout, nil))

    // Initialize a new instance of our application struct, containing the
    // dependencies (for now, just the structured logger).
    app := &application{
        logger: logger,
    }

    mux := http.NewServeMux()

    fileServer := http.FileServer(http.Dir("./ui/static/"))
    mux.Handle("GET /static/", http.StripPrefix("/static", fileServer))

    // Swap the route declarations to use the application struct's methods as the
    // handler functions.
    mux.HandleFunc("GET /${}", app.home)
    mux.HandleFunc("GET /snippet/view/{id}", app.snippetView)
    mux.HandleFunc("GET /snippet/create", app.snippetCreate)
    mux.HandleFunc("POST /snippet/create", app.snippetCreatePost)

    logger.Info("starting server", "addr", *addr)

    err := http.ListenAndServe(*addr, mux)
    logger.Error(err.Error())
    os.Exit(1)
}
```

I understand that this approach might feel a bit complicated and convoluted, especially when an alternative is to simply make `logger` a global variable. But stick with me. As the application grows, and our handlers start to need more dependencies, this pattern will begin to show its worth.

Adding a deliberate error

Let's try this out by quickly adding a deliberate error to our application.

Open your terminal and rename the `ui/html/pages/home tmpl` to `ui/html/pages/home.bak`. When we run our application and make a request for the home page, this now should result in an error because the `ui/html/pages/home tmpl` file no longer exists.

Go ahead and make the change:

```
$ cd $HOME/code/snippetbox
$ mv ui/html/pages/home tmpl ui/html/pages/home.bak
```

Then run the application and make a request to <http://localhost:4000>. You should get an `Internal Server Error` HTTP response in your browser, and see a corresponding log entry at `Error` level in your terminal similar to this:

```
$ go run ./cmd/web
time=2024-03-18T11:29:23.000+00:00 level=INFO msg="starting server" addr=:4000
time=2024-03-18T11:29:23.000+00:00 level=ERROR msg="open ./ui/html/pages/home tmpl: no such file or directory" method=GET uri=
```

This demonstrates nicely that our structured `logger` is now being passed through to our `home` handler as a dependency, and is working as expected.

Leave the deliberate error in place for now; we'll need it again in the next chapter.

Additional information

Closures for dependency injection

The pattern that we're using to inject dependencies won't work if your handlers are spread across multiple packages. In that case, an alternative approach is to create a standalone `config` package which exports an `Application` struct, and have your handler functions close over this to form a *closure*. Very roughly:

```
// package config

type Application struct {
    logger *slog.Logger
}
```

```
// package foo

func ExampleHandler(app *config.Application) http.HandlerFunc {
    return func(w http.ResponseWriter, r *http.Request) {
        ...
        ts, err := template.ParseFiles(files...)
        if err != nil {
            app.logger.Error(err.Error(), "method", r.Method, "uri", r.URL.RequestURI())
            http.Error(w, "Internal Server Error", http.StatusInternalServerError)
            return
        }
        ...
    }
}
```

```
// package main

func main() {
    app := &config.Application{
        Logger: slog.New(slog.NewTextHandler(os.Stdout, nil)),
    }
    ...
    mux.Handle("/", foo.ExampleHandler(app))
    ...
}
```

You can find a complete and more concrete example of how to use the closure pattern in [this Gist](#).

Centralized error handling

Let's neaten up our application by moving some of the error handling code into helper methods. This will help [separate our concerns](#) and stop us repeating code as we progress through the build.

Go ahead and add a new `helpers.go` file under the `cmd/web` directory:

```
$ touch cmd/web/helpers.go
```

And add the following code:

```
File: cmd/web/helpers.go

package main

import (
    "net/http"
)

// The serverError helper writes a log entry at Error level (including the request
// method and URI as attributes), then sends a generic 500 Internal Server Error
// response to the user.
func (app *application) serverError(w http.ResponseWriter, r *http.Request, err error) {
    var (
        method = r.Method
        uri    = r.URL.RequestURI()
    )

    app.logger.Error(err.Error(), "method", method, "uri", uri)
    http.Error(w, http.StatusText(http.StatusInternalServerError), http.StatusInternalServerError)
}

// The clientError helper sends a specific status code and corresponding description
// to the user. We'll use this later in the book to send responses like 400 "Bad
// Request" when there's a problem with the request that the user sent.
func (app *application) clientError(w http.ResponseWriter, status int) {
    http.Error(w, http.StatusText(status), status)
}
```

In this code we've also introduced one other new thing: the `http.StatusText()` function. This returns a human-friendly text representation of a given HTTP status code — for example `http.StatusText(400)` will return the string `"Bad Request"`, and `http.StatusText(500)` will return the string `"Internal Server Error"`.

Now that's done, head back to your `handlers.go` file and update it to use the new `serverError()` helper:

```
File: cmd/web/handlers.go
```

```
package main

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

func (app *application) home(w http.ResponseWriter, r *http.Request) {
    w.Header().Add("Server", "Go")

    files := []string{
        "./ui/html/base tmpl",
        "./ui/html/partials/nav tmpl",
        "./ui/html/pages/home tmpl",
    }

    ts, err := template.ParseFiles(files...)
    if err != nil {
        app.serverError(w, r, err) // Use the serverError() helper.
        return
    }

    err = ts.ExecuteTemplate(w, "base", nil)
    if err != nil {
        app.serverError(w, r, err) // Use the serverError() helper.
    }
}

...
```

When that's updated, restart your application and make a request to <http://localhost:4000> in your browser.

Again, this should result in our (deliberate) error being raised and you should see the corresponding log entry in your terminal, including the request method and URI as attributes.

```
$ go run ./cmd/web
time=2024-03-18T11:29:23.000+00:00 level=INFO msg="starting server" addr=:4000
time=2024-03-18T11:29:23.000+00:00 level=ERROR msg="open ./ui/html/pages/home tmpl: no such file or directory" method=GET uri=
```

Revert the deliberate error

At this point we don't need the deliberate error anymore, so go ahead and fix it like so:

```
$ mv ui/html/pages/home.bak ui/html/pages/home tmpl
```

Additional information

Stack traces

You can use the `debug.Stack()` function to get a *stack trace* outlining the execution path of the application for the *current goroutine*. Including this as an attribute in your log entries can be helpful for debugging errors.

If you want, you could update the `serverError()` method so that it includes a stack trace in the log entries like so:

```
package main

import (
    "net/http"
    "runtime/debug"
)

func (app *application) serverError(w http.ResponseWriter, r *http.Request, err error) {
    var (
        method = r.Method
        uri    = r.URL.RequestURI()
        // Use debug.Stack() to get the stack trace. This returns a byte slice, which
        // we need to convert to a string so that it's readable in the log entry.
        trace = string(debug.Stack())
    )

    // Include the trace in the log entry.
    app.logger.Error(err.Error(), "method", method, "uri", uri, "trace", trace)

    http.Error(w, http.StatusText(http.StatusInternalServerError), http.StatusInternalServerError)
}
```

The log entry output would then look something like this (line breaks added for readability):

```
time=2024-03-18T11:29:23.000+00:00 level=ERROR msg="open ./ui/html/pages/home tmpl:
no such file or directory" method=GET uri=/ trace="goroutine 6 [running]:\nruntime/
debug.Stack()\n\t/usr/local/go/src/runtime/debug/stack.go:24 +0x5e\nmain.(*applic
ation).serverError(0xc00006c048, {0x8221b0, 0xc0000f40e0}, 0x3?, {0x820600, 0xc0000ab
5c0})\n\t/home/alex/code/snippetbox/cmd/web/helpers.go:14 +0x74\nmain.(*application
).home(0x10?, {0x8221b0?, 0xc0000f40e0}, 0xc0000fe000)\n\t/home/alex/code/snippetbo
x/cmd/web/handlers.go:24 +0x16a\nnet/http.HandlerFunc.ServeHTTP(0x4459e0?, {0x8221b
0?, 0xc0000f40e0?}, 0x6cc57a?)\n\t/usr/local/go/src/net/http/server.go:2136 +0x29\n
net/http.(*ServeMux).ServeHTTP(0xa7fde0?, {0x8221b0, 0xc0000f40e0}, 0xc0000fe000)\n
\t/usr/local/go/src/net/http/server.go:2514 +0x142\nnet/http.serverHandler.ServeHTT
P({0xc0000aaef0?}, {0x8221b0?, 0xc0000f40e0?}, 0x6?)\n\t/usr/local/go/src/net/http/
server.go:2938 +0x8e\nnet/http.(*conn).serve(0xc0000c0120, {0x8229e0, 0xc0000aae10})
\n\t/usr/local/go/src/net/http/server.go:2009 +0x5f4\ncreated by net/http.(*Server).
Serve in goroutine 1\n\t/usr/local/go/src/net/http/server.go:3086 +0x5cb\n"
```

Isolating the application routes

While we're refactoring our code there's one more change worth making.

Our `main()` function is beginning to get a bit crowded, so to keep it clear and focused I'd like to move the route declarations for the application into a standalone `routes.go` file, like so:

```
$ touch cmd/web/routes.go
```

```
File: cmd/web/routes.go

package main

import "net/http"

// The routes() method returns a servemux containing our application routes.
func (app *application) routes() *http.ServeMux {
    mux := http.NewServeMux()

    fileServer := http.FileServer(http.Dir("./ui/static/"))
    mux.Handle("GET /static/", http.StripPrefix("/static", fileServer))

    mux.HandleFunc("GET /${}", app.home)
    mux.HandleFunc("GET /snippet/view/{id}", app.snippetView)
    mux.HandleFunc("GET /snippet/create", app.snippetCreate)
    mux.HandleFunc("POST /snippet/create", app.snippetCreatePost)

    return mux
}
```

We can then update the `main.go` file to use this instead:

```
File: cmd/web/main.go
```

```
package main

...

func main() {
    addr := flag.String("addr", ":4000", "HTTP network address")
    flag.Parse()

    logger := slog.New(slog.NewTextHandler(os.Stdout, nil))

    app := &application{
        logger: logger,
    }

    logger.Info("starting server", "addr", *addr)

    // Call the new app.routes() method to get the servemux containing our routes,
    // and pass that to http.ListenAndServe().
    err := http.ListenAndServe(*addr, app.routes())
    logger.Error(err.Error())
    os.Exit(1)
}
```

This is quite a bit neater. The routes for our application are now isolated and encapsulated in the `app.routes()` method, and the responsibilities of our `main()` function are limited to:

- Parsing the runtime configuration settings for the application;
- Establishing the dependencies for the handlers; and
- Running the HTTP server.

Database-driven responses

For our Snippetbox web application to become truly useful we need somewhere to store (or *persist*) the data entered by users, and the ability to query this data store dynamically at runtime.

There are many different data stores we *could* use for our application — each with different pros and cons — but we'll opt for the popular relational database [MySQL](#).

Note: All of the general code patterns in this section of the book also apply to other databases like PostgreSQL or SQLite too. If you're following along and would prefer to use an alternative database, you can, but I recommend using MySQL for now to get a feeling for how everything works and then swapping databases as a practice exercise once you've finished the book.

In this section you'll learn how to:

- [Install a database driver](#) to act as a ‘middleman’ between MySQL and your Go application.
- [Connect to MySQL](#) from your web application (specifically, you'll learn how to establish a pool of reusable connections).
- Create a [standalone models package](#), so that your database logic is reusable and decoupled from your web application.
- Use the appropriate functions in Go's [database/sql](#) package to execute different types of SQL statements, and how to avoid common errors that can lead to your server running out of resources.
- [Prevent SQL injection](#) attacks by correctly using placeholder parameters.
- Use [transactions](#), so that you can execute multiple SQL statements in one atomic action.

Setting up MySQL

If you're following along, you'll need to install MySQL on your computer at this point. The official MySQL documentation contains comprehensive [installation instructions](#) for all types of operating systems, but if you're using Mac OS you should be able to install it with:

```
$ brew install mysql
```

Or if you're using a Linux distribution which supports `apt` (like Debian and Ubuntu) you can install it with:

```
$ sudo apt install mysql-server
```

While you are installing MySQL you might be asked to set a password for the `root` user. Remember to keep a mental note of this if you are; you'll need it in the next step.

Scaffolding the database

Once MySQL is installed you should be able to connect to it from your terminal as the `root` user. The command to do this will vary depending on the version of MySQL that you've got installed. For MySQL 5.7 and newer you should be able to connect by typing this:

```
$ sudo mysql  
mysql>
```

But if that doesn't work then try the following command instead, entering the password that you set during the installation.

```
$ mysql -u root -p  
Enter password:  
mysql>
```

Once connected, the first thing we need to do is establish a database in MySQL to store all the data for our project. Copy and paste the following commands into the mysql prompt to create a new `snippetbox` database using UTF8 encoding.

```
-- Create a new UTF-8 `snippetbox` database.
CREATE DATABASE snippetbox CHARACTER SET utf8mb4 COLLATE utf8mb4_unicode_ci;

-- Switch to using the `snippetbox` database.
USE snippetbox;
```

Then copy and paste the following SQL statement to create a new `snippets` table to hold the text snippets for our application:

```
-- Create a `snippets` table.
CREATE TABLE snippets (
    id INTEGER NOT NULL PRIMARY KEY AUTO_INCREMENT,
    title VARCHAR(100) NOT NULL,
    content TEXT NOT NULL,
    created DATETIME NOT NULL,
    expires DATETIME NOT NULL
);

-- Add an index on the created column.
CREATE INDEX idx_snippets_created ON snippets(created);
```

Each record in this table will have an integer `id` field which will act as the unique identifier for the text snippet. It will also have a short text `title` and the snippet content itself will be stored in the `content` field. We'll also keep some metadata about the times that the snippet was `created` and when it `expires`.

Let's also add some placeholder entries to the `snippets` table (which we'll use in the next couple of chapters). I'll use some short haiku as the content for the text snippets, but it really doesn't matter what they contain.

```
-- Add some dummy records (which we'll use in the next couple of chapters).
INSERT INTO snippets (title, content, created, expires) VALUES (
    'An old silent pond',
    'An old silent pond...\nA frog jumps into the pond,\nsplash! Silence again.\n\n- Matsuo Bashō',
    UTC_TIMESTAMP(),
    DATE_ADD(UTC_TIMESTAMP(), INTERVAL 365 DAY)
);

INSERT INTO snippets (title, content, created, expires) VALUES (
    'Over the wintry forest',
    'Over the wintry\nforest, winds howl in rage\nwith no leaves to blow.\n\n- Natsume Soseki',
    UTC_TIMESTAMP(),
    DATE_ADD(UTC_TIMESTAMP(), INTERVAL 365 DAY)
);

INSERT INTO snippets (title, content, created, expires) VALUES (
    'First autumn morning',
    'First autumn morning\nthe mirror I stare into\nshows my father''s face.\n\n- Murakami Kijo',
    UTC_TIMESTAMP(),
    DATE_ADD(UTC_TIMESTAMP(), INTERVAL 7 DAY)
);
```

Creating a new user

From a security point of view, it's not a good idea to connect to MySQL as the `root` user from a web application. Instead it's better to create a database user with restricted permissions on the database.

So, while you're still connected to the MySQL prompt run the following commands to create a new `web` user with `SELECT` and `INSERT` privileges only on the database.

```
CREATE USER 'web'@'localhost';
GRANT SELECT, INSERT, UPDATE, DELETE ON snippetbox.* TO 'web'@'localhost';
-- Important: Make sure to swap 'pass' with a password of your own choosing.
ALTER USER 'web'@'localhost' IDENTIFIED BY 'pass';
```

Once that's done type `exit` to leave the MySQL prompt.

Test the new user

You should now be able to connect to the `snippetbox` database as the `web` user using the following command. When prompted enter the password that you just set.

```
$ mysql -D snippetbox -u web -p
Enter password:
mysql>
```

If the permissions are working correctly you should find that you're able to perform `SELECT` and `INSERT` operations on the database correctly, but other commands such as `DROP TABLE` and `GRANT` will fail.

```
mysql> SELECT id, title, expires FROM snippets;
+----+-----+-----+
| id | title          | expires        |
+----+-----+-----+
| 1  | An old silent pond | 2025-03-18 10:00:26 |
| 2  | Over the wintry forest | 2025-03-18 10:00:26 |
| 3  | First autumn morning | 2024-03-25 10:00:26 |
+----+-----+-----+
3 rows in set (0.00 sec)

mysql> DROP TABLE snippets;
ERROR 1142 (42000): DROP command denied to user 'web'@'localhost' for table 'snippets'
```

Installing a database driver

To use MySQL from our Go web application we need to install a database driver. This essentially acts as a middleman, translating commands between Go and the MySQL database itself.

You can find a comprehensive [list of available drivers](#) on the Go wiki, but for our application we'll use the popular [go-sql-driver/mysql](#) driver.

To download it, go to your project directory and run the `go get` command like so:

```
$ cd $HOME/code/snippetbox
$ go get github.com/go-sql-driver/mysql@v1
go: added filippo.io/edwards25519 v1.1.0
go: added github.com/go-sql-driver/mysql v1.8.0
```

Note: The `go get` command will recursively download any dependencies that the package has. In this case, the [github.com/go-sql-driver/mysql](#) package itself uses the [filippo.io/edwards25519](#) package, so this will be downloaded too.

Notice here that we're postfixing the package path with `@v1` to indicate that we want to download the [latest available version](#) of [github.com/go-sql-driver/mysql](#) *with the major release number 1*.

At the time of writing the latest version is `v1.8.0`, but the version you download might be `v1.8.1`, `v1.9.0` or similar — and that's OK. Because the [go-sql-driver/mysql](#) package uses [semantic versioning](#) for its releases, any `v1.x.x` version should be compatible with the rest of the code in this book.

As an aside, if you want to download the latest version, irrespective of version number, you can simply omit the `@version` suffix like so:

```
$ go get github.com/go-sql-driver/mysql
```

Or if you want to download a specific version of a package, you can use the full version number. For example:

```
$ go get github.com/go-sql-driver/mysql@v1.0.3
```

Modules and reproducible builds

Now that the MySQL driver is installed, let's take a look at the `go.mod` file (which we created right at the start of the book). You should see a `require` block with two lines containing the path and exact version number of the packages that you downloaded:

```
File: go.mod

module snippetbox.alexewards.net

go 1.22.0

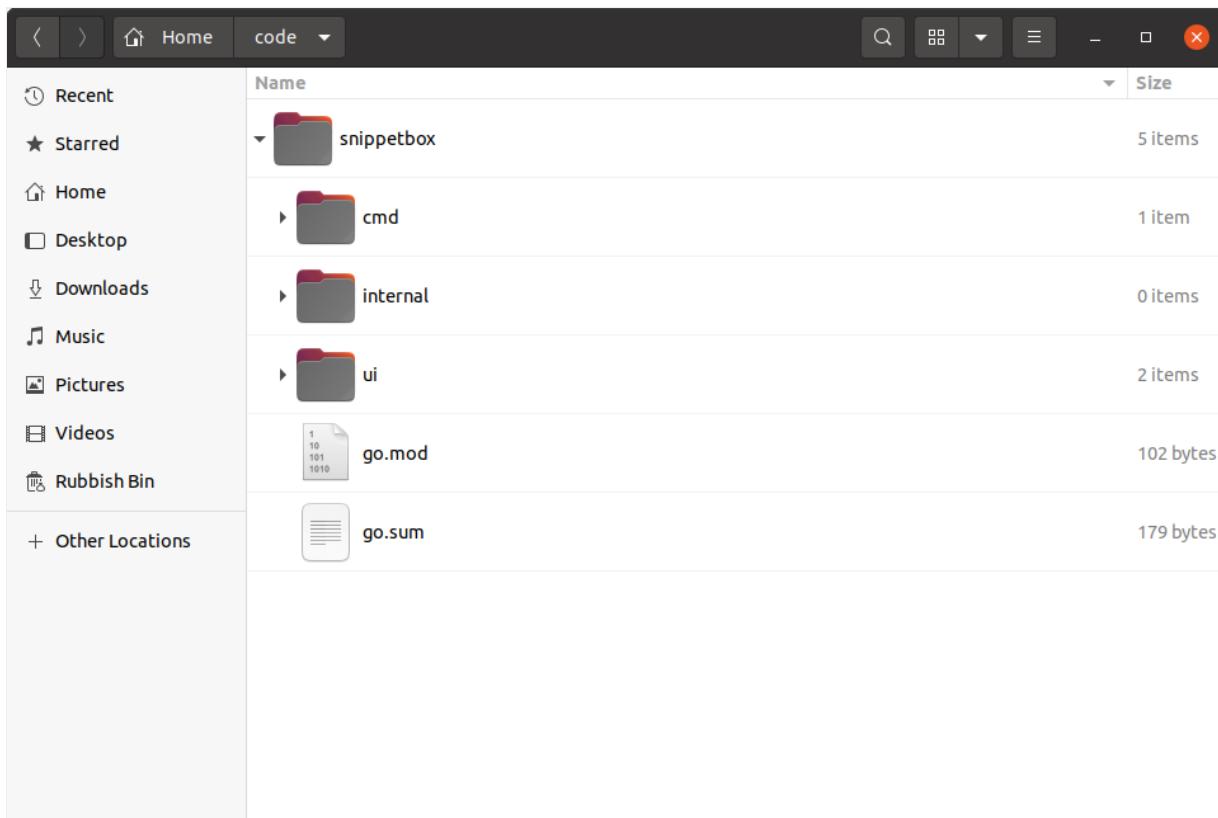
require (
    filippo.io/edwards25519 v1.1.0 // indirect
    github.com/go-sql-driver/mysql v1.8.0 // indirect
)
```

These lines in `go.mod` essentially tell the Go command exactly which version of a package should be used when you run a command like `go run`, `go test` or `go build` from your project directory.

This makes it easy to have multiple projects on the same machine where *different versions of the same package* are used. For example, this project is using `v1.8.0` of the MySQL driver, but you could have another codebase on your computer which uses `v1.5.0` and that would be A-OK.

Note: The `// indirect` annotation indicates that a package doesn't directly appear in any `import` statement in your codebase. Right now, we still haven't written any code that actually uses either the `github.com/go-sql-driver/mysql` or `filippo.io/edwards25519` packages, which is why they are both marked as indirect dependencies. We'll fix that in the next chapter.

You'll also see that a new file has been created in the root of your project directory called `go.sum`.



This `go.sum` file contains the cryptographic checksums representing the content of the required packages. If you open it up you should see something like this:

```
File: go.sum

filippo.io/edwards25519 v1.1.0 h1:FNF4tywRC1HmFuKw5xopWpigGjJKiJSV0Cqo0cJWDaA=
filippo.io/edwards25519 v1.1.0/go.mod h1:BxyFTGdwcka3PhytdK4V28tE5sGfRvvvRV7EaN4VDT4=
github.com/go-sql-driver/mysql v1.8.0 h1:UtktXaU2Nb64z/pLiGiXY4431SJ4/dR5cjMmlVHgnT4=
github.com/go-sql-driver/mysql v1.8.0/go.mod h1:wEB5Xgmk//ZZFJyE+qWhIsVGmvnEKlqwuVSjsCm7DZg=
```

The `go.sum` file isn't designed to be human-editable and generally you won't need to open it. But it serves two useful functions:

- If you run the `go mod verify` command from your terminal, this will verify that the checksums of the downloaded packages on your machine match the entries in `go.sum`, so you can be confident that they haven't been altered.

```
$ go mod verify
all modules verified
```

- If someone else needs to download all the dependencies for the project — which they can do by running `go mod download` — they will get an error if there is any mismatch between the packages they are downloading and the checksums in the file.

So, in summary:

- You (or someone else in the future) can run `go mod download` to download the exact versions of all the packages that your project needs.
- You can run `go mod verify` to ensure that nothing in those downloaded packages has been changed unexpectedly.
- Whenever you run `go run`, `go test` or `go build`, the exact package versions listed in `go.mod` will always be used.

And those things together makes it much easier to reliably create **reproducible builds** of your Go applications.

Additional information

Upgrading packages

Once a package has been downloaded and added to your `go.mod` file the package and version are ‘fixed’. But there are many reasons why you might want to upgrade to use a newer version of a package in the future.

To upgrade to latest available *minor or patch* release of a package, you can simply run `go get` with the `-u` flag like so:

```
$ go get -u github.com/foo/bar
```

Or alternatively, if you want to upgrade to a specific version then you should run the same command but with the appropriate `@version` suffix. For example:

```
$ go get -u github.com/foo/bar@v2.0.0
```

Removing unused packages

Sometimes you might `go get` a package only to realize later that you don’t need it anymore. When this happens you’ve got two choices.

You could either run `go get` and postfix the package path with `@none`, like so:

```
$ go get github.com/foo/bar@none
```

Or if you've removed all references to the package in your code, you can run `go mod tidy`, which will automatically remove any unused packages from your `go.mod` and `go.sum` files.

```
$ go mod tidy
```

Creating a database connection pool

Now that the MySQL database is all set up and we've got a driver installed, the natural next step is to connect to the database from our web application.

To do this we need Go's `sql.Open()` function, which you use a bit like this:

```
// The sql.Open() function initializes a new sql.DB object, which is essentially a
// pool of database connections.
db, err := sql.Open("mysql", "web:pass@/snippetbox?parseTime=true")
if err != nil {
    ...
}
```

There are a few things about this code to explain and emphasize:

- The first parameter to `sql.Open()` is the *driver name* and the second parameter is the *data source name* (sometimes also called a *connection string* or *DSN*) which describes how to connect to your database.
- The format of the data source name will depend on which database and driver you're using. Typically, you can find information and examples in the documentation for your specific driver. For the driver we're using you can find that documentation [here](#).
- The `parseTime=true` part of the DSN above is a *driver-specific* parameter which instructs our driver to convert SQL `TIME` and `DATE` fields to Go `time.Time` objects.
- The `sql.Open()` function returns a `sql.DB` object. This isn't a database connection — it's a *pool of many connections*. This is an important difference to understand. Go manages the connections in this pool as needed, automatically opening and closing connections to the database via the driver.
- The connection pool is safe for concurrent access, so you can use it from web application handlers safely.
- The connection pool is intended to be long-lived. In a web application it's normal to initialize the connection pool in your `main()` function and then pass the pool to your handlers. You shouldn't call `sql.Open()` in a short-lived HTTP handler itself — it would be a waste of memory and network resources.

Using it in our web application

Let's look at how to use `sql.Open()` in practice. Open up your `main.go` file and add the following code:

```
File: cmd/web/main.go

package main

import (
    "database/sql" // New import
    "flag"
    "log/slog"
    "net/http"
    "os"

    _ "github.com/go-sql-driver/mysql" // New import
)

...

func main() {
    addr := flag.String("addr", ":4000", "HTTP network address")
    // Define a new command-line flag for the MySQL DSN string.
    dsn := flag.String("dsn", "web:pass@/snippetbox?parseTime=true", "MySQL data source name")
    flag.Parse()

    logger := slog.New(slog.NewTextHandler(os.Stdout, nil))

    // To keep the main() function tidy I've put the code for creating a connection
    // pool into the separate openDB() function below. We pass openDB() the DSN
    // from the command-line flag.
    db, err := openDB(*dsn)
    if err != nil {
        logger.Error(err.Error())
        os.Exit(1)
    }

    // We also defer a call to db.Close(), so that the connection pool is closed
    // before the main() function exits.
    defer db.Close()

    app := &application{
        logger: logger,
    }

    logger.Info("starting server", "addr", *addr)

    // Because the err variable is now already declared in the code above, we need
    // to use the assignment operator = here, instead of the := 'declare and assign'
    // operator.
    err = http.ListenAndServe(*addr, app.routes())
    logger.Error(err.Error())
    os.Exit(1)
}

// The openDB() function wraps sql.Open() and returns a sql.DB connection pool
// for a given DSN.
func openDB(dsn string) (*sql.DB, error) {
    db, err := sql.Open("mysql", dsn)
    if err != nil {
        return nil, err
    }
}
```

```
    ...  
    }  
  
    err = db.Ping()  
    if err != nil {  
        db.Close()  
        return nil, err  
    }  
  
    return db, nil  
}
```

There're a few things about this code which are interesting:

- Notice how the import path for our driver is prefixed with an underscore? This is because our `main.go` file doesn't actually use anything in the `mysql` package. So if we try to import it normally the Go compiler will raise an error. However, we need the driver's `init()` function to run so that it can register itself with the `database/sql` package. The trick to getting around this is to alias the package name to the blank identifier, like we are here. This is standard practice for most of Go's SQL drivers.
- The `sql.Open()` function doesn't actually create any connections, all it does is initialize the pool for future use. Actual connections to the database are established lazily, as and when needed for the first time. So to verify that everything is set up correctly we need to use the `db.Ping()` method to create a connection and check for any errors. If there is an error, we call `db.Close()` to close the connection pool and return the error.
- Going back to the `main()` function, at this moment in time the call to `defer db.Close()` is a bit superfluous. Our application is only ever terminated by a signal interrupt (i.e. `Ctrl+C`) or by `os.Exit(1)`. In both of those cases, the program exits immediately and deferred functions are never run. But making sure to always close the connection pool is a good habit to get into, and it could be beneficial in the future if you add a graceful shutdown to your application.

Testing a connection

Make sure that the file is saved, and then try running the application. If everything has gone to plan, the connection pool should be established and the `db.Ping()` method should be able to create a connection without any errors. All being well, you should see the normal *starting server* log message like so:

```
$ go run ./cmd/web  
time=2024-03-18T11:29:23.000+00:00 level=INFO msg="starting server" addr=:4000
```

If the application fails to start and you get an "Access denied..." error message like below, then the problem probably lies with your DSN. Double-check that the username and password are correct, that your database users have the right permissions, and that your MySQL instance is using standard settings.

```
$ go run ./cmd/web
time=2024-03-18T11:29:23.000+00:00 level=ERROR msg="Error 1045 (28000): Access denied for user 'web'@'localhost' (using password: YES)"
exit status 1
```

Tidying the go.mod file

Now that our code is actually importing the github.com/go-sql-driver/mysql driver, you can run the `go mod tidy` command to tidy your `go.mod` file and remove any unnecessary `// indirect` annotations.

```
$ go mod tidy
```

Once you've done that, your `go.mod` file should now look like below — with github.com/go-sql-driver/mysql listed as a direct dependency and filippo.io/edwards25519 continuing to be an indirect dependency.

```
File: go.mod

module snippetbox.alexewards.net

go 1.22.0

require github.com/go-sql-driver/mysql v1.8.0

require filippo.io/edwards25519 v1.1.0 // indirect
```

Designing a database model

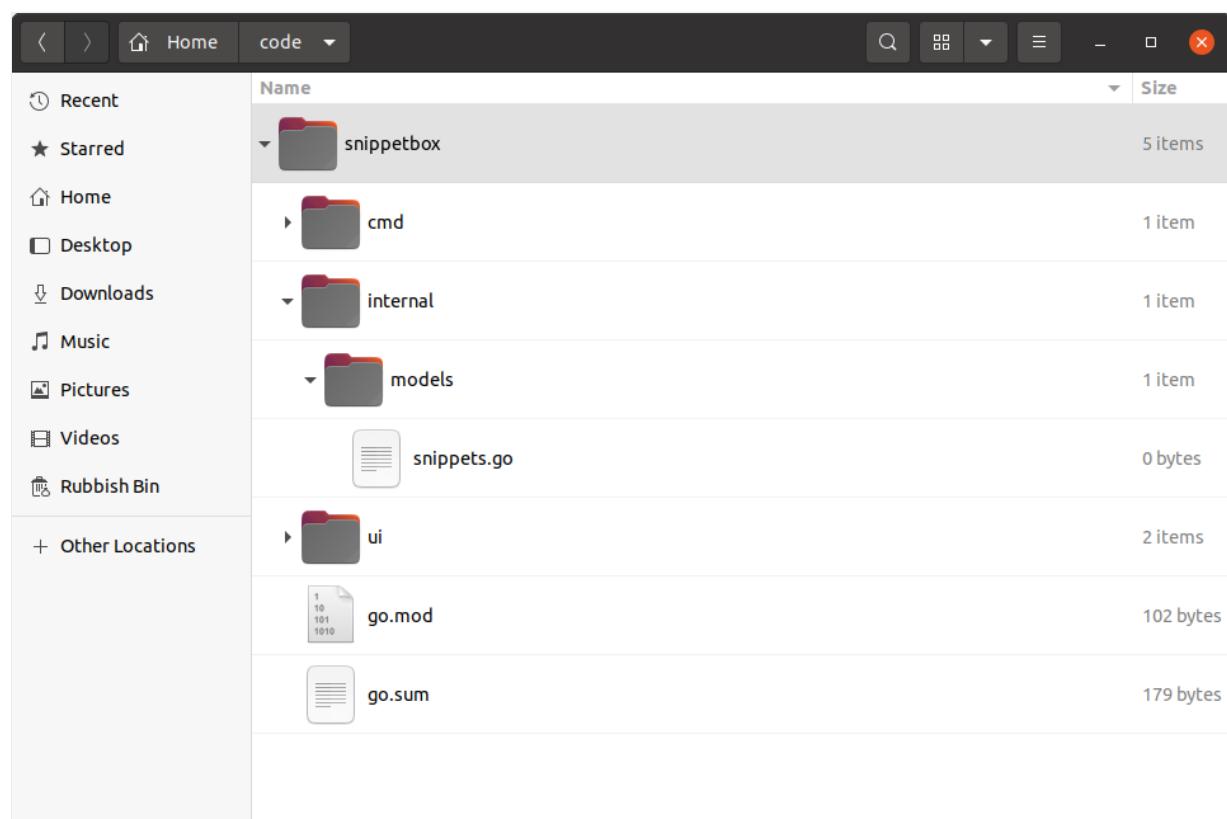
In this chapter we're going to sketch out a database model for our project.

If you don't like the term *model*, you might want to think of it as a *service layer* or *data access layer* instead. Whatever you prefer to call it, the idea is that we will encapsulate the code for working with MySQL in a separate package to the rest of our application.

For now, we'll create a skeleton database model and have it return a bit of dummy data. It won't do much, but I'd like to explain the pattern before we get into the nitty-gritty of SQL queries.

Sound OK? Then let's go ahead and create a new `internal/models` directory containing a `snippets.go` file:

```
$ mkdir -p internal/models  
$ touch internal/models/snippets.go
```



Remember: The `internal` directory is being used to hold ancillary non-application-specific code, which could potentially be reused. A database model which could be used by other applications in the future (like a *command line interface* application) fits the bill here.

Let's open the `internal/models/snippets.go` file and add a new `Snippet` struct to represent the data for an individual snippet, along with a `SnippetModel` type with methods on it to access and manipulate the snippets in our database. Like so:

```
File: internal/models/snippets.go

package models

import (
    "database/sql"
    "time"
)

// Define a Snippet type to hold the data for an individual snippet. Notice how
// the fields of the struct correspond to the fields in our MySQL snippets
// table?
type Snippet struct {
    ID      int
    Title   string
    Content string
    Created time.Time
    Expires time.Time
}

// Define a SnippetModel type which wraps a sql.DB connection pool.
type SnippetModel struct {
    DB *sql.DB
}

// This will insert a new snippet into the database.
func (m *SnippetModel) Insert(title string, content string, expires int) (int, error) {
    return 0, nil
}

// This will return a specific snippet based on its id.
func (m *SnippetModel) Get(id int) (Snippet, error) {
    return Snippet{}, nil
}

// This will return the 10 most recently created snippets.
func (m *SnippetModel) Latest() ([]Snippet, error) {
    return nil, nil
}
```

Using the SnippetModel

To use this model in our handlers we need to establish a new `SnippetModel` struct in our

`main()` function and then inject it as a dependency via the `application` struct — just like we have with our other dependencies.

Here's how:

File: cmd/web/main.go

```
package main

import (
    "database/sql"
    "flag"
    "log/slog"
    "net/http"
    "os"

    // Import the models package that we just created. You need to prefix this with
    // whatever module path you set up back in chapter 02.01 (Project Setup and Creating
    // a Module) so that the import statement looks like this:
    // "{your-module-path}/internal/models". If you can't remember what module path you
    // used, you can find it at the top of the go.mod file.
    "snippetbox.alexandedwards.net/internal/models"

    _ "github.com/go-sql-driver/mysql"
)

// Add a snippets field to the application struct. This will allow us to
// make the SnippetModel object available to our handlers.
type application struct {
    logger *slog.Logger
    snippets *models.SnippetModel
}

func main() {
    addr := flag.String("addr", ":4000", "HTTP network address")
    dsn := flag.String("dsn", "web:pass@snippetbox?parseTime=true", "MySQL data source name")
    flag.Parse()

    logger := slog.New(slog.NewTextHandler(os.Stdout, nil))

    db, err := openDB(*dsn)
    if err != nil {
        logger.Error(err.Error())
        os.Exit(1)
    }
    defer db.Close()

    // Initialize a models.SnippetModel instance containing the connection pool
    // and add it to the application dependencies.
    app := &application{
        logger: logger,
        snippets: &models.SnippetModel{DB: db},
    }

    logger.Info("starting server", "addr", *addr)

    err = http.ListenAndServe(*addr, app.routes())
    logger.Error(err.Error())
    os.Exit(1)
}

...
```

Additional information

Benefits of this structure

If you take a step back, you might be able to see a few benefits of setting up our project in this way:

- There's a clean separation of concerns. Our database logic won't be tied to our handlers, which means that handler responsibilities are limited to HTTP stuff (i.e. validating requests and writing responses). This will make it easier to write tight, focused, unit tests in the future.
- By creating a custom `SnippetModel` type and implementing methods on it we've been able to make our model a single, neatly encapsulated object, which we can easily initialize and then pass to our handlers as a dependency. Again, this makes for easier to maintain, testable code.
- Because the model actions are defined as methods on an object — in our case `SnippetModel` — there's the opportunity to create an *interface* and mock it for unit testing purposes.
- *And finally*, we have total control over which database is used at runtime, just by using the `-dsn` command-line flag.

Executing SQL statements

Now let's update the `SnippetModel.Insert()` method — which we've just made — so that it creates a new record in our `snippets` table and then returns the integer `id` for the new record.

To do this we'll want to execute the following SQL query on our database:

```
INSERT INTO snippets (title, content, created, expires)
VALUES(?, ?, UTC_TIMESTAMP(), DATE_ADD(UTC_TIMESTAMP(), INTERVAL ? DAY))
```

Notice how in this query we're using the `?` character to indicate *placeholder parameters* for the data that we want to insert in the database? Because the data we'll be using will ultimately be untrusted user input from a form, it's good practice to use placeholder parameters instead of interpolating data in the SQL query.

Executing the query

Go provides three different methods for executing database queries:

- `DB.Query()` is used for `SELECT` queries which return multiple rows.
- `DB.QueryRow()` is used for `SELECT` queries which return a single row.
- `DB.Exec()` is used for statements which don't return rows (like `INSERT` and `DELETE`).

So, in our case, the most appropriate tool for the job is `DB.Exec()`. Let's jump in the deep end and demonstrate how to use this in our `SnippetModel.Insert()` method. We'll discuss the details afterwards.

Open your `internal/models/snippets.go` file and update it like so:

```
File: internal/models/snippets.go
```

```
package models

...

type SnippetModel struct {
    DB *sql.DB
}

func (m *SnippetModel) Insert(title string, content string, expires int) (int, error) {
    // Write the SQL statement we want to execute. I've split it over two lines
    // for readability (which is why it's surrounded with backquotes instead
    // of normal double quotes).
    stmt := `INSERT INTO snippets (title, content, created, expires)
VALUES(?, ?, UTC_TIMESTAMP(), DATE_ADD(UTC_TIMESTAMP(), INTERVAL ? DAY))`

    // Use the Exec() method on the embedded connection pool to execute the
    // statement. The first parameter is the SQL statement, followed by the
    // values for the placeholder parameters: title, content and expiry in
    // that order. This method returns a sql.Result type, which contains some
    // basic information about what happened when the statement was executed.
    result, err := m.DB.Exec(stmt, title, content, expires)
    if err != nil {
        return 0, err
    }

    // Use the LastInsertId() method on the result to get the ID of our
    // newly inserted record in the snippets table.
    id, err := result.LastInsertId()
    if err != nil {
        return 0, err
    }

    // The ID returned has the type int64, so we convert it to an int type
    // before returning.
    return int(id), nil
}
```

Let's quickly discuss the `sql.Result` type returned by `DB.Exec()`. This provides two methods:

- `LastInsertId()` — which returns the integer (an `int64`) generated by the database in response to a command. Typically this will be from an “auto increment” column when inserting a new row, which is exactly what’s happening in our case.
- `RowsAffected()` — which returns the number of rows (as an `int64`) affected by the statement.

Important: Not all drivers and databases support the `LastInsertId()` and `RowsAffected()` methods. For example, `LastInsertId()` is not supported by PostgreSQL. So if you're planning on using these methods it's important to check the documentation for your particular driver first.

Also, it is perfectly acceptable (and common) to ignore the `sql.Result` return value if you don't need it. Like so:

```
_ , err := m.DB.Exec("INSERT INTO ...", ...)
```

Using the model in our handlers

Let's bring this back to something more concrete and demonstrate how to call this new code from our handlers. Open your `cmd/web/handlers.go` file and update the `snippetCreatePost` handler like so:

```
File: cmd/web/handlers.go

package main

...

func (app *application) snippetCreatePost(w http.ResponseWriter, r *http.Request) {
    // Create some variables holding dummy data. We'll remove these later on
    // during the build.
    title := "O snail"
    content := "O snail\nClimb Mount Fuji,\nBut slowly, slowly!\n\n- Kobayashi Issa"
    expires := 7

    // Pass the data to the SnippetModel.Insert() method, receiving the
    // ID of the new record back.
    id, err := app.snippets.Insert(title, content, expires)
    if err != nil {
        app.serverError(w, r, err)
        return
    }

    // Redirect the user to the relevant page for the snippet.
    http.Redirect(w, r, fmt.Sprintf("/snippet/view/%d", id), http.StatusSeeOther)
}
```

Start up the application, then open a second terminal window and use curl to make a `POST /snippet/create` request, like so (note that the `-L` flag instructs curl to automatically follow redirects):

```
$ curl -iL -d "" http://localhost:4000/snippet/create
HTTP/1.1 303 See Other
Location: /snippet/view/4
Date: Wed, 18 Mar 2024 11:29:23 GMT
Content-Length: 0

HTTP/1.1 200 OK
Date: Wed, 18 Mar 2024 11:29:23 GMT
Content-Length: 39
Content-Type: text/plain; charset=utf-8

Display a specific snippet with ID 4...
```

So this is working pretty nicely. We've just sent a HTTP request which triggered our `snippetCreatePost` handler, which in turn called our `SnippetModel.Insert()` method. This inserted a new record in the database and returned the ID of this new record. Our handler then issued a redirect to another URL with the ID interpolated.

Feel free to take a look in the `snippets` table of your MySQL database. You should see the new record with an ID of `4` similar to this:

```
mysql> SELECT id, title, expires FROM snippets;
+----+-----+-----+
| id | title          | expires        |
+----+-----+-----+
| 1  | An old silent pond | 2025-03-18 10:00:26 |
| 2  | Over the wintry forest | 2025-03-18 10:00:26 |
| 3  | First autumn morning | 2024-03-25 10:00:26 |
| 4  | O snail           | 2024-03-25 10:13:04 |
+----+-----+-----+
4 rows in set (0.00 sec)
```

Additional information

Placeholder parameters

In the code above we constructed our SQL statement using placeholder parameters, where `?` acted as a placeholder for the data we want to insert.

The reason for using placeholder parameters to construct our query (rather than string interpolation) is to help avoid SQL injection attacks from any untrusted user-provided input.

Behind the scenes, the `DB.Exec()` method works in three steps:

1. It creates a new **prepared statement** on the database using the provided SQL statement. The database parses and compiles the statement, then stores it ready for execution.
2. In a second separate step, **DB.Exec()** passes the parameter values to the database. The database then executes the prepared statement using these parameters. Because the parameters are transmitted later, after the statement has been compiled, the database treats them as pure data. They can't change the *intent* of the statement. So long as the original statement is not derived from untrusted data, injection cannot occur.
3. It then closes (or *deallocates*) the prepared statement on the database.

The placeholder parameter syntax differs depending on your database. MySQL, SQL Server and SQLite use the `?` notation, but PostgreSQL uses the `$N` notation. For example, if you were using PostgreSQL instead you would write:

```
_ , err := m.DB.Exec("INSERT INTO ... VALUES ($1, $2, $3)", ...)
```

Single-record SQL queries

The pattern for executing a `SELECT` statement to retrieve a single record from the database is a little more complicated. Let's explain how to do it by updating our `SnippetModel.Get()` method so that it returns a single specific snippet based on its ID.

To do this, we'll need to run the following SQL query on the database:

```
SELECT id, title, content, created, expires FROM snippets  
WHERE expires > UTC_TIMESTAMP() AND id = ?
```

Because our `snippets` table uses the `id` column as its primary key, this query will only ever return exactly one database row (or none at all). The query also includes a check on the expiry time so that we don't return any snippets that have expired.

Notice too that we're using a placeholder parameter again for the `id` value?

Open the `internal/models/snippets.go` file and add the following code:

```
File: internal/models/snippets.go
```

```
package models

import (
    "database/sql"
    "errors" // New import
    "time"
)

...

func (m *SnippetModel) Get(id int) (Snippet, error) {
    // Write the SQL statement we want to execute. Again, I've split it over two
    // lines for readability.
    stmt := `SELECT id, title, content, created, expires FROM snippets
WHERE expires > UTC_TIMESTAMP() AND id = ?`

    // Use the QueryRow() method on the connection pool to execute our
    // SQL statement, passing in the untrusted id variable as the value for the
    // placeholder parameter. This returns a pointer to a sql.Row object which
    // holds the result from the database.
    row := m.DB.QueryRow(stmt, id)

    // Initialize a new zeroed Snippet struct.
    var s Snippet

    // Use row.Scan() to copy the values from each field in sql.Row to the
    // corresponding field in the Snippet struct. Notice that the arguments
    // to row.Scan are *pointers* to the place you want to copy the data into,
    // and the number of arguments must be exactly the same as the number of
    // columns returned by your statement.
    err := row.Scan(&s.ID, &s.Title, &s.Content, &s.Created, &s.Expires)
    if err != nil {
        // If the query returns no rows, then row.Scan() will return a
        // sql.ErrNoRows error. We use the errors.Is() function check for that
        // error specifically, and return our own ErrNoRecord error
        // instead (we'll create this in a moment).
        if errors.Is(err, sql.ErrNoRows) {
            return Snippet{}, ErrNoRecord
        } else {
            return Snippet{}, err
        }
    }

    // If everything went OK, then return the filled Snippet struct.
    return s, nil
}

...
```

Behind the scenes of `rows.Scan()` your driver will automatically convert the raw output from the SQL database to the required native Go types. So long as you're sensible with the types that you're mapping between SQL and Go, these conversions should generally Just Work. Usually:

- `CHAR`, `VARCHAR` and `TEXT` map to `string`.
- `BOOLEAN` maps to `bool`.

- `INT` maps to `int`; `BIGINT` maps to `int64`.
- `DECIMAL` and `NUMERIC` map to `float`.
- `TIME`, `DATE` and `TIMESTAMP` map to `time.Time`.

Note: A quirk of our MySQL driver is that we need to use the `parseTime=true` parameter in our DSN to force it to convert `TIME` and `DATE` fields to `time.Time`. Otherwise it returns these as `[]byte` objects. This is one of the many [driver-specific parameters](#) that it offers.

If you try to run the application at this point, you should get a compile-time error saying that the `ErrNoRecord` value is undefined:

```
$ go run ./cmd/web/
# snippetbox.alexewards.net/internal/models
internal/models/snippets.go:82:25: undefined: ErrNoRecord
```

Let's go ahead and create that now in a new `internal/models/errors.go` file. Like so:

```
$ touch internal/models/errors.go
```

```
File: internal/models/errors.go

package models

import (
    "errors"
)

var ErrNoRecord = errors.New("models: no matching record found")
```

As an aside, you might be wondering why we're returning the `ErrNoRecord` error from our `SnippetModel.Get()` method, instead of `sql.ErrNoRows` directly. The reason is to help encapsulate the model completely, so that our handlers aren't concerned with the underlying datastore or reliant on datastore-specific errors (like `sql.ErrNoRows`) for its behavior.

Using the model in our handlers

Alright, let's put the `SnippetModel.Get()` method into action.

Open your `cmd/web/handlers.go` file and update the `snippetView` handler so that it returns the data for a specific record as a HTTP response:

```
File: cmd/web/handlers.go

package main

import (
    "errors" // New import
    "fmt"
    "html/template"
    "net/http"
    "strconv"

    "snippetbox.alexedwards.net/internal/models" // New import
)

...

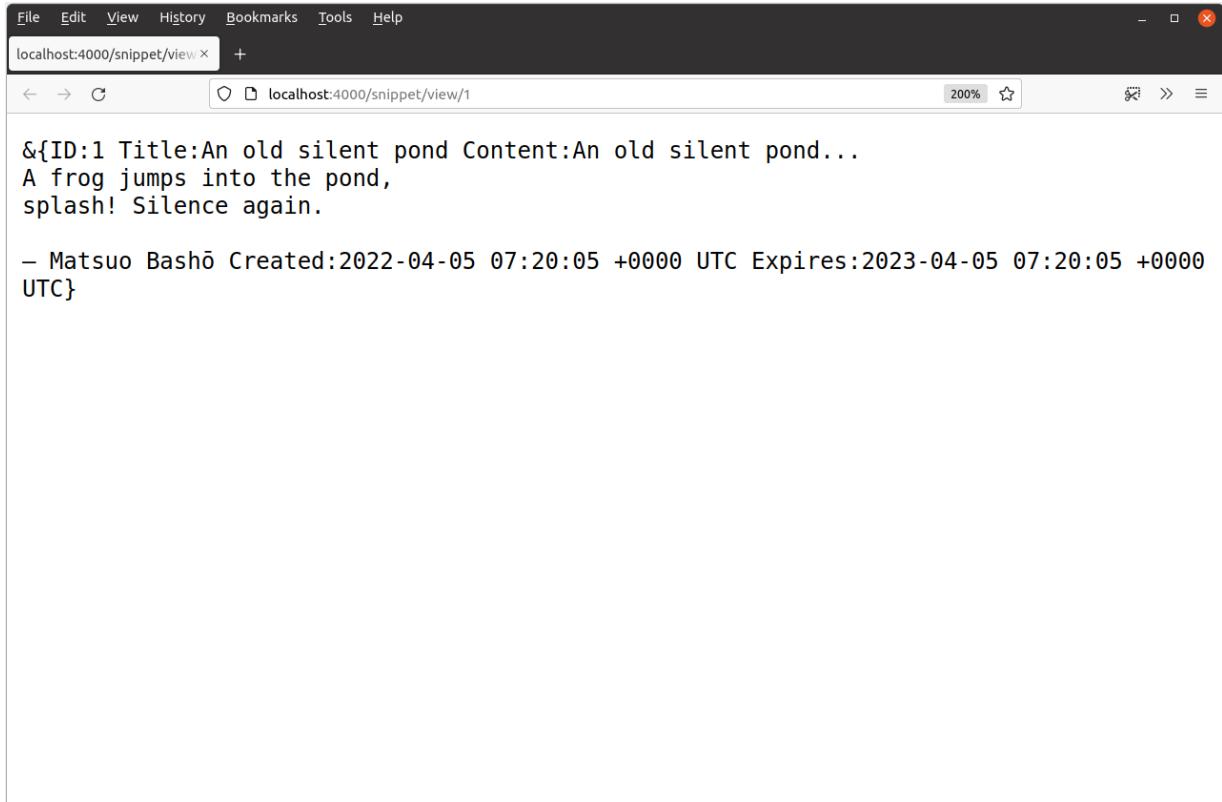
func (app *application) snippetView(w http.ResponseWriter, r *http.Request) {
    id, err := strconv.Atoi(r.PathValue("id"))
    if err != nil || id < 1 {
        http.NotFound(w, r)
        return
    }

    // Use the SnippetModel's Get() method to retrieve the data for a
    // specific record based on its ID. If no matching record is found,
    // return a 404 Not Found response.
    snippet, err := app.snippets.Get(id)
    if err != nil {
        if errors.Is(err, models.ErrNoRecord) {
            http.NotFound(w, r)
        } else {
            app.serverError(w, r, err)
        }
        return
    }

    // Write the snippet data as a plain-text HTTP response body.
    fmt.Fprintf(w, "%+v", snippet)
}

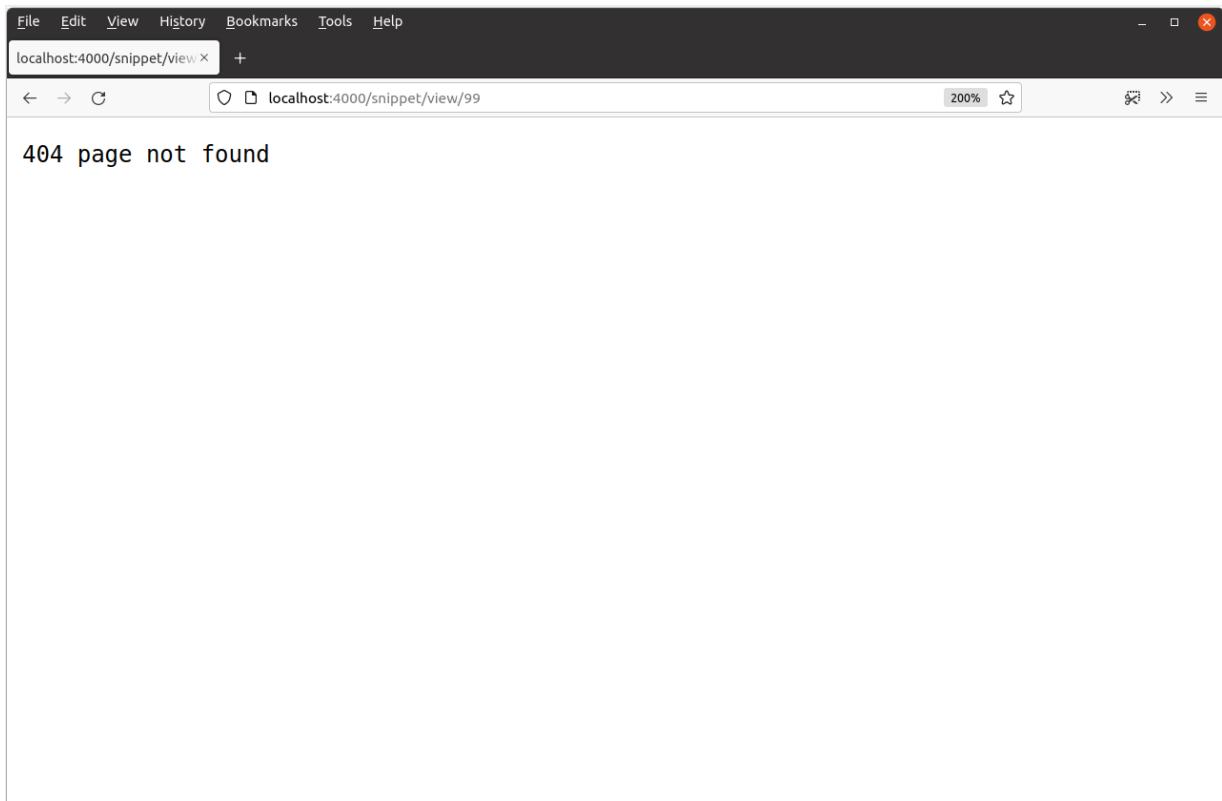
...
```

Let's give this a try. Restart the application, then open your web browser and visit <http://localhost:4000/snippet/view/1>. You should see a HTTP response which looks similar to this:



A screenshot of a web browser window. The title bar says "localhost:4000/snippet/view". The address bar also shows "localhost:4000/snippet/view/1". The content area displays a snippet with ID 1. The snippet's title is "An old silent pond" and its content is "A frog jumps into the pond, splash! Silence again.". Below the snippet, there is a timestamp and metadata: "– Matsuo Bashō Created:2022-04-05 07:20:05 +0000 UTC Expires:2023-04-05 07:20:05 +0000 UTC".

You might also want to try making some requests for other snippets which are expired or don't yet exist (like an `id` value of `99`) to verify that they return a `404 page not found` response:



A screenshot of a web browser window. The title bar says "localhost:4000/snippet/view". The address bar shows "localhost:4000/snippet/view/99". The content area displays the text "404 page not found".

Additional information

Checking for specific errors

A couple of times in this chapter we've used the `errors.Is()` function to check whether an error matches a specific value. Like this:

```
if errors.Is(err, models.ErrNoRecord) {
    http.NotFound(w, r)
} else {
    app.serverError(w, r, err)
}
```

In very old versions of Go (prior to 1.13), the idiomatic way to compare errors was to use the equality operator `==`, like so:

```
if err == models.ErrNoRecord {
    http.NotFound(w, r)
} else {
    app.serverError(w, r, err)
}
```

But, while this code still compiles, it's safer and best practice to use the `errors.Is()` function instead.

This is because Go 1.13 introduced the ability to add additional information to errors by *wrapping them*. If an error happens to get wrapped, a entirely new error value is created — which in turn means that it's not possible to check the value of the original underlying error using the regular `==` equality operator.

The `errors.Is()` function works by *unwrapping* errors as necessary before checking for a match.

There is also another function, `errors.As()` which you can use to check if a (potentially wrapped) error has a specific *type*. We'll use this later on this book.

Shorthand single-record queries

I've deliberately made the code in `SnippetModel.Get()` slightly long-winded to help clarify and emphasize what is going on behind the scenes of the code.

In practice, you can shorten the code slightly by leveraging the fact that errors from

`DB.QueryRow()` are deferred until `Scan()` is called. It makes no functional difference, but if you want it's perfectly OK to re-write the code to look something like this:

```
func (m *SnippetModel) Get(id int) (Snippet, error) {
    var s Snippet

    err := m.DB.QueryRow("SELECT ...", id).Scan(&s.ID, &s.Title, &s.Content, &s.Created, &s.Expires)
    if err != nil {
        if errors.Is(err, sql.ErrNoRows) {
            return Snippet{}, ErrNoRecord
        } else {
            return Snippet{}, err
        }
    }

    return s, nil
}
```

Multiple-record SQL queries

Finally let's look at the pattern for executing SQL statements which return multiple rows. I'll demonstrate this by updating the `SnippetModel.Latest()` method to return the ten *most-recently created snippets* (so long as they haven't expired) using the following SQL query:

```
SELECT id, title, content, created, expires FROM snippets  
WHERE expires > UTC_TIMESTAMP() ORDER BY id DESC LIMIT 10
```

Open up the `internal/models/snippets.go` file and add the following code:

File: internal/models/snippets.go

```
package models

...

func (m *SnippetModel) Latest() ([]Snippet, error) {
    // Write the SQL statement we want to execute.
    stmt := `SELECT id, title, content, created, expires FROM snippets
    WHERE expires > UTC_TIMESTAMP() ORDER BY id DESC LIMIT 10`

    // Use the Query() method on the connection pool to execute our
    // SQL statement. This returns a sql.Rows resultset containing the result of
    // our query.
    rows, err := m.DB.Query(stmt)
    if err != nil {
        return nil, err
    }

    // We defer rows.Close() to ensure the sql.Rows resultset is
    // always properly closed before the Latest() method returns. This defer
    // statement should come *after* you check for an error from the Query()
    // method. Otherwise, if Query() returns an error, you'll get a panic
    // trying to close a nil resultset.
    defer rows.Close()

    // Initialize an empty slice to hold the Snippet structs.
    var snippets []Snippet

    // Use rows.Next to iterate through the rows in the resultset. This
    // prepares the first (and then each subsequent) row to be acted on by the
    // rows.Scan() method. If iteration over all the rows completes then the
    // resultset automatically closes itself and frees-up the underlying
    // database connection.
    for rows.Next() {
        // Create a pointer to a new zeroed Snippet struct.
        var s Snippet
        // Use rows.Scan() to copy the values from each field in the row to the
        // new Snippet object that we created. Again, the arguments to row.Scan()
        // must be pointers to the place you want to copy the data into, and the
        // number of arguments must be exactly the same as the number of
        // columns returned by your statement.
        err = rows.Scan(&s.ID, &s.Title, &s.Content, &s.Created, &s.Expires)
        if err != nil {
            return nil, err
        }
        // Append it to the slice of snippets.
        snippets = append(snippets, s)
    }

    // When the rows.Next() loop has finished we call rows.Err() to retrieve any
    // error that was encountered during the iteration. It's important to
    // call this - don't assume that a successful iteration was completed
    // over the whole resultset.
    if err = rows.Err(); err != nil {
        return nil, err
    }

    // If everything went OK then return the Snippets slice.
    return snippets, nil
}
```

Important: Closing a resultset with `defer rows.Close()` is critical in the code above. As long as a resultset is open it will keep the underlying database connection open... so if something goes wrong in this method and the resultset isn't closed, it can rapidly lead to all the connections in your pool being used up.

Using the model in our handlers

Head back to your `cmd/web/handlers.go` file and update the `home` handler to use the `SnippetModel.Latest()` method, dumping the snippet contents to a HTTP response. For now just comment out the code relating to template rendering, like so:

File: cmd/web/handlers.go

```
package main

import (
    "errors"
    "fmt"
    // "html/template"
    "net/http"
    "strconv"

    "snippetbox.alexedwards.net/internal/models"
)

func (app *application) home(w http.ResponseWriter, r *http.Request) {
    w.Header().Add("Server", "Go")

    snippets, err := app.snippets.Latest()
    if err != nil {
        app.serverError(w, r, err)
        return
    }

    for _, snippet := range snippets {
        fmt.Fprintf(w, "%+v\n", snippet)
    }

    // files := []string{
    //     "./ui/html/base.tmpl",
    //     "./ui/html/partials/nav.tmpl",
    //     "./ui/html/pages/home.tmpl",
    // }

    // ts, err := template.ParseFiles(files...)
    // if err != nil {
    //     app.serverError(w, r, err)
    //     return
    // }

    // err = ts.ExecuteTemplate(w, "base", nil)
    // if err != nil {
    //     app.serverError(w, r, err)
    // }
}

...
```

If you run the application now and visit <http://localhost:4000> in your browser you should get a response similar to this:

```
File Edit View History Bookmarks Tools Help
localhost:4000 × +
← → ⌂ localhost:4000 200% ☆ ⌂ >> ⌂
&{ID:4 Title:0 snail Content:0 snail
Climb Mount Fuji,
But slowly, slowly!
-
– Kobayashi Issa Created:2022-04-05 07:32:45 +0000 UTC Expires:2022-04-12 07:32:45
+0000 UTC}
&{ID:1 Title:An old silent pond Content:An old silent pond...
A frog jumps into the pond,
splash! Silence again.
-
– Matsuo Bashō Created:2022-04-05 07:20:05 +0000 UTC Expires:2023-04-05 07:20:05 +0000
UTC}
&{ID:2 Title:Over the wintry forest Content:Over the wintry
forest, winds howl in rage
with no leaves to blow.
-
– Natsume Soseki Created:2022-04-05 07:20:05 +0000 UTC Expires:2023-04-05 07:20:05
+0000 UTC}
&{ID:3 Title:First autumn morning Content:First autumn morning
the mirror I stare into
shows my father's face.
-
– Murakami Kijo Created:2022-04-05 07:20:05 +0000 UTC Expires:2022-04-12 07:20:05
+0000 UTC}
```

Transactions and other details

The database/sql package

As you're probably starting to realize, the [database/sql](#) package essentially provides a standard interface between your Go application and the world of SQL databases.

So long as you use the [database/sql](#) package, the Go code you write will generally be portable and will work with any kind of SQL database — whether it's MySQL, PostgreSQL, SQLite or something else. This means that your application isn't so tightly coupled to the database that you're currently using, and the theory is that you can swap databases in the future without re-writing all of your code (driver-specific quirks and SQL implementations aside).

It's important to note that while [database/sql](#) generally does a good job of providing a standard interface for working with SQL databases, there *are* some idiosyncrasies in the way that different drivers and databases operate. It's always a good idea to read over the documentation for a new driver to understand any quirks and edge cases before you begin using it.

Verbosity

If you're coming from Ruby, Python or PHP, the code for querying SQL databases may feel a bit verbose, especially if you're used to dealing with an abstraction layer or ORM.

But the upside of the verbosity is that our code is non-magical; we can understand and control exactly what is going on. And with a bit of time, you'll find that the patterns for making SQL queries become familiar and you can copy-and-paste from previous work, or use developer tools like GitHub copilot to write the first draft of the code for you.

If the verbosity really is starting to grate on you, you might want to consider trying the [jmoiron/sqlx](#) package. It's well designed and provides some good extensions that make working with SQL queries quicker and easier. Another, newer, option you may want to consider is the [blockloop/scan](#) package.

Managing null values

One thing that Go doesn't do very well is managing `NULL` values in database records.

Let's pretend that the `title` column in our `snippets` table contains a `NULL` value in a particular row. If we queried that row, then `rows.Scan()` would return the following error because it can't convert `NULL` into a string:

```
sql: Scan error on column index 1: unsupported Scan, storing driver.Value type  
&lt;nil&gt; into type *string
```

Very roughly, the fix for this is to change the field that you're scanning into from a `string` to a `sql.NullString` type. See [this gist](#) for a working example.

But, as a rule, the easiest thing to do is simply avoid `NULL` values altogether. Set `NOT NULL` constraints on all your database columns, like we have done in this book, along with sensible `DEFAULT` values as necessary.

Working with transactions

It's important to realize that calls to `Exec()`, `Query()` and `QueryRow()` can use *any connection from the `sql.DB` pool*. Even if you have two calls to `Exec()` immediately next to each other in your code, there is no guarantee that they will use the same database connection.

Sometimes this isn't acceptable. For instance, if you lock a table with MySQL's `LOCK TABLES` command you must call `UNLOCK TABLES` on exactly the same connection to avoid a deadlock.

To guarantee that the same connection is used you can wrap multiple statements in a transaction. Here's the basic pattern:

```

type ExampleModel struct {
    DB *sql.DB
}

func (m *ExampleModel) ExampleTransaction() error {
    // Calling the Begin() method on the connection pool creates a new sql.Tx
    // object, which represents the in-progress database transaction.
    tx, err := m.DB.Begin()
    if err != nil {
        return err
    }

    // Defer a call to tx.Rollback() to ensure it is always called before the
    // function returns. If the transaction succeeds it will be already be
    // committed by the time tx.Rollback() is called, making tx.Rollback() a
    // no-op. Otherwise, in the event of an error, tx.Rollback() will rollback
    // the changes before the function returns.
    defer tx.Rollback()

    // Call Exec() on the transaction, passing in your statement and any
    // parameters. It's important to notice that tx.Exec() is called on the
    // transaction object just created, NOT the connection pool. Although we're
    // using tx.Exec() here you can also use tx.Query() and tx.QueryRow() in
    // exactly the same way.
    _, err = tx.Exec("INSERT INTO ...")
    if err != nil {
        return err
    }

    // Carry out another transaction in exactly the same way.
    _, err = tx.Exec("UPDATE ...")
    if err != nil {
        return err
    }

    // If there are no errors, the statements in the transaction can be committed
    // to the database with the tx.Commit() method.
    err = tx.Commit()
    return err
}

```

Important: You must *always* call either `Rollback()` or `Commit()` before your function returns. If you don't the connection will stay open and not be returned to the connection pool. This can lead to hitting your maximum connection limit/running out of resources. The simplest way to avoid this is to use `defer tx.Rollback()` like we are in the example above.

Transactions are also super-useful if you want to execute multiple SQL statements as a *single atomic action*. So long as you use the `tx.Rollback()` method in the event of any errors, the transaction ensures that either:

- *All* statements are executed successfully; or
- *No* statements are executed and the database remains unchanged.

Prepared statements

As I mentioned earlier, the `Exec()`, `Query()` and `QueryRow()` methods all use prepared statements behind the scenes to help prevent SQL injection attacks. They set up a prepared statement on the database connection, run it with the parameters provided, and then close the prepared statement.

This might feel rather inefficient because we are creating and recreating the same prepared statements every single time.

In theory, a better approach could be to make use of the `DB.Prepare()` method to create our own prepared statement once, and reuse that instead. This is particularly true for complex SQL statements (e.g. those which have multiple JOINS) *and* are repeated very often (e.g. a bulk insert of tens of thousands of records). In these instances, the cost of re-preparing statements may have a noticeable effect on run time.

Here's the basic pattern for using your own prepared statement in a web application:

```

// We need somewhere to store the prepared statement for the lifetime of our
// web application. A neat way is to embed it in the model alongside the
// connection pool.
type ExampleModel struct {
    DB          *sql.DB
    InsertStmt *sql.Stmt
}

// Create a constructor for the model, in which we set up the prepared
// statement.
func NewExampleModel(db *sql.DB) (*ExampleModel, error) {
    // Use the Prepare method to create a new prepared statement for the
    // current connection pool. This returns a sql.Stmt object which represents
    // the prepared statement.
    insertStmt, err := db.Prepare("INSERT INTO ...")
    if err != nil {
        return nil, err
    }

    // Store it in our ExampleModel struct, alongside the connection pool.
    return &ExampleModel{DB: db, InsertStmt: insertStmt}, nil
}

// Any methods implemented against the ExampleModel struct will have access to
// the prepared statement.
func (m *ExampleModel) Insert(args...) error {
    // We then need to call Exec directly against the prepared statement, rather
    // than against the connection pool. Prepared statements also support the
    // Query and QueryRow methods.
    _, err := m.InsertStmt.Exec(args...)

    return err
}

// In the web application's main function we will need to initialize a new
// ExampleModel struct using the constructor function.
func main() {
    db, err := sql.Open(...)
    if err != nil {
        logger.Error(err.Error())
        os.Exit(1)
    }
    defer db.Close()

    // Use the constructor function to create a new ExampleModel struct.
    exampleModel, err := NewExampleModel(db)
    if err != nil {
        logger.Error(err.Error())
        os.Exit(1)
    }

    // Defer a call to Close() on the prepared statement to ensure that it is
    // properly closed before our main function terminates.
    defer exampleModel.InsertStmt.Close()
}

```

There are a few things to be wary of though.

Prepared statements exist on *database connections*. So, because Go uses a pool of *many database connections*, what actually happens is that the first time a prepared statement (i.e. the `sql.Stmt` object) is used it gets created on a particular database connection. The

`sql.Stmt` object then remembers which connection in the pool was used. The next time, the `sql.Stmt` object will attempt to use the same database connection again. If that connection is closed or in use (i.e. not idle) the statement will be re-prepared on another connection.

Under heavy load, it's possible that a large number of prepared statements will be created on multiple connections. This can lead to statements being prepared and re-prepared more often than you would expect — or even running into server-side limits on the number of statements (in MySQL the default maximum is 16,382 prepared statements).

The code is also more complicated than not using prepared statements.

So, there is a trade-off to be made between performance and complexity. As with anything, you should measure the actual performance benefit of implementing your own prepared statements to determine if it's worth doing. For most cases, I would suggest that using the regular `Query()`, `QueryRow()` and `Exec()` methods — without preparing statements yourself — is a reasonable starting point.

Dynamic HTML templates

In this section of the book we're going to concentrate on displaying the dynamic data from our MySQL database in some proper HTML pages.

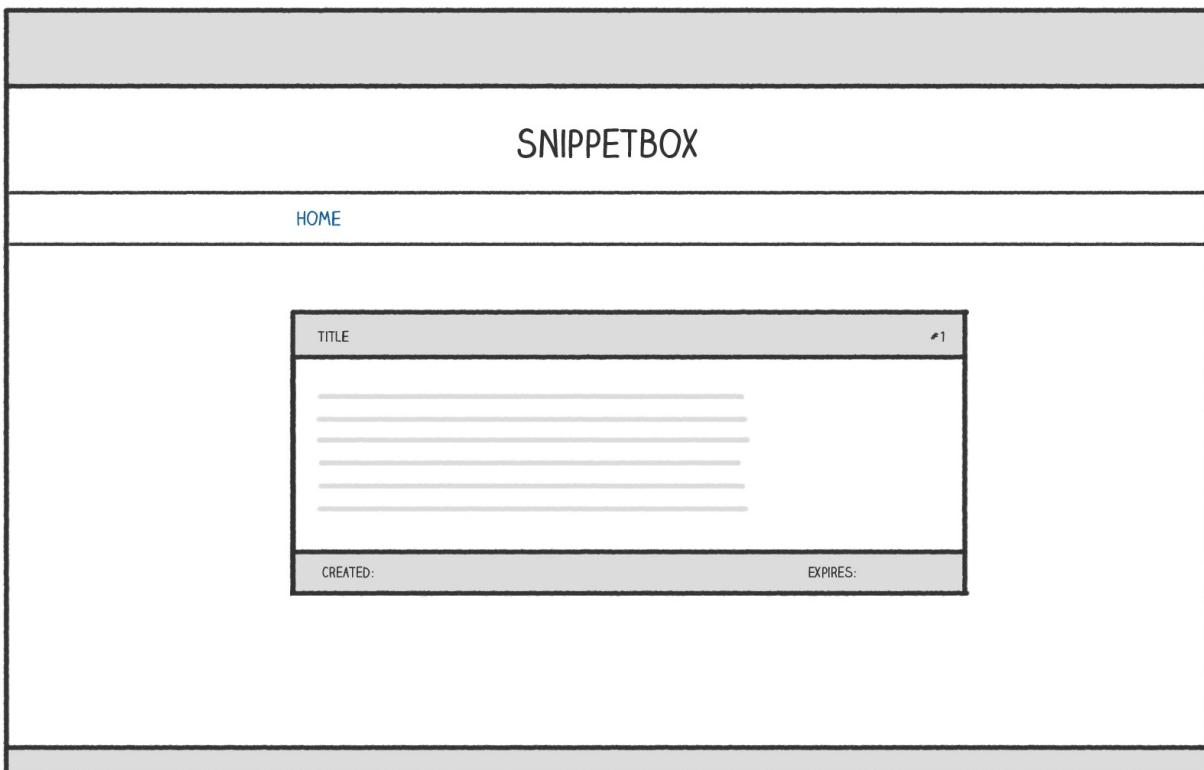
You'll learn how to:

- Pass [dynamic data](#) to your HTML templates in a simple, scalable and type-safe way.
- Use the various [actions and functions](#) in Go's [html/template](#) package to control the display of dynamic data.
- Create a [template cache](#) so that your templates aren't being read from disk and parsed for each HTTP request.
- Gracefully handle [template rendering errors](#) at runtime.
- Implement a pattern for passing [common dynamic data](#) to your web pages without repeating code.
- Create your own [custom functions](#) to format and display data in your HTML templates.

Displaying dynamic data

Currently our `snippetView` handler function fetches a `models.Snippet` object from the database and then dumps the contents out in a plain-text HTTP response.

In this chapter we'll improve this so that the data is displayed in a proper HTML webpage which looks a bit like this:



Let's start in the `snippetView` handler and add some code to render a new `view tmpl` template file (which we will create in a minute). Hopefully this should look familiar to you from earlier in the book.

File: cmd/web/handlers.go

```
package main

import (
    "errors"
    "fmt"
    "html/template" // Uncomment import
    "net/http"
    "strconv"

    "snippetbox.alexedwards.net/internal/models"
)

...

func (app *application) snippetView(w http.ResponseWriter, r *http.Request) {
    id, err := strconv.Atoi(r.PathValue("id"))
    if err != nil || id < 1 {
        http.NotFound(w, r)
        return
    }

    snippet, err := app.snippets.Get(id)
    if err != nil {
        if errors.Is(err, models.ErrNoRecord) {
            http.NotFound(w, r)
        } else {
            app.serverError(w, r, err)
        }
        return
    }

    // Initialize a slice containing the paths to the view tmpl file,
    // plus the base layout and navigation partial that we made earlier.
    files := []string{
        "./ui/html/base.tmpl",
        "./ui/html/partials/nav.tmpl",
        "./ui/html/pages/view.tmpl",
    }

    // Parse the template files...
    ts, err := template.ParseFiles(files...)
    if err != nil {
        app.serverError(w, r, err)
        return
    }

    // And then execute them. Notice how we are passing in the snippet
    // data (a models.Snippet struct) as the final parameter?
    err = ts.ExecuteTemplate(w, "base", snippet)
    if err != nil {
        app.serverError(w, r, err)
    }
}

...


```

Next up we need to create the `view.tmpl` file containing the HTML markup for the page. But before we do, there's a little theory that I need to explain...

Any data that you pass as the final parameter to `ts.ExecuteTemplate()` is represented within your HTML templates by the `.` character (referred to as *dot*).

In this specific case, the underlying type of dot will be a `models.Snippet` struct. When the underlying type of dot is a struct, you can render (or *yield*) the value of any exported field in your templates by postfixing dot with the field name. So, because our `models.Snippet` struct has a `Title` field, we could yield the snippet title by writing `{{.Title}}` in our templates.

I'll demonstrate. Create a new file at `ui/html/pages/view.tpl` and add the following markup:

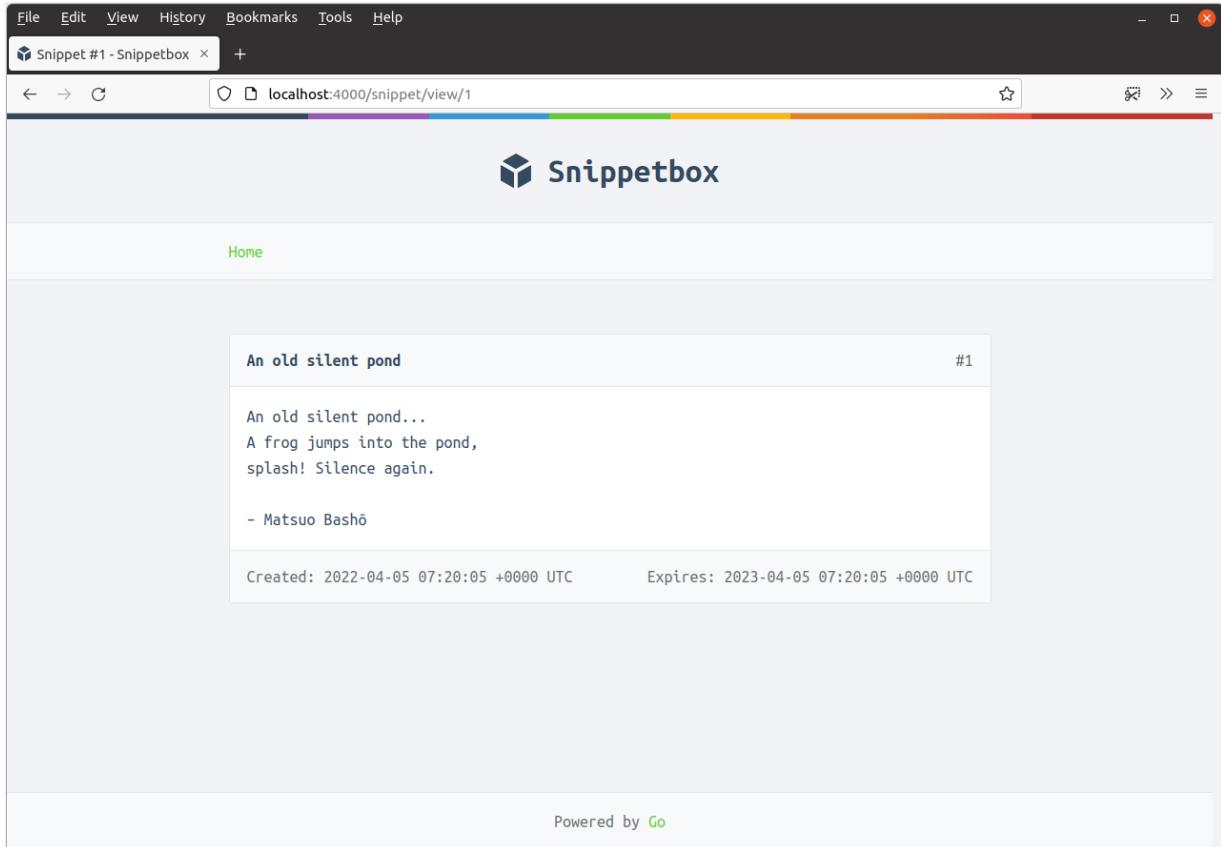
```
$ touch ui/html/pages/view.tpl
```

```
File: ui/html/pages/view.tpl

{{define "title"}}Snippet #{{.ID}}{{end}}

{{define "main"}}
<div class='snippet'>
  <div class='metadata'>
    <strong>{{.Title}}</strong>
    <span>#{{.ID}}</span>
  </div>
  <pre><code>{{.Content}}</code></pre>
  <div class='metadata'>
    <time>Created: {{.Created}}</time>
    <time>Expires: {{.Expires}}</time>
  </div>
</div>
{{end}}
```

If you restart the application and visit <http://localhost:4000/snippet/view/1> in your browser, you should find that the relevant snippet is fetched from the database, passed to the template, and the content is rendered correctly.



Rendering multiple pieces of data

An important thing to explain is that Go's `html/template` package allows you to pass in one — and only one — item of dynamic data when rendering a template. But in a real-world application there are often multiple pieces of dynamic data that you want to display in the same page.

A lightweight and type-safe way to achieve this is to wrap your dynamic data in a struct which acts like a single ‘holding structure’ for your data.

Let's create a new `cmd/web/templates.go` file, containing a `templateData` struct to do exactly that.

```
$ touch cmd/web/templates.go
```

```
File: cmd/web/templates.go
```

```
package main

import "snippetbox.alexedwards.net/internal/models"

// Define a templateData type to act as the holding structure for
// any dynamic data that we want to pass to our HTML templates.
// At the moment it only contains one field, but we'll add more
// to it as the build progresses.
type templateData struct {
    Snippet models.Snippet
}
```

And then let's update the `snippetView` handler to use this new struct when executing our templates:

File: cmd/web/handlers.go

```
package main

...

func (app *application) snippetView(w http.ResponseWriter, r *http.Request) {
    id, err := strconv.Atoi(r.PathValue("id"))
    if err != nil || id < 1 {
        http.NotFound(w, r)
        return
    }

    snippet, err := app.snippets.Get(id)
    if err != nil {
        if errors.Is(err, models.ErrNoRecord) {
            http.NotFound(w, r)
        } else {
            app.serverError(w, r, err)
        }
        return
    }

    files := []string{
        "./ui/html/base tmpl",
        "./ui/html/partials/nav tmpl",
        "./ui/html/pages/view tmpl",
    }

    ts, err := template.ParseFiles(files...)
    if err != nil {
        app.serverError(w, r, err)
        return
    }

    // Create an instance of a templateData struct holding the snippet data.
    data := templateData{
        Snippet: snippet,
    }

    // Pass in the templateData struct when executing the template.
    err = ts.ExecuteTemplate(w, "base", data)
    if err != nil {
        app.serverError(w, r, err)
    }
}

...


```

So now, our snippet data is contained in a `models.Snippet` struct *within* a `templateData` struct. To yield the data, we need to chain the appropriate field names together like so:

```
File: ui/html/pages/view.tpl
```

```
{{define "title"}}Snippet #{{.Snippet.ID}}{{end}}
```

```
{{define "main"}}


<div class='metadata'>
    <strong>{{.Snippet.Title}}</strong>
    <span>#{{.Snippet.ID}}</span>
</div>
<pre><code>{{.Snippet.Content}}</code></pre>
<div class='metadata'>
    <time>Created: {{.Snippet.Created}}</time>
    <time>Expires: {{.Snippet.Expires}}</time>
</div>
</div>
{{end}}


```

Feel free to restart the application and visit <http://localhost:4000/snippet/view/1> again. You should see the same page rendered in your browser as before.

Additional information

Dynamic content escaping

The `html/template` package automatically escapes any data that is yielded between `{{ }}` tags. This behavior is hugely helpful in avoiding cross-site scripting (XSS) attacks, and is the reason that you should use the `html/template` package instead of the more generic `text/template` package that Go also provides.

As an example of escaping, if the dynamic data you wanted to yield was:

```
<span>{{"<script>alert('xss attack')</script>"}}</span>
```

It would be rendered harmlessly as:

```
<span>&lt;script&gt;alert(&#39;xss attack&#39;)&lt;/script&gt;</span>
```

The `html/template` package is also smart enough to make escaping context-dependent. It will use the appropriate escape sequences depending on whether the data is rendered in a part of the page that contains HTML, CSS, Javascript or a URI.

Nested templates

It's really important to note that when you're invoking one template from another template, dot needs to be explicitly passed or *pipelined* to the template being invoked. You do this by including it at the end of each `{{template}}` or `{{block}}` action, like so:

```
 {{template "main" .}}
 {{block "sidebar" .}}{{end}}
```

As a general rule, my advice is to get into the habit of always pipelining dot whenever you invoke a template with the `{{template}}` or `{{block}}` actions, unless you have a good reason not to.

Calling methods

If the type that you're yielding between `{{ }}` tags has methods defined against it, you can call these methods (so long as they are exported and they return only a single value — or a single value and an error).

For example, our `.Snippet.Created` struct field has the underlying type `time.Time`, meaning that you could render the name of the weekday by calling its `Weekday()` method like so:

```
<span>{{.Snippet.Created.Weekday}}</span>
```

You can also pass parameters to methods. For example, you could use the `AddDate()` method to add six months to a time like so:

```
<span>{{.Snippet.Created.AddDate 0 6 0}}</span>
```

Notice that this is different syntax to calling functions in Go — the parameters are *not* surrounded by parentheses and are separated by a single space character, not a comma.

HTML comments

Finally, the `html/template` package always strips out any HTML comments you include in your templates, including any `conditional comments`.

The reason for this is to help avoid XSS attacks when rendering dynamic content. Allowing conditional comments would mean that Go isn't always able to anticipate how a browser

will interpret the markup in a page, and therefore it wouldn't necessarily be able to escape everything appropriately. To solve this, Go simply strips out *all* HTML comments.

Template actions and functions

In this section we're going to look at the template actions and functions that Go provides.

We've already talked about some of the actions — `{{define}}`, `{{template}}` and `{{block}}` — but there are three more which you can use to control the display of dynamic data — `{{if}}`, `{{with}}` and `{{range}}`.

Action	Description
<code>{{if .Foo}} C1 {{else}} C2 {{end}}</code>	If <code>.Foo</code> is not empty then render the content <code>C1</code> , otherwise render the content <code>C2</code> .
<code>{{with .Foo}} C1 {{else}} C2 {{end}}</code>	If <code>.Foo</code> is not empty, then set dot to the value of <code>.Foo</code> and render the content <code>C1</code> , otherwise render the content <code>C2</code> .
<code>{{range .Foo}} C1 {{else}} C2 {{end}}</code>	If the length of <code>.Foo</code> is greater than zero then loop over each element, setting dot to the value of each element and rendering the content <code>C1</code> . If the length of <code>.Foo</code> is zero then render the content <code>C2</code> . The underlying type of <code>.Foo</code> must be an array, slice, map, or channel.

There are a few things about these actions to point out:

- For all three actions the `{{else}}` clause is optional. For instance, you can write `{{if .Foo}} C1 {{end}}` if there's no `C2` content that you want to render.
- The *empty* values are false, 0, any nil pointer or interface value, and any array, slice, map, or string of length zero.
- It's important to grasp that the `with` and `range` actions change the value of dot. Once you start using them, *what dot represents can be different depending on where you are in the template and what you're doing*.

The `html/template` package also provides some template functions which you can use to add extra logic to your templates and control what is rendered at runtime. You can find a complete listing of functions [here](#), but the most important ones are:

Function	Description
<code>{{eq .Foo .Bar}}</code>	Yields true if <code>.Foo</code> is equal to <code>.Bar</code>
<code>{{ne .Foo .Bar}}</code>	Yields true if <code>.Foo</code> is not equal to <code>.Bar</code>
<code>{{not .Foo}}</code>	Yields the boolean negation of <code>.Foo</code>
<code>{{or .Foo .Bar}}</code>	Yields <code>.Foo</code> if <code>.Foo</code> is not empty; otherwise yields <code>.Bar</code>
<code>{{index .Foo i}}</code>	Yields the value of <code>.Foo</code> at index <code>i</code> . The underlying type of <code>.Foo</code> must be a map, slice or array, and <code>i</code> must be an integer value.
<code>{{printf "%s-%s" .Foo .Bar}}</code>	Yields a formatted string containing the <code>.Foo</code> and <code>.Bar</code> values. Works in the same way as <code>fmt.Sprintf()</code> .
<code>{{len .Foo}}</code>	Yields the length of <code>.Foo</code> as an integer.
<code>{{\$bar := len .Foo}}</code>	Assign the length of <code>.Foo</code> to the template variable <code>\$bar</code>

The final row is an example of declaring a template variable. Template variables are particularly useful if you want to store the result from a function and use it in multiple places in your template. Variable names must be prefixed by a dollar sign and can contain alphanumeric characters only.

Using the `with` action

A good opportunity to use the `{{with}}` action is the `view tmpl` file that we created in the previous chapter. Go ahead and update it like so:

File: ui/html/pages/view.tpl

```
 {{define "title"}}Snippet #{{.Snippet.ID}}{{end}}
```

```
 {{define "main"}}
 {{with .Snippet}}
 <div class='snippet'>
 <div class='metadata'>
 <strong>{{.Title}}</strong>
 <span>#{{.ID}}</span>
 </div>
 <pre><code>{{.Content}}</code></pre>
 <div class='metadata'>
 <time>Created: {{.Created}}</time>
 <time>Expires: {{.Expires}}</time>
 </div>
 </div>
 {{end}}
 {{end}}
```

So now, between `{{with .Snippet}}` and the corresponding `{{end}}` tag, the value of dot is set to `.Snippet`. Dot essentially becomes the `models.Snippet` struct instead of the parent `templateData` struct.

Using the if and range actions

Let's also use the `{{if}}` and `{{range}}` actions in a concrete example and update our homepage to display a table of the latest snippets, a bit like this:

The screenshot shows a web page titled "SNIPPETBOX". A navigation bar at the top has a single item labeled "HOME". Below the navigation bar, the page title "SNIPPETBOX" is centered. Underneath the title, the text "LATEST SNIPPETS" is displayed. A table is shown below this text, containing four rows of data. The table has three columns: "TITLE", "CREATED", and "ID". The data in the table is as follows:

TITLE	CREATED	ID
LOREM IPSUM DOLOR SIT AMET	2017-08-21	4
LOREM IPSUM DOLOR SIT AMET	2017-08-21	3
LOREM IPSUM DOLOR SIT AMET	2017-08-21	2
LOREM IPSUM DOLOR SIT AMET	2017-08-21	1

First update the `templateData` struct so that it contains a `Snippets` field for holding a slice of snippets, like so:

```
File: cmd/web/templates.go

package main

import "snippetbox.alexewards.net/internal/models"

// Include a Snippets field in the templateData struct.
type templateData struct {
    Snippet models.Snippet
    Snippets []models.Snippet
}
```

Then update the `home` handler function so that it fetches the latest snippets from our database model and passes them to the `home tmpl` template:

```
File: cmd/web/handlers.go
```

```
package main

...

func (app *application) home(w http.ResponseWriter, r *http.Request) {
    w.Header().Add("Server", "Go")

    snippets, err := app.snippets.Latest()
    if err != nil {
        app.serverError(w, r, err)
        return
    }

    files := []string{
        "./ui/html/base tmpl",
        "./ui/html/partials/nav tmpl",
        "./ui/html/pages/home tmpl",
    }
}

ts, err := template.ParseFiles(files...)
if err != nil {
    app.serverError(w, r, err)
    return
}

// Create an instance of a templateData struct holding the slice of
// snippets.
data := templateData{
    Snippets: snippets,
}

// Pass in the templateData struct when executing the template.
err = ts.ExecuteTemplate(w, "base", data)
if err != nil {
    app.serverError(w, r, err)
}
}

...


```

Now let's head over to the `ui/html/pages/home tmpl` file and update it to display these snippets in a table using the `{{if}}` and `{{range}}` actions. Specifically:

- We want to use the `{{if}}` action to check whether the slice of snippets is empty or not. If it's empty, we want to display a "There's nothing to see here yet!" message. Otherwise, we want to render a table containing the snippet information.
- We want to use the `{{range}}` action to iterate over all snippets in the slice, rendering the contents of each snippet in a table row.

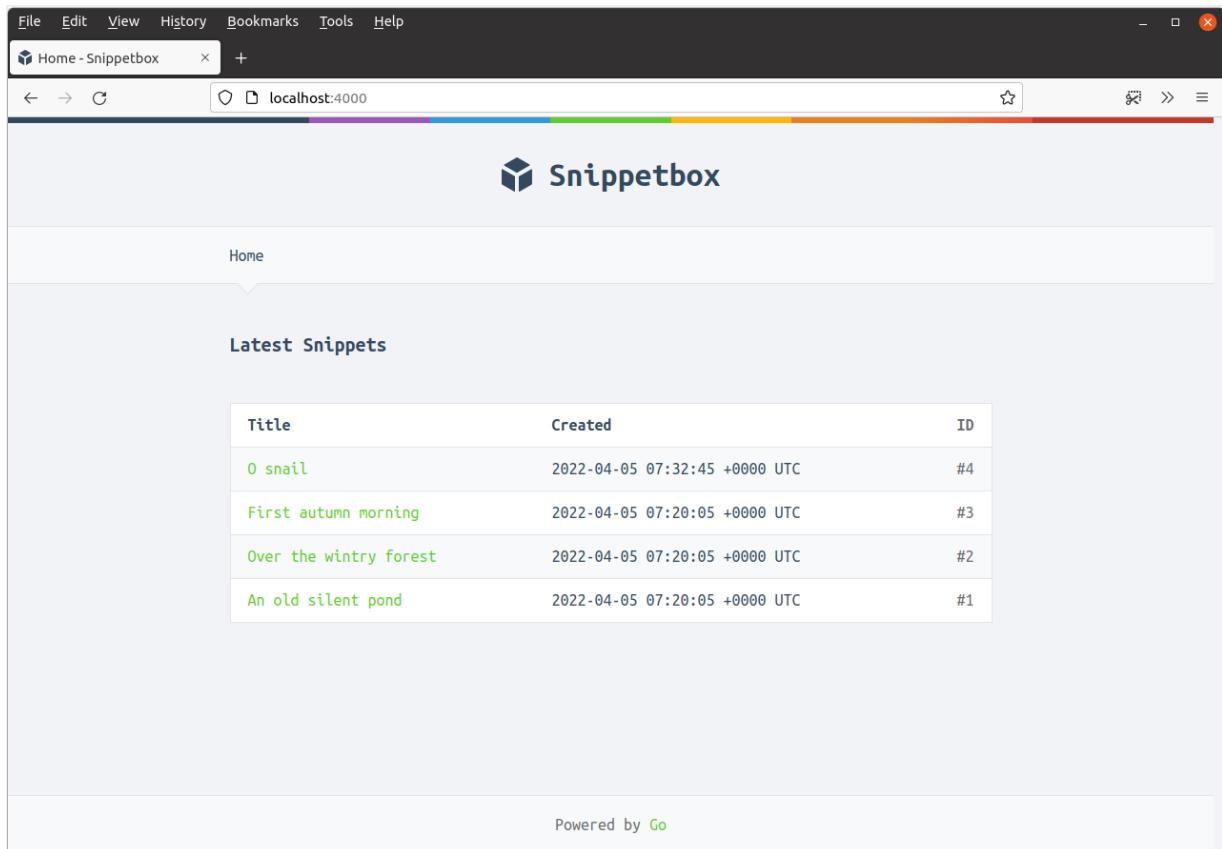
Here's the markup:

```
File: ui/html/pages/home.tpl
```

```
{{define "title"}}Home{{end}}
```

```
 {{define "main"}}
    <h2>Latest Snippets</h2>
    {{if .Snippets}}
        <table>
            <tr>
                <th>Title</th>
                <th>Created</th>
                <th>ID</th>
            </tr>
            {{range .Snippets}}
                <tr>
                    <td><a href='/snippet/view/{{.ID}}'>{{.Title}}</a></td>
                    <td>{{.Created}}</td>
                    <td>#{{.ID}}</td>
                </tr>
            {{end}}
        </table>
    {{else}}
        <p>There's nothing to see here... yet!</p>
    {{end}}
{{end}}
```

Make sure all your files are saved, restart the application and visit <http://localhost:4000> in a web browser. If everything has gone to plan, you should see a page which looks a bit like this:



Additional information

Combining functions

It's possible to combine multiple functions in your template tags, using the parentheses () to surround the functions and their arguments as necessary.

For example, the following tag will render the content **C1** if the length of **Foo** is greater than 99:

```
{if (gt (len .Foo) 99)} C1 {{end}}
```

Or as another example, the following tag will render the content **C1** if **.Foo** equals 1 *and* **.Bar** is less than or equal to 20:

```
{if (and (eq .Foo 1) (le .Bar 20))} C1 {{end}}
```

Controlling loop behavior

Within a **{{range}}** action you can use the **{{break}}** command to end the loop early, and **{{continue}}** to immediately start the next loop iteration.

```
 {{range .Foo}}
    // Skip this iteration if the .ID value equals 99.
    {{if eq .ID 99}}
        {{continue}}
    {{end}}
    // ...
{{end}}
```

```
 {{range .Foo}}
    // End the loop if the .ID value equals 99.
    {{if eq .ID 99}}
        {{break}}
    {{end}}
    // ...
{{end}}
```

Caching templates

Before we add any more functionality to our HTML templates, it's a good time to make some optimizations to our codebase. There are two main issues at the moment:

1. Each and every time we render a web page, our application reads and parses the relevant template files using the `template.ParseFiles()` function. We could avoid this duplicated work by parsing the files once — when starting the application — and storing the parsed templates in an in-memory cache.
2. There's duplicated code in the `home` and `snippetView` handlers, and we could reduce this duplication by creating a helper function.

Let's tackle the first point first, and create an in-memory map with the type `map[string]*template.Template` to cache the parsed templates. Open your `cmd/web/templates.go` file and add the following code:

```
File: cmd/web/templates.go
```

```
package main

import (
    "html/template" // New import
    "path/filepath" // New import

    "snippetbox.alexedwards.net/internal/models"
)

...

func newTemplateCache() (map[string]*template.Template, error) {
    // Initialize a new map to act as the cache.
    cache := map[string]*template.Template{}

    // Use the filepath.Glob() function to get a slice of all filepaths that
    // match the pattern "./ui/html/pages/*.tmpl". This will essentially give
    // us a slice of all the filepaths for our application 'page' templates
    // like: [ui/html/pages/home.tmpl ui/html/pages/view.tmpl]
    pages, err := filepath.Glob("./ui/html/pages/*.tmpl")
    if err != nil {
        return nil, err
    }

    // Loop through the page filepaths one-by-one.
    for _, page := range pages {
        // Extract the file name (like 'home.tmpl') from the full filepath
        // and assign it to the name variable.
        name := filepath.Base(page)

        // Create a slice containing the filepaths for our base template, any
        // partials and the page.
        files := []string{
            "./ui/html/base.tmpl",
            "./ui/html/partials/nav.tmpl",
            page,
        }

        // Parse the files into a template set.
        ts, err := template.ParseFiles(files...)
        if err != nil {
            return nil, err
        }

        // Add the template set to the map, using the name of the page
        // (like 'home.tmpl') as the key.
        cache[name] = ts
    }

    // Return the map.
    return cache, nil
}
```

The next step is to initialize this cache in the `main()` function and make it available to our handlers as a dependency via the `application` struct, like this:

File: cmd/web/main.go

```
package main

import (
    "database/sql"
    "flag"
    "html/template" // New import
    "log/slog"
    "net/http"
    "os"

    "snippetbox.alexedwards.net/internal/models"

    _ "github.com/go-sql-driver/mysql"
)

// Add a templateCache field to the application struct.
type application struct {
    logger      *slog.Logger
    snippets    *models.SnippetModel
    templateCache map[string]*template.Template
}

func main() {
    addr := flag.String("addr", ":4000", "HTTP network address")
    dsn := flag.String("dsn", "web:pass@/snippetbox?parseTime=true", "MySQL data source name")
    flag.Parse()

    logger := slog.New(slog.NewTextHandler(os.Stdout, nil))

    db, err := openDB(*dsn)
    if err != nil {
        logger.Error(err.Error())
        os.Exit(1)
    }
    defer db.Close()

    // Initialize a new template cache...
    templateCache, err := newTemplateCache()
    if err != nil {
        logger.Error(err.Error())
        os.Exit(1)
    }

    // And add it to the application dependencies.
    app := &application{
        logger:      logger,
        snippets:    &models.SnippetModel{DB: db},
        templateCache: templateCache,
    }

    logger.Info("starting server", "addr", *addr)

    err = http.ListenAndServe(*addr, app.routes())
    logger.Error(err.Error())
    os.Exit(1)
}

...
```

So, at this point, we've got an in-memory cache of the relevant template set for each of our

pages, and our handlers have access to this cache via the `application` struct.

Let's now tackle the second issue of duplicated code, and create a helper method so that we can easily render the templates from the cache.

Open up your `cmd/web/helpers.go` file and add the following `render()` method:

```
File: cmd/web/helpers.go

package main

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

...

func (app *application) render(w http.ResponseWriter, r *http.Request, status int, page string, data templateData) {
    // Retrieve the appropriate template set from the cache based on the page
    // name (like 'home.tmpl'). If no entry exists in the cache with the
    // provided name, then create a new error and call the serverError() helper
    // method that we made earlier and return.
    ts, ok := app.templateCache[page]
    if !ok {
        err := fmt.Errorf("the template %s does not exist", page)
        app.serverError(w, r, err)
        return
    }

    // Write out the provided HTTP status code ('200 OK', '400 Bad Request' etc).
    w.WriteHeader(status)

    // Execute the template set and write the response body. Again, if there
    // is any error we call the serverError() helper.
    err := ts.ExecuteTemplate(w, "base", data)
    if err != nil {
        app.serverError(w, r, err)
    }
}
```

With that complete, we now get to see the payoff from these changes and can dramatically simplify the code in our handlers:

File: cmd/web/handlers.go

```
package main

import (
    "errors"
    "fmt"
    "net/http"
    "strconv"

    "snippetbox.alexedwards.net/internal/models"
)

func (app *application) home(w http.ResponseWriter, r *http.Request) {
    w.Header().Add("Server", "Go")

    snippets, err := app.snippets.Latest()
    if err != nil {
        app.serverError(w, r, err)
        return
    }

    // Use the new render helper.
    app.render(w, r, http.StatusOK, "home tmpl", templateData{
        Snippets: snippets,
    })
}

func (app *application) snippetView(w http.ResponseWriter, r *http.Request) {
    id, err := strconv.Atoi(r.PathValue("id"))
    if err != nil || id < 1 {
        http.NotFound(w, r)
        return
    }

    snippet, err := app.snippets.Get(id)
    if err != nil {
        if errors.Is(err, models.ErrNoRecord) {
            http.NotFound(w, r)
        } else {
            app.serverError(w, r, err)
        }
        return
    }

    // Use the new render helper.
    app.render(w, r, http.StatusOK, "view tmpl", templateData{
        Snippet: snippet,
    })
}

...
```

If you restart the application and try visiting <http://localhost:4000> and <http://localhost:4000/snippet/view/1> again you should see that the pages are rendered in exactly the same way as before.

A screenshot of a web browser window titled "Home - Snippetbox". The address bar shows "localhost:4000". The page content includes the Snippetbox logo, a "Home" link, and a section titled "Latest Snippets" containing a table with four rows of snippet data. At the bottom, it says "Powered by Go".

Title	Created	ID
O snail	2022-04-05 07:32:45 +0000 UTC	#4
First autumn morning	2022-04-05 07:20:05 +0000 UTC	#3
Over the wintry forest	2022-04-05 07:20:05 +0000 UTC	#2

A screenshot of a web browser window titled "Snippet #1 - Snippetbox". The address bar shows "localhost:4000/snippet/view/1". The page content includes the Snippetbox logo, a "Home" link, and a snippet detail card for "An old silent pond". The card shows the snippet text, author information, and creation/expiration dates.

An old silent pond #1

An old silent pond...
A frog jumps into the pond,
splash! Silence again.

- Matsuo Bashō

Created: 2022-04-05 07:20:05 +0000 UTC Expires: 2023-04-05 07:20:05 +0000 UTC

Automatically parsing partials

Before we move on, let's make our `newTemplateCache()` function a bit more flexible so that it automatically parses *all templates in the ui/html/partials folder* — rather than only our `nav.tpl` file.

This will save us time, typing and potential bugs if we want to add additional partials in the future.

```
File: cmd/web/templates.go

package main

...

func newTemplateCache() (map[string]*template.Template, error) {
    cache := map[string]*template.Template{}

    pages, err := filepath.Glob("./ui/html/pages/*.tmpl")
    if err != nil {
        return nil, err
    }

    for _, page := range pages {
        name := filepath.Base(page)

        // Parse the base template file into a template set.
        ts, err := template.ParseFiles("./ui/html/base.tmpl")
        if err != nil {
            return nil, err
        }

        // Call ParseGlob() *on this template set* to add any partials.
        ts, err = ts.ParseGlob("./ui/html/partials/*.tmpl")
        if err != nil {
            return nil, err
        }

        // Call ParseFiles() *on this template set* to add the page template.
        ts, err = ts.ParseFiles(page)
        if err != nil {
            return nil, err
        }

        // Add the template set to the map as normal...
        cache[name] = ts
    }

    return cache, nil
}
```

Catching runtime errors

As soon as we begin adding dynamic behavior to our HTML templates there's a risk of encountering runtime errors.

Let's add a deliberate error to the `view tmpl` template and see what happens:

```
File: ui/html/pages/view.tmpl

{{define "title"}}Snippet #{{.Snippet.ID}}{{end}}

{{define "main"}}
{{with .Snippet}}


<div class='metadata'>
    <strong>{{.Title}}</strong>
    <span>#{{.ID}}</span>
</div>
{{len nil}} <!-- Deliberate error -->
<pre><code>{{.Content}}</code></pre>
<div class='metadata'>
    <time>Created: {{.Created}}</time>
    <time>Expires: {{.Expires}}</time>
</div>
{{end}}
{{end}}


```

In this markup above we've added the line `{{len nil}}`, which should generate an error at runtime because in Go the value `nil` does not have a length.

Try running the application now. You'll find that everything still compiles OK:

```
$ go run ./cmd/web
time=2024-03-18T11:29:23.000+00:00 level=INFO msg="starting server" addr=:4000
```

But if you use curl to make a request to `http://localhost:4000/snippet/view/1` you'll get a response which looks a bit like this.

```
$ curl -i http://localhost:4000/snippet/view/1
HTTP/1.1 200 OK
Date: Wed, 18 Mar 2024 11:29:23 GMT
Content-Length: 734
Content-Type: text/html; charset=utf-8

<!doctype html>
<html lang='en'>
  <head>
    <meta charset='utf-8'>
    <title>Snippet #1 - Snippetbox</title>
    <link rel='stylesheet' href='/static/css/main.css'>
    <link rel='shortcut icon' href='/static/img/favicon.ico' type='image/x-icon'>
    <link rel='stylesheet' href='https://fonts.googleapis.com/css?family=Ubuntu+Mono:400,700'>
  </head>
  <body>
    <header>
      <h1><a href='/'>Snippetbox</a></h1>
    </header>

    <nav>
      <a href='/'>Home</a>
    </nav>

    <main>

      <div class='snippet'>
        <div class='metadata'>
          <strong>An old silent pond</strong>
          <span>#1</span>
        </div>
        Internal Server Error
      </div>
    </main>
  </body>
</html>
```

This is pretty bad. Our application has thrown an error, but the user has wrongly been sent a `200 OK` response. And even worse, they've received a half-complete HTML page.

To fix this we need to make the template render a two-stage process. First, we should make a ‘trial’ render by writing the template into a buffer. If this fails, we can respond to the user with an error message. But if it works, we can then write the contents of the buffer to our `http.ResponseWriter`.

Let’s update the `render()` helper to use this approach instead:

```
File: cmd/web/helpers.go
```

```
package main

import (
    "bytes" // New import
    "fmt"
    "net/http"
)

...

func (app *application) render(w http.ResponseWriter, r *http.Request, status int, page string, data templateData) {
    ts, ok := app.templateCache[page]
    if !ok {
        err := fmt.Errorf("the template %s does not exist", page)
        app.serverError(w, r, err)
        return
    }

    // Initialize a new buffer.
    buf := new(bytes.Buffer)

    // Write the template to the buffer, instead of straight to the
    // http.ResponseWriter. If there's an error, call our serverError() helper
    // and then return.
    err := ts.ExecuteTemplate(buf, "base", data)
    if err != nil {
        app.serverError(w, r, err)
        return
    }

    // If the template is written to the buffer without any errors, we are safe
    // to go ahead and write the HTTP status code to http.ResponseWriter.
    w.WriteHeader(status)

    // Write the contents of the buffer to the http.ResponseWriter. Note: this
    // is another time where we pass our http.ResponseWriter to a function that
    // takes an io.Writer.
    buf.WriteTo(w)
}
```

Restart the application and try making the same request again. You should now get a proper error message and **500 Internal Server Error** response.

```
$ curl -i http://localhost:4000/snippet/view/1
HTTP/1.1 500 Internal Server Error
Content-Type: text/plain; charset=utf-8
X-Content-Type-Options: nosniff
Date: Wed, 18 Mar 2024 11:29:23 GMT
Content-Length: 22

Internal Server Error
```

Great stuff. That's looking much better.

Before we move on to the next chapter, head back to the `view tmpl` file and remove the

deliberate error:

```
File: ui/html/pages/view.tpl

{{define "title"}}Snippet #{{.Snippet.ID}}{{end}}

{{define "main"}}
{{with .Snippet}}


<div class='metadata'>
    <strong>{{.Title}}</strong>
    <span>#{{.ID}}</span>
</div>
<pre><code>{{.Content}}</code></pre>
<div class='metadata'>
    <time>Created: {{.Created}}</time>
    <time>Expires: {{.Expires}}</time>
</div>
{{end}}
{{end}}


```

Common dynamic data

In some web applications there may be common dynamic data that you want to include on more than one — or even every — webpage. For example, you might want to include the name and profile picture of the current user, or a CSRF token in all pages with forms.

In our case let's begin with something simple, and say that we want to include the current year in the footer on every page.

To do this we'll begin by adding a new `CurrentYear` field to the `templateData` struct, like so:

```
File: cmd/web/templates.go

package main

...

// Add a CurrentYear field to the templateData struct.
type templateData struct {
    CurrentYear int
    Snippet      models.Snippet
    Snippets     []models.Snippet
}

...
```

The next step is to add a `newTemplateData()` helper method to our application, which will return a `templateData` struct initialized with the current year.

I'll demonstrate:

```
File: cmd/web/helpers.go
```

```
package main

import (
    "bytes"
    "fmt"
    "net/http"
    "time" // New import
)

...

// Create an newTemplateData() helper, which returns a pointer to a templateData
// struct initialized with the current year. Note that we're not using the
// *http.Request parameter here at the moment, but we will do later in the book.
func (app *application) newTemplateData(r *http.Request) templateData {
    return templateData{
        CurrentYear: time.Now().Year(),
    }
}

...
```

Then let's update our `home` and `snippetView` handlers to use the `newTemplateData()` helper, like so:

File: cmd/web/handlers.go

```
package main

...

func (app *application) home(w http.ResponseWriter, r *http.Request) {
    w.Header().Add("Server", "Go")

    snippets, err := app.snippets.Latest()
    if err != nil {
        app.serverError(w, r, err)
        return
    }

    // Call the newTemplateData() helper to get a templateData struct containing
    // the 'default' data (which for now is just the current year), and add the
    // snippets slice to it.
    data := app.newTemplateData(r)
    data.Snippets = snippets

    // Pass the data to the render() helper as normal.
    app.render(w, r, http.StatusOK, "home tmpl", data)
}

func (app *application) snippetView(w http.ResponseWriter, r *http.Request) {
    id, err := strconv.Atoi(r.PathValue("id"))
    if err != nil || id < 1 {
        http.NotFound(w, r)
        return
    }

    snippet, err := app.snippets.Get(id)
    if err != nil {
        if errors.Is(err, models.ErrNoRecord) {
            http.NotFound(w, r)
        } else {
            app.serverError(w, r, err)
        }
        return
    }

    // And do the same thing again here...
    data := app.newTemplateData(r)
    data.Snippet = snippet

    app.render(w, r, http.StatusOK, "view tmpl", data)
}

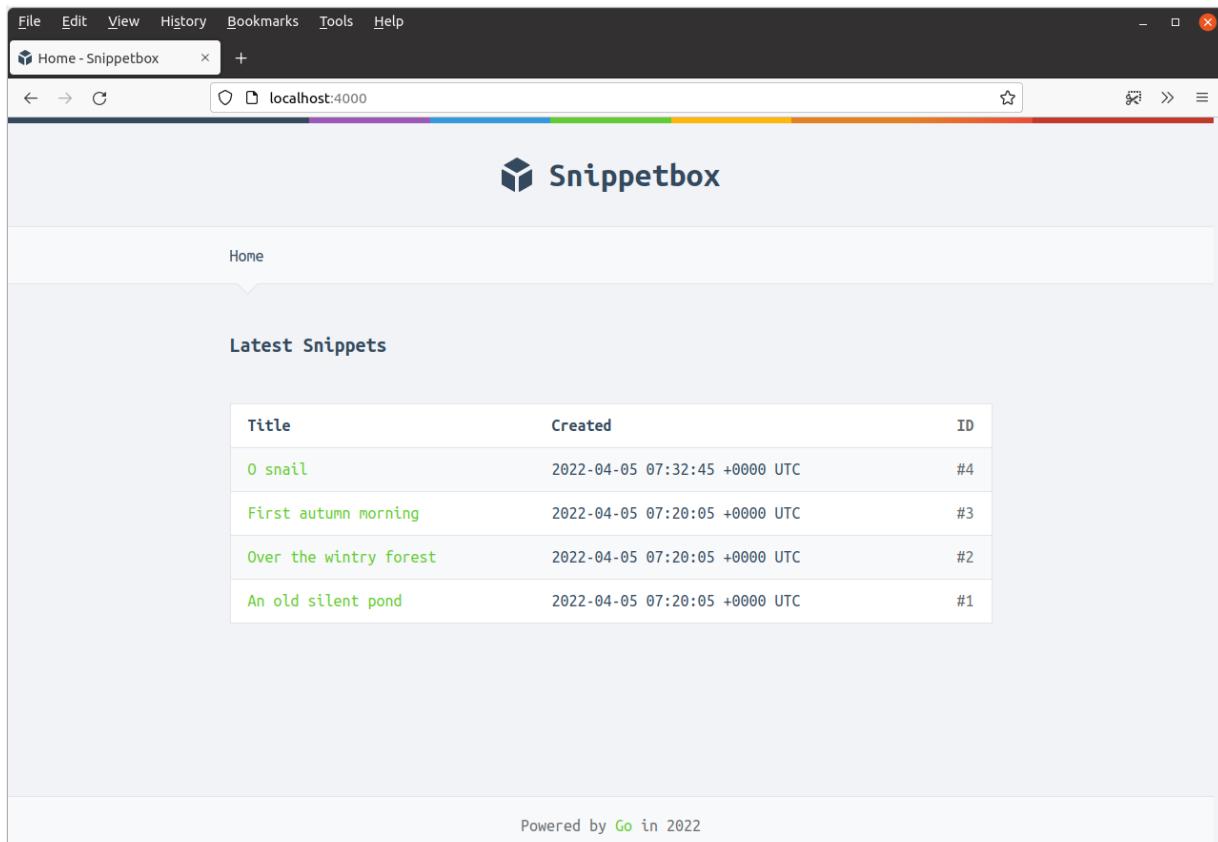
...
```

And then the final thing we need to do is update the `ui/html/base tmpl` file to display the year in the footer, like so:

```
File: ui/html/base tmpl
```

```
 {{define "base"}}
<!doctype html>
<html lang='en'>
  <head>
    <meta charset='utf-8'>
    <title>{{template "title" .}} - Snippetbox</title>
    <link rel='stylesheet' href='/static/css/main.css'>
    <link rel='shortcut icon' href='/static/img/favicon.ico' type='image/x-icon'>
    <link rel='stylesheet' href='https://fonts.googleapis.com/css?family=Ubuntu+Mono:400,700'>
  </head>
  <body>
    <header>
      <h1><a href='/'>Snippetbox</a></h1>
    </header>
    {{template "nav" .}}
    <main>
      {{template "main" .}}
    </main>
    <footer>
      <!-- Update the footer to include the current year -->
      Powered by <a href='https://golang.org/'>Go</a> in {{.CurrentYear}}
    </footer>
    <script src='/static/js/main.js' type='text/javascript'></script>
  </body>
</html>
{{end}}
```

If you restart the application and visit the home page at <http://localhost:4000>, you now should see the current year in the footer. Like this:



Custom template functions

In the last part of this section about templating and dynamic data, I'd like to explain how to create your own custom functions to use in Go templates.

To illustrate this, let's create a custom `humanDate()` function which outputs datetimes in a nice 'humanized' format like `1 Jan 2024 at 10:47` or `18 Mar 2024 at 15:04` instead of outputting dates in the default format of `YYYY-MM-DD HH:MM:SS +0000 UTC` like we are currently.

There are two main steps to doing this:

1. We need to create a `template.FuncMap` object containing the custom `humanDate()` function.
2. We need to use the `template.Funcs()` method to register this before parsing the templates.

Go ahead and add the following code to your `templates.go` file:

File: cmd/web/templates.go

```
package main

import (
    "html/template"
    "path/filepath"
    "time" // New import

    "snippetbox.alexedwards.net/internal/models"
)

...

// Create a humanDate function which returns a nicely formatted string
// representation of a time.Time object.
func humanDate(t time.Time) string {
    return t.Format("02 Jan 2006 at 15:04")
}

// Initialize a template.FuncMap object and store it in a global variable. This is
// essentially a string-keyed map which acts as a lookup between the names of our
// custom template functions and the functions themselves.
var functions = template.FuncMap{
    "humanDate": humanDate,
}

func newTemplateCache() (*map[string]*template.Template, error) {
    cache := map[string]*template.Template{}

    pages, err := filepath.Glob("./ui/html/pages/*.tmpl")
    if err != nil {
        return nil, err
    }

    for _, page := range pages {
        name := filepath.Base(page)

        // The template.FuncMap must be registered with the template set before you
        // call the ParseFiles() method. This means we have to use template.New() to
        // create an empty template set, use the Funcs() method to register the
        // template.FuncMap, and then parse the file as normal.
        ts, err := template.New(name).Funcs(functions).ParseFiles("./ui/html/base.tmpl")
        if err != nil {
            return nil, err
        }

        ts, err = ts.ParseGlob("./ui/html/partials/*.tmpl")
        if err != nil {
            return nil, err
        }

        ts, err = ts.ParseFiles(page)
        if err != nil {
            return nil, err
        }

        cache[name] = ts
    }

    return cache, nil
}
```

Before we continue, I should explain: custom template functions (like our `humanDate()` function) can accept as many parameters as they need to, but they *must* return one value only. The only exception to this is if you want to return an error as the second value, in which case that's OK too.

Now we can use our `humanDate()` function in the same way as the built-in template functions:

```
File: ui/html/pages/home.tpl

{{define "title"}}Home{{end}}

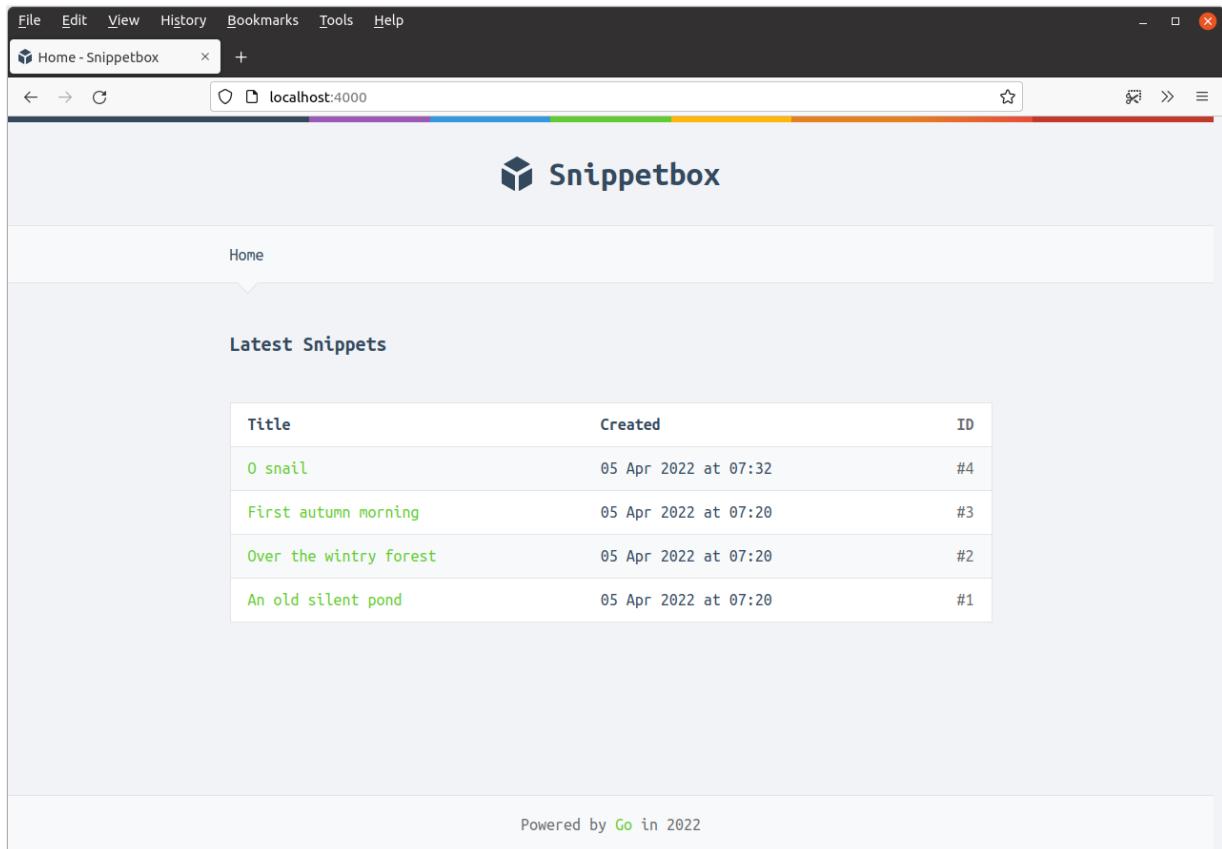
{{define "main"}}
<h2>Latest Snippets</h2>
{{if .Snippets}}
<table>
  <tr>
    <th>Title</th>
    <th>Created</th>
    <th>ID</th>
  </tr>
  {{range .Snippets}}
  <tr>
    <td><a href='/snippet/view/{{.ID}}'>{{.Title}}</a></td>
    <!-- Use the new template function here -->
    <td>{{humanDate .Created}}</td>
    <td>#{{.ID}}</td>
  </tr>
  {{end}}
</table>
{{else}}
  <p>There's nothing to see here... yet!</p>
{{end}}
{{end}}
```

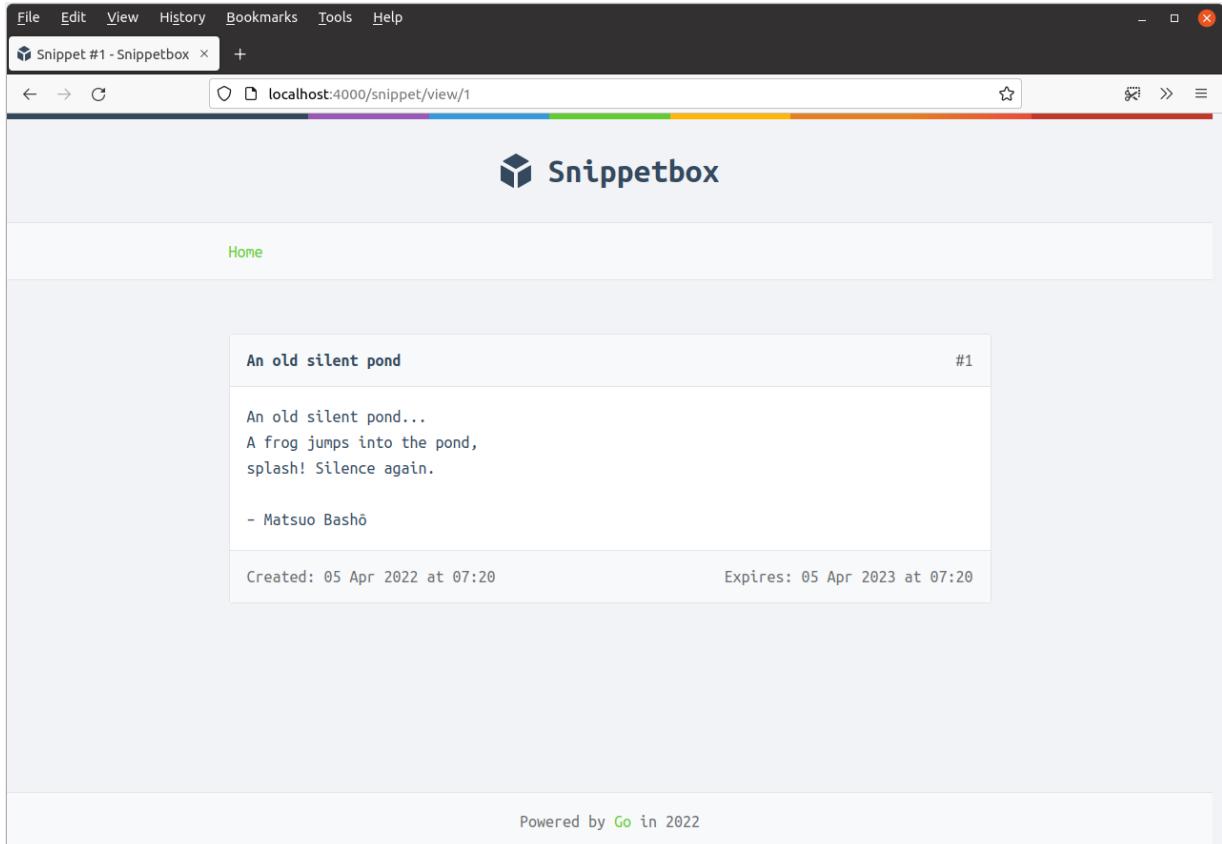
```
File: ui/html/pages/view.tpl

{{define "title"}}Snippet #{{.Snippet.ID}}{{end}}

{{define "main"}}
{{with .Snippet}}
<div class='snippet'>
  <div class='metadata'>
    <strong>{{.Title}}</strong>
    <span>#{{.ID}}</span>
  </div>
  <pre><code>{{.Content}}</code></pre>
  <div class='metadata'>
    <!-- Use the new template function here -->
    <time>Created: {{humanDate .Created}}</time>
    <time>Expires: {{humanDate .Expires}}</time>
  </div>
</div>
{{end}}
{{end}}
```

Once that's done, restart the application. If you visit <http://localhost:4000> and <http://localhost:4000/snippet/view/1> in your browser you should see the new, nicely formatted, dates being used.





Additional information

Pipelining

In the code above, we called our custom template function like this:

```
<time>Created: {{humanDate .Created}}</time>
```

An alternative approach is to use the `|` character to *pipeline* values to a function. This works a bit like piping outputs from one command to another in Unix terminals. We could rewrite the above as:

```
<time>Created: {{.Created | humanDate}}</time>
```

A nice feature of pipelining is that you can make an arbitrarily long chain of template functions which use the output from one as the input for the next. For example, we could pipeline the output from our `humanDate` function to the inbuilt `printf` function like so:

```
<time>{{.Created | humanDate | printf "Created: %s"}}</time>
```

Middleware

When you're building a web application there's probably some shared functionality that you want to use for many (or even all) HTTP requests. For example, you might want to log every request, compress every response, or check a cache before passing the request to your handlers.

A common way of organizing this shared functionality is to set it up as middleware. This is essentially some self-contained code which independently acts on a request before or after your normal application handlers.

In this section of the book you'll learn:

- An idiomatic pattern for [building and using custom middleware](#) which is compatible with `net/http` and many third-party packages.
- How to create middleware which [sets common HTTP headers](#) on every HTTP response.
- How to create middleware which [logs the requests](#) received by your application.
- How to create middleware which [recovers panics](#) so that they are gracefully handled by your application.
- How to create and use composable [middleware chains](#) to help manage and organize your middleware.

How middleware works

[Earlier in the book](#) I said something that I'd like to expand on in this chapter:

"You can think of a Go web application as a chain of `ServeHTTP()` methods being called one after another."

Currently, in our application, when our server receives a new HTTP request it calls the servemux's `ServeHTTP()` method. This looks up the relevant handler based on the request method and URL path, and in turn calls that handler's `ServeHTTP()` method.

The basic idea of middleware is to insert another handler into this chain. The middleware handler executes some logic, like logging a request, and then calls the `ServeHTTP()` method of the *next* handler in the chain.

In fact, we're actually already using some middleware in our application — the `http.StripPrefix()` function from [serving static files](#), which removes a specific prefix from the request's URL path before passing the request on to the file server.

The pattern

The standard pattern for creating your own middleware looks like this:

```
func myMiddleware(next http.Handler) http.Handler {
    fn := func(w http.ResponseWriter, r *http.Request) {
        // TODO: Execute our middleware logic here...
        next.ServeHTTP(w, r)
    }

    return http.HandlerFunc(fn)
}
```

The code itself is pretty succinct, but there's quite a lot in it to get your head around.

- The `myMiddleware()` function is essentially a wrapper around the `next` handler, which we pass to it as a parameter.
- It establishes a function `fn` which *closes over* the `next` handler to form a closure. When `fn` is run it executes our middleware logic and then transfers control to the `next` handler by calling its `ServeHTTP()` method.

- Regardless of what you do with a closure it will always be able to access the variables that are local to the scope it was created in — which in this case means that `fn` will always have access to the `next` variable.
- In the final line of code, we then convert this closure to a `http.Handler` and return it using the `http.HandlerFunc()` adapter.

If this feels confusing, you can think of it more simply: `myMiddleware()` is a function that accepts the next handler in a chain as a parameter. It *returns a handler* which executes some logic and then calls the next handler.

Simplifying the middleware

A tweak to this pattern is to use an anonymous function inside `myMiddleware()` middleware, like so:

```
func myMiddleware(next http.Handler) http.Handler {
    return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
        // TODO: Execute our middleware logic here...
        next.ServeHTTP(w, r)
    })
}
```

This pattern is very common in the wild, and the one that you'll probably see most often if you're reading the source code of other applications or third-party packages.

Positioning the middleware

It's important to explain that where you position the middleware in the chain of handlers will affect the behavior of your application.

If you position your middleware before the servemux in the chain then it will act on every request that your application receives.

```
myMiddleware → servemux → application handler
```

A good example of where this would be useful is middleware to log requests — as that's typically something you would want to do for *all* requests.

Alternatively, you can position the middleware after the servemux in the chain — by

wrapping a specific application handler. This would cause your middleware to only be executed for a specific route.

```
servemux → myMiddleware → application handler
```

An example of this would be something like authorization middleware, which you may only want to run on specific routes.

We'll demonstrate how to do both of these things in practice as we progress through the book.

Setting common headers

Let's put the pattern we learned in the previous chapter to use, and make some middleware which automatically adds our `Server: Go` header to every response, along with the following HTTP security headers (inline with [current OWASP guidance](#)).

```
Content-Security-Policy: default-src 'self'; style-src 'self' fonts.googleapis.com; font-src fonts.gstatic.com  
Referrer-Policy: origin-when-cross-origin  
X-Content-Type-Options: nosniff  
X-Frame-Options: deny  
X-XSS-Protection: 0
```

If you're not familiar with these headers, I'll quickly explain what they do.

- **Content-Security-Policy** (often abbreviated to *CSP*) headers are used to restrict where the resources for your web page (e.g. JavaScript, images, fonts etc) can be loaded from. Setting a strict CSP policy helps prevent a variety of cross-site scripting, clickjacking, and other code-injection attacks.

CSP headers and how they work is a big topic, and I recommend reading [this primer](#) if you haven't come across them before. But, in our case, the header tells the browser that it's OK to load fonts from `fonts.gstatic.com`, stylesheets from `fonts.googleapis.com` and `self` (our own origin), and then *everything else* only from `self`. Inline JavaScript is blocked by default.
- **Referrer-Policy** is used to control what information is included in a `Referer` header when a user navigates away from your web page. In our case, we'll set the value to `origin-when-cross-origin`, which means that the full URL will be included for [same-origin requests](#), but for all other requests information like the URL path and any query string values will be stripped out.
- **X-Content-Type-Options: nosniff** instructs browsers to *not* MIME-type sniff the content-type of the response, which in turn helps to prevent [content-sniffing attacks](#).
- **X-Frame-Options: deny** is used to help prevent [clickjacking](#) attacks in older browsers that don't support CSP headers.
- **X-XSS-Protection: 0** is used to *disable* the blocking of cross-site scripting attacks. Previously it was good practice to set this header to `X-XSS-Protection: 1; mode=block`, but when you're using CSP headers like we are [the recommendation](#) is to disable this

feature altogether.

OK, let's get back to our Go code and begin by creating a new `middleware.go` file. We'll use this to hold all the custom middleware that we write throughout this book.

```
$ touch cmd/web/middleware.go
```

Then open it up and add a `commonHeaders()` function using the pattern that we introduced in the previous chapter:

```
File: cmd/web/middleware.go

package main

import (
    "net/http"
)

func commonHeaders(next http.Handler) http.Handler {
    return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
        // Note: This is split across multiple lines for readability. You don't
        // need to do this in your own code.
        w.Header().Set("Content-Security-Policy",
            "default-src 'self'; style-src 'self' fonts.googleapis.com; font-src fonts.gstatic.com")

        w.Header().Set("Referrer-Policy", "origin-when-cross-origin")
        w.Header().Set("X-Content-Type-Options", "nosniff")
        w.Header().Set("X-Frame-Options", "deny")
        w.Header().Set("X-XSS-Protection", "0")

        w.Header().Set("Server", "Go")

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

Because we want this middleware to act on every request that is received, we need it to be executed *before* a request hits our servemux. We want the flow of control through our application to look like:

```
commonHeaders → servemux → application handler
```

To do this we'll need the `commonHeaders` middleware function to *wrap our servemux*. Let's update the `routes.go` file to do exactly that:

```

File: cmd/web/routes.go

package main

import "net/http"

// Update the signature for the routes() method so that it returns a
// http.Handler instead of *http.ServeMux.
func (app *application) routes() http.Handler {
    mux := http.NewServeMux()

    fileServer := http.FileServer(http.Dir("./ui/static/"))
    mux.Handle("GET /static/", http.StripPrefix("/static", fileServer))

    mux.HandleFunc("GET /{$}", app.home)
    mux.HandleFunc("GET /snippet/view/{id}", app.snippetView)
    mux.HandleFunc("GET /snippet/create", app.snippetCreate)
    mux.HandleFunc("POST /snippet/create", app.snippetCreatePost)

    // Pass the servemux as the 'next' parameter to the commonHeaders middleware.
    // Because commonHeaders is just a function, and the function returns a
    // http.Handler we don't need to do anything else.
    return commonHeaders(mux)
}

```

Important: Make sure that you update the signature of the `routes()` method so that it returns a `http.Handler` here, otherwise you'll get a compile-time error.

We also need to quickly update our `home` handler code to remove the `w.Header().Add("Server", "Go")` line, otherwise we'll end up adding that header twice on responses from the homepage.

```

File: cmd/web/handlers.go

package main

...

func (app *application) home(w http.ResponseWriter, r *http.Request) {
    snippets, err := app.snippets.Latest()
    if err != nil {
        app.serverError(w, r, err)
        return
    }

    data := app.newTemplateData(r)
    data.Snippets = snippets

    app.render(w, r, http.StatusOK, "home.tmpl", data)
}

...

```

Go ahead and give this a try. Run the application then open a terminal window and try

making some requests with curl. You should see that the security headers are now included in every response.

```
$ curl --head http://localhost:4000/
HTTP/1.1 200 OK
Content-Security-Policy: default-src 'self'; style-src 'self' fonts.googleapis.com; font-src fonts.gstatic.com
Referrer-Policy: origin-when-cross-origin
Server: Go
X-Content-Type-Options: nosniff
X-Frame-Options: deny
X-Xss-Protection: 0
Date: Wed, 18 Mar 2024 11:29:23 GMT
Content-Length: 1700
Content-Type: text/html; charset=utf-8
```

Additional information

Flow of control

It's important to know that when the last handler in the chain returns, control is passed back up the chain in the reverse direction. So when our code is being executed the flow of control actually looks like this:

```
commonHeaders → servemux → application handler → servemux → commonHeaders
```

In any middleware handler, code which comes before `next.ServeHTTP()` will be executed on the way down the chain, and any code after `next.ServeHTTP()` — or in a deferred function — will be executed on the way back up.

```
func myMiddleware(next http.Handler) http.Handler {
    return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
        // Any code here will execute on the way down the chain.
        next.ServeHTTP(w, r)
        // Any code here will execute on the way back up the chain.
    })
}
```

Early returns

Another thing to mention is that if you call `return` in your middleware function *before* you call `next.ServeHTTP()`, then the chain will stop being executed and control will flow back upstream.

As an example, a common use-case for early returns is authentication middleware which only allows execution of the chain to continue if a particular check is passed. For instance:

```
func myMiddleware(next http.Handler) http.Handler {
    return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
        // If the user isn't authorized, send a 403 Forbidden status and
        // return to stop executing the chain.
        if !isAuthorized(r) {
            w.WriteHeader(http.StatusForbidden)
            return
        }

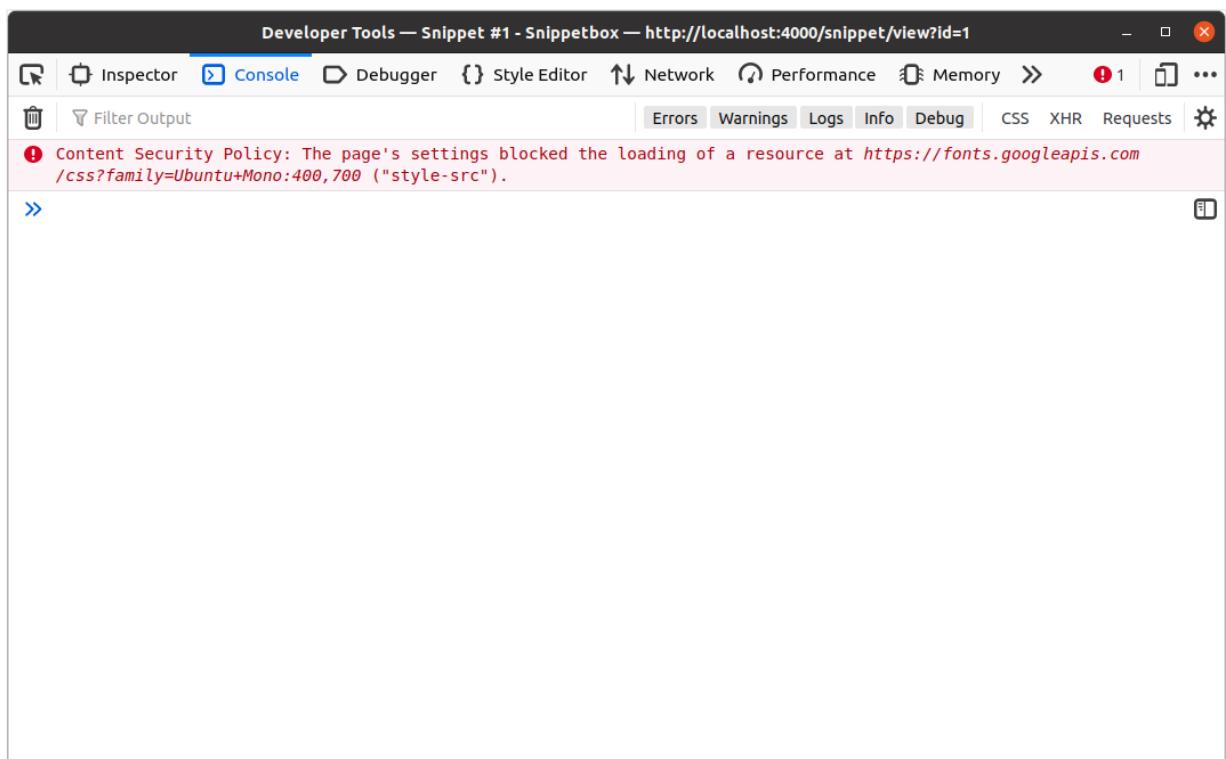
        // Otherwise, call the next handler in the chain.
        next.ServeHTTP(w, r)
    })
}
```

We'll use this 'early return' pattern [later in the book](#) to restrict access to certain parts of our application.

Debugging CSP issues

While CSP headers are great and you should definitely use them, it's worth saying that I've spent many hours trying to debug problems, only to eventually realize that a critical resource or script is being blocked by my own CSP rules .

If you're working on a project which is using CSP headers, like this one, I recommend keeping your web browser developer tools handy and getting into the habit of checking the logs early on if you run into any unexpected problems. In Firefox, any blocked resources will be shown as an error in the console logs — similar to this:



Request logging

Let's continue in the same vein and add some middleware to *log HTTP requests*.

Specifically, we're going to use the structured logger that we created earlier to record the IP address of the user, and the method, URI and HTTP version for the request.

Open your `middleware.go` file and create a `logRequest()` method using the standard middleware pattern, like so:

```
File: cmd/web/middleware.go

package main

...

func (app *application) logRequest(next http.Handler) http.Handler {
    return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
        var (
            ip      = r.RemoteAddr
            proto   = r.Proto
            method  = r.Method
            uri    = r.URL.RequestURI()
        )

        app.logger.Info("received request", "ip", ip, "proto", proto, "method", method, "uri", uri)

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

Notice that this time we're implementing the middleware as a method on `application`?

This is perfectly valid to do. Our middleware method has the same signature as before, but because it is a method against `application` it *also* has access to the handler dependencies including the structured logger.

Now let's update our `routes.go` file so that the `logRequest` middleware is executed first, and for all requests, so that the flow of control (reading from left to right) looks like this:

```
logRequest ⇄ commonHeaders ⇄ servemux ⇄ application handler
```

```
File: cmd/web/routes.go
```

```
package main

import "net/http"

func (app *application) routes() http.Handler {
    mux := http.NewServeMux()

    fileServer := http.FileServer(http.Dir("./ui/static/"))
    mux.Handle("GET /static/", http.StripPrefix("/static", fileServer))

    mux.HandleFunc("GET /{$}", app.home)
    mux.HandleFunc("GET /snippet/view/{id}", app.snippetView)
    mux.HandleFunc("GET /snippet/create", app.snippetCreate)
    mux.HandleFunc("POST /snippet/create", app.snippetCreatePost)

    // Wrap the existing chain with the logRequest middleware.
    return app.logRequest(commonHeaders(mux))
}
```

Alright... let's give it a try!

Restart your application, browse around, and then check your terminal window. You should see log output which looks a bit like this:

```
$ go run ./cmd/web
time=2024-03-18T11:29:23.000+00:00 level=INFO msg="starting server" addr=:4000
time=2024-03-18T11:29:23.000+00:00 level=INFO msg="received request" ip=127.0.0.1:56536 proto=HTTP/1.1 method=GET uri=/
time=2024-03-18T11:29:23.000+00:00 level=INFO msg="received request" ip=127.0.0.1:56536 proto=HTTP/1.1 method=GET uri=/static
time=2024-03-18T11:29:23.000+00:00 level=INFO msg="received request" ip=127.0.0.1:56546 proto=HTTP/1.1 method=GET uri=/static
time=2024-03-18T11:29:23.000+00:00 level=INFO msg="received request" ip=127.0.0.1:56536 proto=HTTP/1.1 method=GET uri=/static
time=2024-03-18T11:29:23.000+00:00 level=INFO msg="received request" ip=127.0.0.1:56536 proto=HTTP/1.1 method=GET uri=/static
time=2024-03-18T11:29:23.000+00:00 level=INFO msg="received request" ip=127.0.0.1:56536 proto=HTTP/1.1 method=GET uri="/snipp
```

Note: Depending on how your browser caches static files, you might need to do a hard refresh (or open a new incognito/private browsing tab) to see any requests for static files.

Panic recovery

In a simple Go application, [when your code panics](#) it will result in the application being terminated straight away.

But our web application is a bit more sophisticated. Go's HTTP server assumes that the effect of any panic is isolated to the goroutine serving the active HTTP request ([remember](#), every request is handled in its own goroutine).

Specifically, following a panic our server will log a stack trace to the server error log (which we will talk about [later in the book](#)), unwind the stack for the affected goroutine (calling any deferred functions along the way) and close the underlying HTTP connection. But it won't terminate the application, so importantly, any panic in your handlers *won't* bring down your server.

But if a panic does happen in one of our handlers, what will the user see?

Let's take a look and introduce a deliberate panic into our [home](#) handler.

```
File: cmd/web/handlers.go

package main

...

func (app *application) home(w http.ResponseWriter, r *http.Request) {
    panic("oops! something went wrong") // Deliberate panic

    snippets, err := app.snippets.Latest()
    if err != nil {
        app.serverError(w, r, err)
        return
    }

    data := app.newTemplateData(r)
    data.Snippets = snippets

    app.render(w, r, http.StatusOK, "home tmpl", data)
}

...
```

Restart your application...

```
$ go run ./cmd/web
time=2024-03-18T11:29:23.000+00:00 level=INFO msg="starting server" addr=:4000
```

... and make a HTTP request for the home page from a second terminal window:

```
$ curl -i http://localhost:4000
curl: (52) Empty reply from server
```

Unfortunately, all we get is an empty response due to Go closing the underlying HTTP connection following the panic.

This isn't a great experience for the user. It would be more appropriate and meaningful to send them a proper HTTP response with a **500 Internal Server Error** status instead.

A neat way of doing this is to create some middleware which *recovers* the panic and calls our `app.serverError()` helper method. To do this, we can leverage the fact that deferred functions are always called when the stack is being unwound following a panic.

Open up your `middleware.go` file and add the following code:

```
File: cmd/web/middleware.go

package main

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

...

func (app *application) recoverPanic(next http.Handler) http.Handler {
    return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
        // Create a deferred function (which will always be run in the event
        // of a panic as Go unwinds the stack).
        defer func() {
            // Use the builtin recover function to check if there has been a
            // panic or not. If there has...
            if err := recover(); err != nil {
                // Set a "Connection: close" header on the response.
                w.Header().Set("Connection", "close")
                // Call the app.serverError helper method to return a 500
                // Internal Server response.
                app.serverError(w, r, fmt.Errorf("%s", err))
            }
        }()
        next.ServeHTTP(w, r)
    })
}
```

There are two details about this which are worth explaining:

- Setting the `Connection: Close` header on the response acts as a trigger to make Go's HTTP server automatically close the current connection after a response has been sent.

It also informs the user that the connection *will be closed*. Note: If the protocol being used is HTTP/2, Go will automatically strip the `Connection: Close` header from the response (so it is not malformed) and send a `GOAWAY` frame.

- The value returned by the builtin `recover()` function has the type `any`, and its underlying type could be `string`, `error`, or something else — whatever the parameter passed to `panic()` was. In our case, it's the string `"oops! something went wrong"`. In the code above, we normalize this into an `error` by using the `fmt.Errorf()` function to create a new `error` object containing the default textual representation of the `any` value, and then pass this `error` to the `app.serverError()` helper method.

Let's now put this to use in the `routes.go` file, so that it is the *first* thing in our chain to be executed (so that it covers panics in all subsequent middleware and handlers).

```
File: cmd/web/routes.go

package main

import "net/http"

func (app *application) routes() http.Handler {
    mux := http.NewServeMux()

    fileServer := http.FileServer(http.Dir("./ui/static/"))
    mux.Handle("GET /static/", http.StripPrefix("/static", fileServer))

    mux.HandleFunc("GET /${}", app.home)
    mux.HandleFunc("GET /snippet/view/{id}", app.snippetView)
    mux.HandleFunc("GET /snippet/create", app.snippetCreate)
    mux.HandleFunc("POST /snippet/create", app.snippetCreatePost)

    // Wrap the existing chain with the recoverPanic middleware.
    return app.recoverPanic(app.logRequest(commonHeaders(mux)))
}
```

If you restart the application and make a request for the homepage now, you should see a nicely formed `500 Internal Server Error` response following the panic, including the `Connection: close` header that we talked about.

```
$ go run ./cmd/web
time=2024-03-18T11:29:23.000+00:00 level=INFO msg="starting server" addr=:4000
```

```
$ curl -i http://localhost:4000
HTTP/1.1 500 Internal Server Error
Connection: close
Content-Security-Policy: default-src 'self'; style-src 'self' fonts.googleapis.com; font-src fonts.gstatic.com
Content-Type: text/plain; charset=utf-8
Referrer-Policy: origin-when-cross-origin
Server: Go
X-Content-Type-Options: nosniff
X-Frame-Options: deny
X-Xss-Protection: 0
Date: Wed, 18 Mar 2024 11:29:23 GMT
Content-Length: 22

Internal Server Error
```

Before we continue, head back to your `home` handler and remove the deliberate panic from the code.

```
File: cmd/web/handlers.go

package main

...
func (app *application) home(w http.ResponseWriter, r *http.Request) {
    snippets, err := app.snippetsLatest()
    if err != nil {
        app.serverError(w, r, err)
        return
    }

    data := app.newTemplateData(r)
    data.Snippets = snippets

    app.render(w, r, http.StatusOK, "home tmpl", data)
}
...
```

Additional information

Panic recovery in background goroutines

It's important to realize that our middleware will only recover panics that happen in the *same goroutine that executed the `recoverPanic()` middleware*.

If, for example, you have a handler which spins up another goroutine (e.g. to do some background processing), then any panics that happen in the second goroutine will not be recovered — not by the `recoverPanic()` middleware... and not by the panic recovery built

into Go HTTP server. They will cause your application to exit and bring down the server.

So, if you are spinning up additional goroutines from within your web application and there is any chance of a panic, you must make sure that you recover any panics from within those too. For example:

```
func (app *application) myHandler(w http.ResponseWriter, r *http.Request) {
    ...

    // Spin up a new goroutine to do some background processing.
    go func() {
        defer func() {
            if err := recover(); err != nil {
                app.logger.Error(fmt.Sprint(err))
            }
        }()
        doSomeBackgroundProcessing()
    }()
    w.Write([]byte("OK"))
}
```

Composable middleware chains

In this chapter I'd like to introduce the [justinas/alice](#) package to help us manage our middleware/handler chains.

You don't *need* to use this package, but the reason I recommend it is because it makes it easy to create composable, reusable, middleware chains — and that can be a real help as your application grows and your routes become more complex. The package itself is also small and lightweight, and the code is clear and well written.

To demonstrate its features in one example, it allows you to rewrite a handler chain from this:

```
return myMiddleware1(myMiddleware2(myMiddleware3(myHandler)))
```

Into this, which is a bit clearer to understand at a glance:

```
return alice.New(myMiddleware1, myMiddleware2, myMiddleware3).Then(myHandler)
```

But the real power lies in the fact that you can use it to create middleware chains that can be assigned to variables, appended to, and reused. For example:

```
myChain := alice.New(myMiddlewareOne, myMiddlewareTwo)
myOtherChain := myChain.Append(myMiddleware3)
return myOtherChain.Then(myHandler)
```

If you're following along, please install the [justinas/alice](#) package using `go get`:

```
$ go get github.com/justinas/alice@v1
go: downloading github.com/justinas/alice v1.2.0
```

And if you open the `go.mod` file for your project, you should see a new corresponding `require` statement, like so:

```
File: go.mod

module snippetbox.alexewards.net

go 1.22.0

require github.com/go-sql-driver/mysql v1.8.0

require (
    filippo.io/edwards25519 v1.1.0 // indirect
    github.com/justinas/alice v1.2.0 // indirect
)
```

Again, this is currently listed as an indirect dependency because we're not actually importing and using it in our code yet.

Let's go ahead and do that now, updating our `routes.go` file to use the `justinas/alice` package as follows:

```
File: cmd/web/routes.go

package main

import (
    "net/http"

    "github.com/justinas/alice" // New import
)

func (app *application) routes() http.Handler {
    mux := http.NewServeMux()

    fileServer := http.FileServer(http.Dir("./ui/static/"))
    mux.Handle("GET /static/", http.StripPrefix("/static", fileServer))

    mux.HandleFunc("GET /${}", app.home)
    mux.HandleFunc("GET /snippet/view/{id}", app.snippetView)
    mux.HandleFunc("GET /snippet/create", app.snippetCreate)
    mux.HandleFunc("POST /snippet/create", app.snippetCreatePost)

    // Create a middleware chain containing our 'standard' middleware
    // which will be used for every request our application receives.
    standard := alice.New(app.recoverPanic, app.logRequest, commonHeaders)

    // Return the 'standard' middleware chain followed by the servemux.
    return standard.Then(mux)
}
```

If you want, feel free to restart the application at this point. You should find that everything compiles correctly and the application continues to work in the same way as before. You can also run `go mod tidy` again to remove the `// indirect` annotation from the `go.mod` file.

Processing forms

In this section of the book we're going to focus on adding an HTML form for creating new snippets. The form will look a bit like this:

The form is a wireframe representation of a snippet creation interface. It features a header 'SNIPPETBOX', a navigation bar with 'HOME' and 'NEW SNIPPET' links, and a main section titled 'ADD A NEW SNIPPET'. It contains two input fields: 'TITLE:' and 'CONTENT:', each with a large rectangular input area. Below the 'CONTENT:' field is a 'DELETE IN:' section with three radio buttons labeled 'ONE YEAR', 'ONE DAY', and 'ONE HOUR'. At the bottom is a 'PUBLISH SNIPPET' button.

The high-level flow for processing this form will follow a standard **Post-Redirect-Get** pattern and will work like so:

1. The user is shown the blank form when they make a **GET** request to `/snippet/create`.
2. The user completes the form and it's submitted to the server via a **POST** request to `/snippet/create`.
3. The form data will be validated by our `snippetCreatePost` handler. If there are any validation failures the form will be re-displayed with the appropriate form fields highlighted. If it passes our validation checks, the data for the new snippet will be added to the database and then we'll redirect the user to **GET** `/snippet/view/{id}`.

As part of this you'll learn:

- How to **parse and access** form data sent in a **POST** request.
- Some techniques for performing common **validation checks** on the form data.

- A [user-friendly pattern](#) for alerting the user to validation failures and re-populating form fields with previously submitted data.
- How to [keep your handlers clean](#) by using helpers for form processing and validation.

Setting up an HTML form

Let's begin by making a new `ui/html/pages/create.tpl` file to hold the HTML for the form:

```
$ touch ui/html/pages/create.tpl
```

... and then add the following markup, using the same template patterns that we used earlier in the book.

```
File: ui/html/pages/create.tpl

{{define "title"}}Create a New Snippet{{end}}

{{define "main"}}
<form action='/snippet/create' method='POST'>
  <div>
    <label>Title:</label>
    <input type='text' name='title'>
  </div>
  <div>
    <label>Content:</label>
    <textarea name='content'></textarea>
  </div>
  <div>
    <label>Delete in:</label>
    <input type='radio' name='expires' value='365' checked> One Year
    <input type='radio' name='expires' value='7'> One Week
    <input type='radio' name='expires' value='1'> One Day
  </div>
  <div>
    <input type='submit' value='Publish snippet'>
  </div>
</form>
{{end}}
```

There's nothing particularly special about this so far. Our `main` template contains a standard HTML form which sends three form values: `title`, `content` and `expires` (the number of days until the snippet should expire). The only thing to really point out is the form's `action` and `method` attributes — we've set these up so that the form will `POST` the data to the URL `/snippet/create` when it's submitted.

Now let's add a new 'Create snippet' link to the navigation bar for our application, so that clicking it will take the user to this new form.

```
File: ui/html/partials/nav.tpl
```

```
{{define "nav"}}
<nav>
  <a href='/'>Home</a>
  <!-- Add a link to the new form -->
  <a href='/snippet/create'>Create snippet</a>
</nav>
{{end}}
```

And finally, we need to update the `snippetcreateForm` handler so that it renders our new page like so:

```
File: cmd/web/handlers.go
```

```
package main

...

func (app *application) snippetCreate(w http.ResponseWriter, r *http.Request) {
    data := app.newTemplateData(r)

    app.render(w, r, http.StatusOK, "create.tpl", data)
}

...
```

At this point you can fire up the application and visit <http://localhost:4000/snippet/create> in your browser. You should see a form which looks like this:

File Edit View History Bookmarks Tools Help

Create a New Snippet - Snippetbox

localhost:4000/snippet/create

Snippetbox

Home Create snippet

Title:

Content:

Delete in: One Year One Week One Day

Publish snippet

Parsing form data

Thanks to the work we did previously in the [foundations](#) section, any `POST /snippets/create` requests are already being dispatched to our `snippetCreatePost` handler. We'll now update this handler to process and use the form data when it's submitted.

At a high-level, we can break this down into two distinct steps.

1. First, we need to use the `r.ParseForm()` method to parse the request body. This checks that the request body is well-formed, and then stores the form data in the request's `r.PostForm` map. If there are any errors encountered when parsing the body (like there is no body, or it's too large to process) then it will return an error. The `r.ParseForm()` method is also idempotent; it can safely be called multiple times on the same request without any side-effects.
2. We can then get to the form data contained in `r.PostForm` by using the `r.PostForm.Get()` method. For example, we can retrieve the value of the `title` field with `r.PostForm.Get("title")`. If there is no matching field name in the form this will return the empty string `""`.

Open your `cmd/web/handlers.go` file and update it to include the following code:

File: cmd/web/handlers.go

```
package main

...

func (app *application) snippetCreatePost(w http.ResponseWriter, r *http.Request) {
    // First we call r.ParseForm() which adds any data in POST request bodies
    // to the r.PostForm map. This also works in the same way for PUT and PATCH
    // requests. If there are any errors, we use our app.ClientError() helper to
    // send a 400 Bad Request response to the user.
    err := r.ParseForm()
    if err != nil {
        app.clientError(w, http.StatusBadRequest)
        return
    }

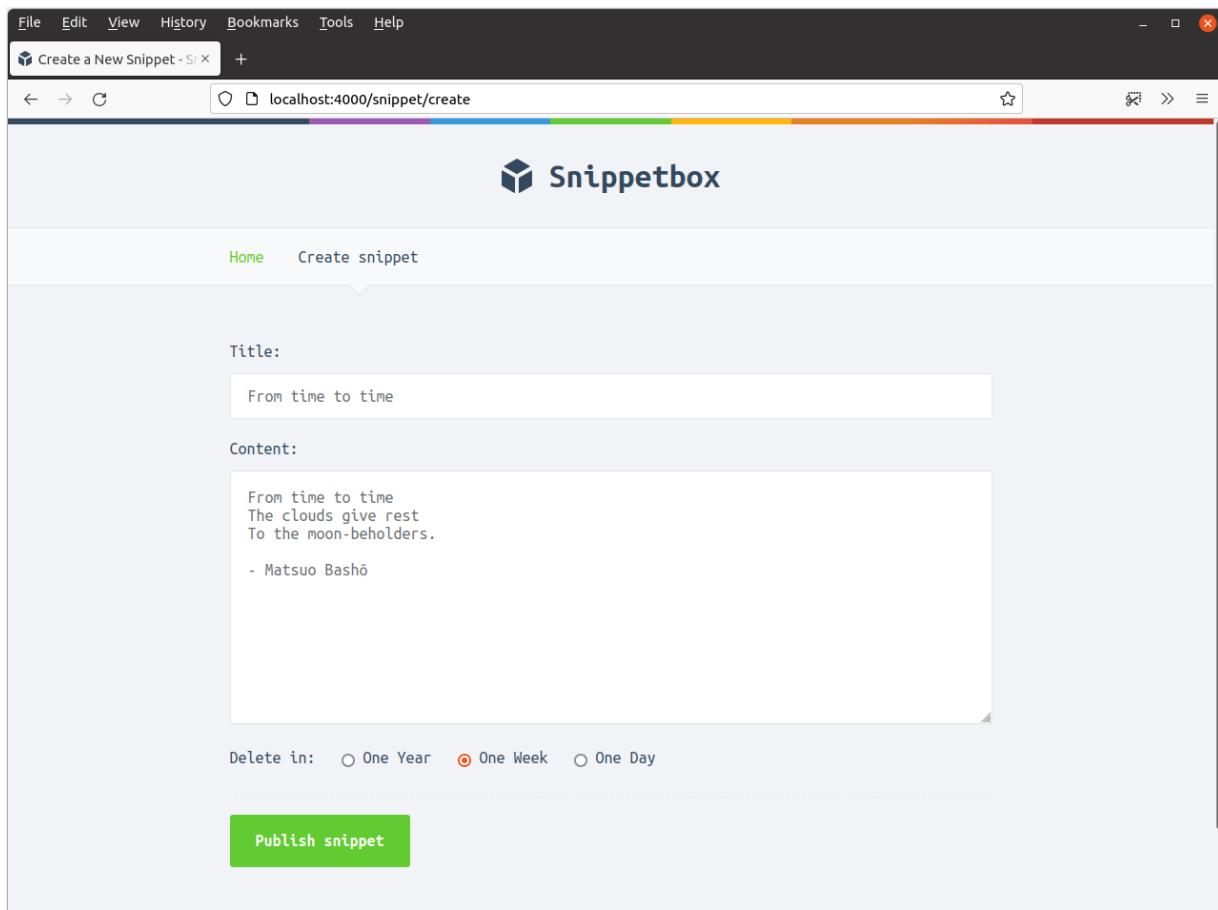
    // Use the r.PostForm.Get() method to retrieve the title and content
    // from the r.PostForm map.
    title := r.PostForm.Get("title")
    content := r.PostForm.Get("content")

    // The r.PostForm.Get() method always returns the form data as a *string*.
    // However, we're expecting our expires value to be a number, and want to
    // represent it in our Go code as an integer. So we need to manually convert
    // the form data to an integer using strconv.Atoi(), and we send a 400 Bad
    // Request response if the conversion fails.
    expires, err := strconv.Atoi(r.PostForm.Get("expires"))
    if err != nil {
        app.clientError(w, http.StatusBadRequest)
        return
    }

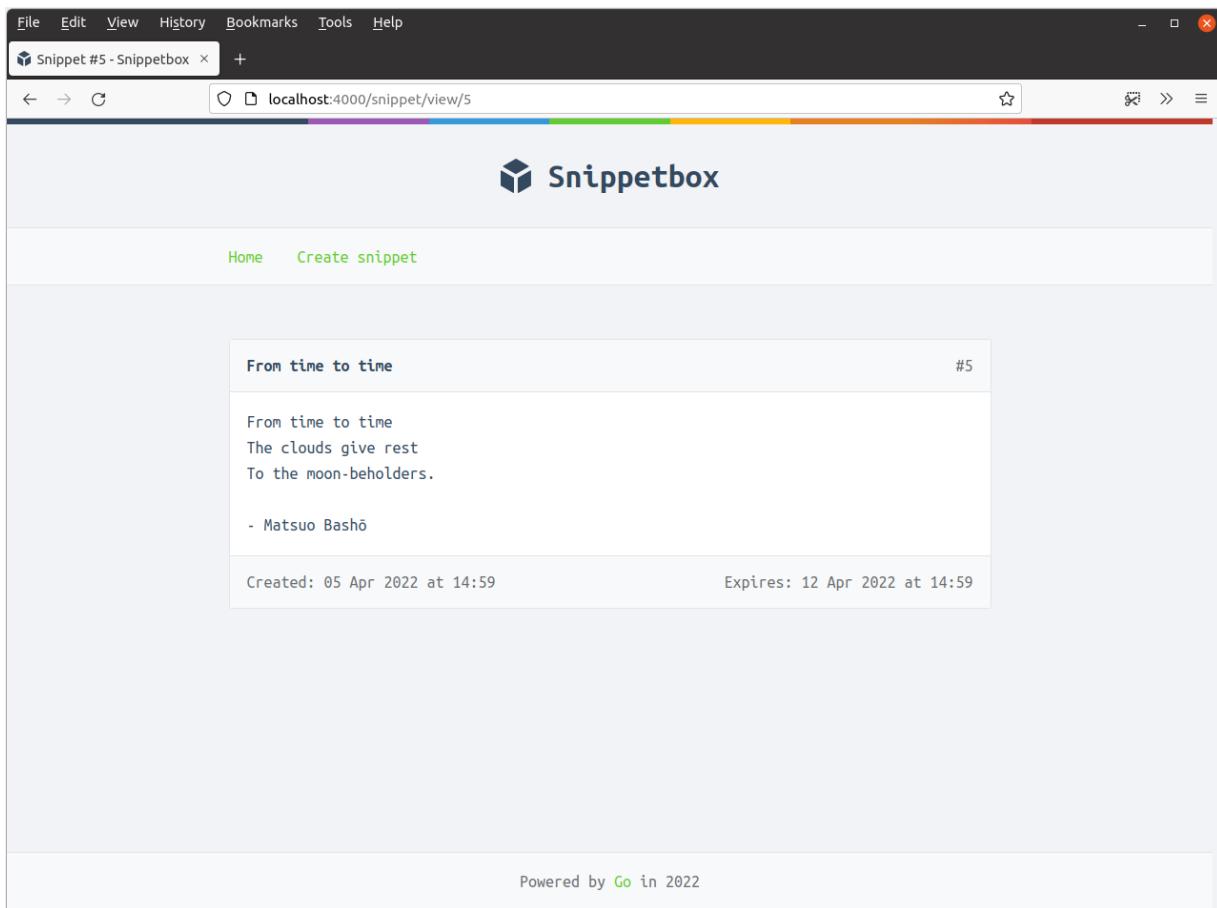
    id, err := app.snippets.Insert(title, content, expires)
    if err != nil {
        app.serverError(w, r, err)
        return
    }

    http.Redirect(w, r, fmt.Sprintf("/snippet/view/%d", id), http.StatusSeeOther)
}
```

Alright, let's give this a try! Restart the application and try filling in the form with the title and content of a snippet, a bit like this:



And then submit the form. If everything has worked, you should be redirected to a page displaying your new snippet like so:



Additional information

The PostFormValue method

The `net/http` package also provides the `r.PostFormValue()` method, which is essentially a shortcut function that calls `r.ParseForm()` for you and then fetches the appropriate field value from `r.PostForm`.

I recommend avoiding this shortcut because it *silently ignores any errors* returned by `r.ParseForm()`. If you use it, it means that the parsing step could be encountering errors and failing for users, but there's no feedback mechanism to let them (or you) know about the problem.

Multiple-value fields

Strictly speaking, the `r.PostForm.Get()` method that we've used in this chapter only returns the *first* value for a specific form field. This means you can't use it with form fields which potentially send multiple values, such as a group of checkboxes.

```
<input type="checkbox" name="items" value="foo"> Foo  
<input type="checkbox" name="items" value="bar"> Bar  
<input type="checkbox" name="items" value="baz"> Baz
```

In this case you'll need to work with the `r.PostForm` map directly. The underlying type of the `r.PostForm` map is `url.Values`, which in turn has the underlying type `map[string][]string`. So, for fields with multiple values you can loop over the underlying map to access them like so:

```
for i, item := range r.PostForm["items"] {  
    fmt.Fprintf(w, "%d: %s\n", i, item)  
}
```

Limiting form size

By default, forms submitted with a `POST` method have a size limit of 10MB of data. The exception to this is if your form has the `enctype="multipart/form-data"` attribute and is sending multipart data, in which case there is no default limit.

If you want to change the 10MB limit, you can use the `http.MaxBytesReader()` function like so:

```
// Limit the request body size to 4096 bytes  
r.Body = http.MaxBytesReader(w, r.Body, 4096)  
  
err := r.ParseForm()  
if err != nil {  
    http.Error(w, "Bad Request", http.StatusBadRequest)  
    return  
}
```

With this code only the first 4096 bytes of the request body will be read during `r.ParseForm()`. Trying to read beyond this limit will cause the `MaxBytesReader` to return an error, which will subsequently be surfaced by `r.ParseForm()`.

Additionally — if the limit is hit — `MaxBytesReader` sets a flag on `http.ResponseWriter` which instructs the server to close the underlying TCP connection.

Query string parameters

If you have a form that submits data using the HTTP method `GET`, rather than `POST`, the form data will be included as URL *query string parameters*. For example, if you have a HTML form that looks like this:

```
<form action='/foo/bar' method='GET'>
  <input type='text' name='title'>
  <input type='text' name='content'>

  <input type='submit' value='Submit'>
</form>
```

When the form is submitted, it will send a **GET** request with a URL that looks like this:
`/foo/bar?title=value&content=value`.

You can retrieve the values for the query string parameters in your handlers via the `r.URL.Query().Get()` method. This will always return a string value for a parameter, or the empty string `" "` if no matching parameter exists. For example:

```
func exampleHandler(w http.ResponseWriter, r *http.Request) {
    title := r.URL.Query().Get("title")
    content := r.URL.Query().Get("content")

    ...
}
```

The `r.Form` map

An alternative way to access query string parameters is via the `r.Form` map. This is similar to the `r.PostForm` map that we've used in this chapter, except that it contains the form data from any **POST** request body **and** any query string parameters.

Let's say that you have some code in your handler that looks like this:

```
err := r.ParseForm()
if err != nil {
    http.Error(w, "Bad Request", http.StatusBadRequest)
    return
}

title := r.Form.Get("title")
```

In this code, the line `r.Form.Get("title")` will return the `title` value from the **POST** request body *or* from a query string parameter with the name `title`. In the event of a conflict, the request body value will take precedent over the query string parameter.

Using `r.Form` can be very helpful if you want your application to be agnostic about how data values are passed to it. But outside of that scenario, `r.Form` doesn't offer any benefits and it is clearer and more explicit to read data from the **POST** request body via `r.PostForm` or from query string parameters via `r.URL.Query().Get()`.

Validating form data

Right now there's a glaring problem with our code: we're not validating the (untrusted) user input from the form in any way. We should do this to ensure that the form data is present, of the correct type and meets any business rules that we have.

Specifically for this form we want to:

- Check that the `title` and `content` fields are not empty.
- Check that the `title` field is not more than 100 characters long.
- Check that the `expires` value exactly matches one of our permitted values (`1`, `7` or `365` days).

All of these checks are fairly straightforward to implement using some `if` statements and various functions in Go's `strings` and `unicode/utf8` packages.

Open up your `handlers.go` file and update the `snippetCreatePost` handler to include the appropriate validation rules like so:

```
File: cmd/web/handlers.go

package main

import (
    "errors"
    "fmt"
    "net/http"
    "strconv"
    "strings"      // New import
    "unicode/utf8" // New import

    "snippetbox.alexandedwards.net/internal/models"
)

...

func (app *application) snippetCreatePost(w http.ResponseWriter, r *http.Request) {
    err := r.ParseForm()
    if err != nil {
        app.clientError(w, http.StatusBadRequest)
        return
    }

    title := r.PostForm.Get("title")
    content := r.PostForm.Get("content")

    expires, err := strconv.Atoi(r.PostForm.Get("expires"))
    if err != nil {
        app.clientError(w, http.StatusBadRequest)
        return
    }
}
```

```

}

// Initialize a map to hold any validation errors for the form fields.
fieldErrors := make(map[string]string)

// Check that the title value is not blank and is not more than 100
// characters long. If it fails either of those checks, add a message to the
// errors map using the field name as the key.
if strings.TrimSpace(title) == "" {
    fieldErrors["title"] = "This field cannot be blank"
} else if utf8.RuneCountInString(title) > 100 {
    fieldErrors["title"] = "This field cannot be more than 100 characters long"
}

// Check that the Content value isn't blank.
if strings.TrimSpace(content) == "" {
    fieldErrors["content"] = "This field cannot be blank"
}

// Check the expires value matches one of the permitted values (1, 7 or
// 365).
if expires != 1 && expires != 7 && expires != 365 {
    fieldErrors["expires"] = "This field must equal 1, 7 or 365"
}

// If there are any errors, dump them in a plain text HTTP response and
// return from the handler.
if len(fieldErrors) > 0 {
    fmt.Fprint(w, fieldErrors)
    return
}

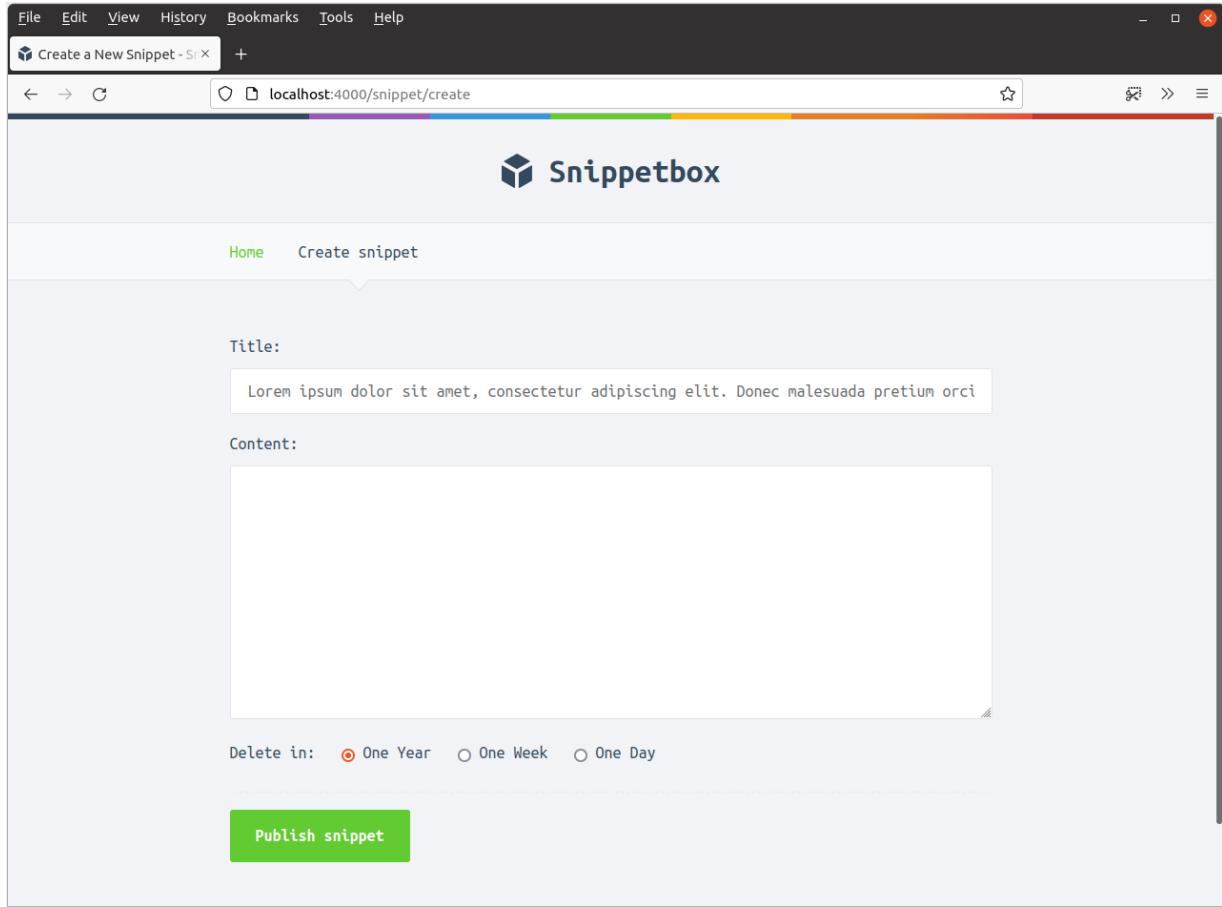
id, err := app.snippets.Insert(title, content, expires)
if err != nil {
    app.serverError(w, r, err)
    return
}

http.Redirect(w, r, fmt.Sprintf("/snippet/view/%d", id), http.StatusSeeOther)
}

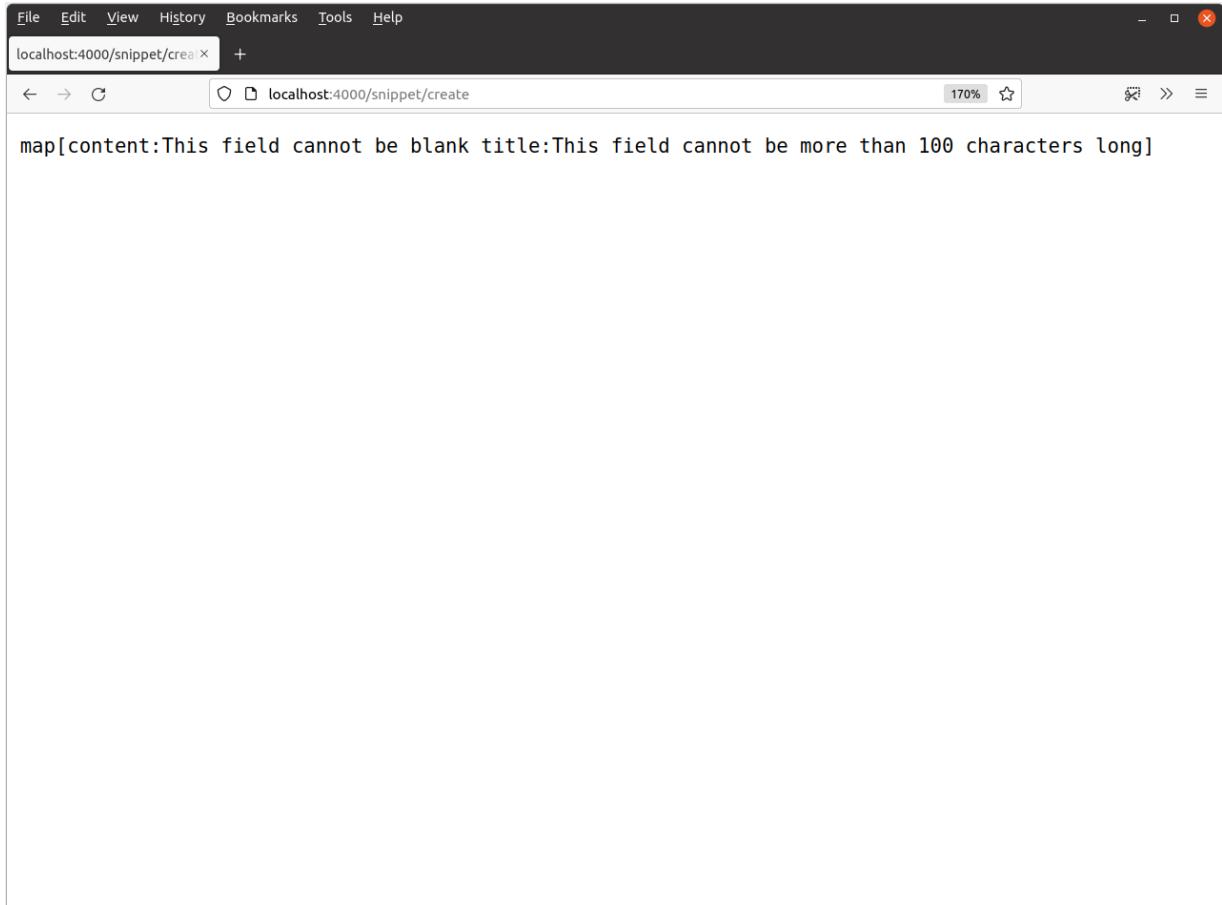
```

Note: When we check the length of the `title` field, we're using the `utf8.RuneCountInString()` function — not Go's `len()` function. This will count the number of *Unicode code points* in the title rather than the number of bytes. To illustrate the difference, the string "`Zoë`" contains 3 Unicode code points, but 4 bytes because of the umlauted `ë` character.

Alright, let's give this a try! Restart the application and try submitting the form with a too-long snippet title and blank content field, a bit like this...



And you should see a dump of the appropriate validation failure messages, like so:



Tip: You can find a bunch of code patterns for processing and validating different types of inputs in [this blog post](#).

Displaying errors and repopulating fields

Now that the `snippetCreatePost` handler is validating the data, the next stage is to manage these validation errors gracefully.

If there are any validation errors, we want to re-display the HTML form, highlighting the fields which failed validation and automatically re-populating any previously submitted data (so that the user doesn't need to enter it again).

To do this, let's begin by adding a new `Form` field to our `templateData` struct:

```
File: cmd/web/templates.go

package main

import (
    "html/template"
    "path/filepath"
    "time"

    "snippetbox.alexedwards.net/internal/models"
)

// Add a Form field with the type "any".
type templateData struct {
    CurrentYear int
    Snippet      models.Snippet
    Snippets     []models.Snippet
    Form         any
}
...
```

We'll use this `Form` field to pass the validation errors and previously submitted data back to the template when we re-display the form.

Next let's head back to our `cmd/web/handlers.go` file and define a new `snippetcreateForm` struct to hold the form data and any validation errors, and update our `snippetCreatePost` handler to use this.

Like so:

```
File: cmd/web/handlers.go
```

```

package main

...

// Define a snippetCreateForm struct to represent the form data and validation
// errors for the form fields. Note that all the struct fields are deliberately
// exported (i.e. start with a capital letter). This is because struct fields
// must be exported in order to be read by the html/template package when
// rendering the template.
type snippetCreateForm struct {
    Title      string
    Content    string
    Expires    int
    FieldErrors map[string]string
}

func (app *application) snippetCreatePost(w http.ResponseWriter, r *http.Request) {
    err := r.ParseForm()
    if err != nil {
        app.clientError(w, http.StatusBadRequest)
        return
    }

    // Get the expires value from the form as normal.
    expires, err := strconv.Atoi(r.PostForm.Get("expires"))
    if err != nil {
        app.clientError(w, http.StatusBadRequest)
        return
    }

    // Create an instance of the snippetCreateForm struct containing the values
    // from the form and an empty map for any validation errors.
    form := snippetCreateForm{
        Title:      r.PostForm.Get("title"),
        Content:    r.PostForm.Get("content"),
        Expires:    expires,
        FieldErrors: map[string]string{},
    }

    // Update the validation checks so that they operate on the snippetCreateForm
    // instance.
    if strings.TrimSpace(form.Title) == "" {
        form.FieldErrors["title"] = "This field cannot be blank"
    } else if utf8.RuneCountInString(form.Title) > 100 {
        form.FieldErrors["title"] = "This field cannot be more than 100 characters long"
    }

    if strings.TrimSpace(form.Content) == "" {
        form.FieldErrors["content"] = "This field cannot be blank"
    }

    if form.Expires != 1 && form.Expires != 7 && form.Expires != 365 {
        form.FieldErrors["expires"] = "This field must equal 1, 7 or 365"
    }

    // If there are any validation errors, then re-display the create tmpl template,
    // passing in the snippetCreateForm instance as dynamic data in the Form
    // field. Note that we use the HTTP status code 422 Unprocessable Entity
    // when sending the response to indicate that there was a validation error.
    if len(form.FieldErrors) > 0 {
        data := app.newTemplateData(r)
        data.Form = form
        app.render(w, r, http.StatusUnprocessableEntity, "create.tmpl", data)
        return
    }
}

```

```
// We also need to update this line to pass the data from the
// snippetCreateForm instance to our Insert() method.
id, err := app.snippets.Insert(form.Title, form.Content, form.Expires)
if err != nil {
    app.serverError(w, r, err)
    return
}

http.Redirect(w, r, fmt.Sprintf("/snippet/view/%d", id), http.StatusSeeOther)
}
```

OK, so now when there are any validation errors we are re-displaying the `create tmpl` template, passing in the previous data and validation errors in a `snippetcreateForm` struct via the template data's `Form` field.

If you like, you should be able to run the application at this point and the code should compile without any errors.

Updating the HTML template

The next thing that we need to do is update our `create tmpl` template to display the validation errors and re-populate any previous data.

Re-populating the form data is straightforward enough — we should be able to render this in the templates using tags like `{{ .Form.Title }}` and `{{ .Form.Content }}`, in the same way that we displayed the snippet data earlier in the book.

For the validation errors, the underlying type of our `FieldErrors` field is a `map[string]string`, which uses the form field names as keys. For maps, it's possible to access the value for a given key by simply chaining the key name. So, for example, to render a validation error for the `title` field we can use the tag `{{ .Form.FieldErrors.title }}` in our template.

Note: Unlike struct fields, map key names *don't* have to be capitalized in order to access them from a template.

With that in mind, let's update the `create tmpl` file to re-populate the data and display the error messages for each field, if they exist.

```
File: ui/html/pages/create tmpl
```

```
 {{define "title"}}Create a New Snippet{{end}}
```

```
 {{define "main"}}
<form action='/snippet/create' method='POST'>
  <div>
    <label>Title:</label>
    <!-- Use the `with` action to render the value of .Form.FieldErrors.title
    if it is not empty. -->
    {{with .Form.FieldErrors.title}}
      <label class='error'>{{.}}</label>
    {{end}}
    <!-- Re-populate the title data by setting the `value` attribute. -->
    <input type='text' name='title' value='{{.Form.Title}}'>
  </div>
  <div>
    <label>Content:</label>
    <!-- Likewise render the value of .Form.FieldErrors.content if it is not
    empty. -->
    {{with .Form.FieldErrors.content}}
      <label class='error'>{{.}}</label>
    {{end}}
    <!-- Re-populate the content data as the inner HTML of the textarea. -->
    <textarea name='content'>{{.Form.Content}}</textarea>
  </div>
  <div>
    <label>Delete in:</label>
    <!-- And render the value of .Form.FieldErrors.expires if it is not empty. -->
    {{with .Form.FieldErrors.expires}}
      <label class='error'>{{.}}</label>
    {{end}}
    <!-- Here we use the `if` action to check if the value of the re-populated
    expires field equals 365. If it does, then we render the `checked`
    attribute so that the radio input is re-selected. -->
    <input type='radio' name='expires' value='365' {{if (eq .Form.Expires 365)}}checked{{end}}> One Year
    <!-- And we do the same for the other possible values too... -->
    <input type='radio' name='expires' value='7' {{if (eq .Form.Expires 7)}}checked{{end}}> One Week
    <input type='radio' name='expires' value='1' {{if (eq .Form.Expires 1)}}checked{{end}}> One Day
  </div>
  <div>
    <input type='submit' value='Publish snippet'>
  </div>
</form>
{{end}}
```

Hopefully this markup and our use of Go's templating actions is generally clear — it's just using techniques that we've already [seen and discussed](#) earlier in the book.

There's one final thing we need to do. If we tried to run the application now, we would get a [500 Internal Server Error](#) when we first visit the form at <http://localhost:4000/snippet/create>. This is because our `snippetCreate` handler currently doesn't set a value for the `templateData.Form` field, meaning that when Go tries to evaluate a template tag like `{{with .Form.FieldErrors.title}}` it would result in an error because `Form` is `nil`.

Let's fix that by updating our `snippetCreate` handler so that it initializes a new

`createSnippetForm` instance and passes it to the template, like so:

```
File: cmd/web/handlers.go

package main

...

func (app *application) snippetCreate(w http.ResponseWriter, r *http.Request) {
    data := app.newTemplateData(r)

    // Initialize a new createSnippetForm instance and pass it to the template.
    // Notice how this is also a great opportunity to set any default or
    // 'initial' values for the form --- here we set the initial value for the
    // snippet expiry to 365 days.
    data.Form = snippetcreateForm{
        Expires: 365,
    }

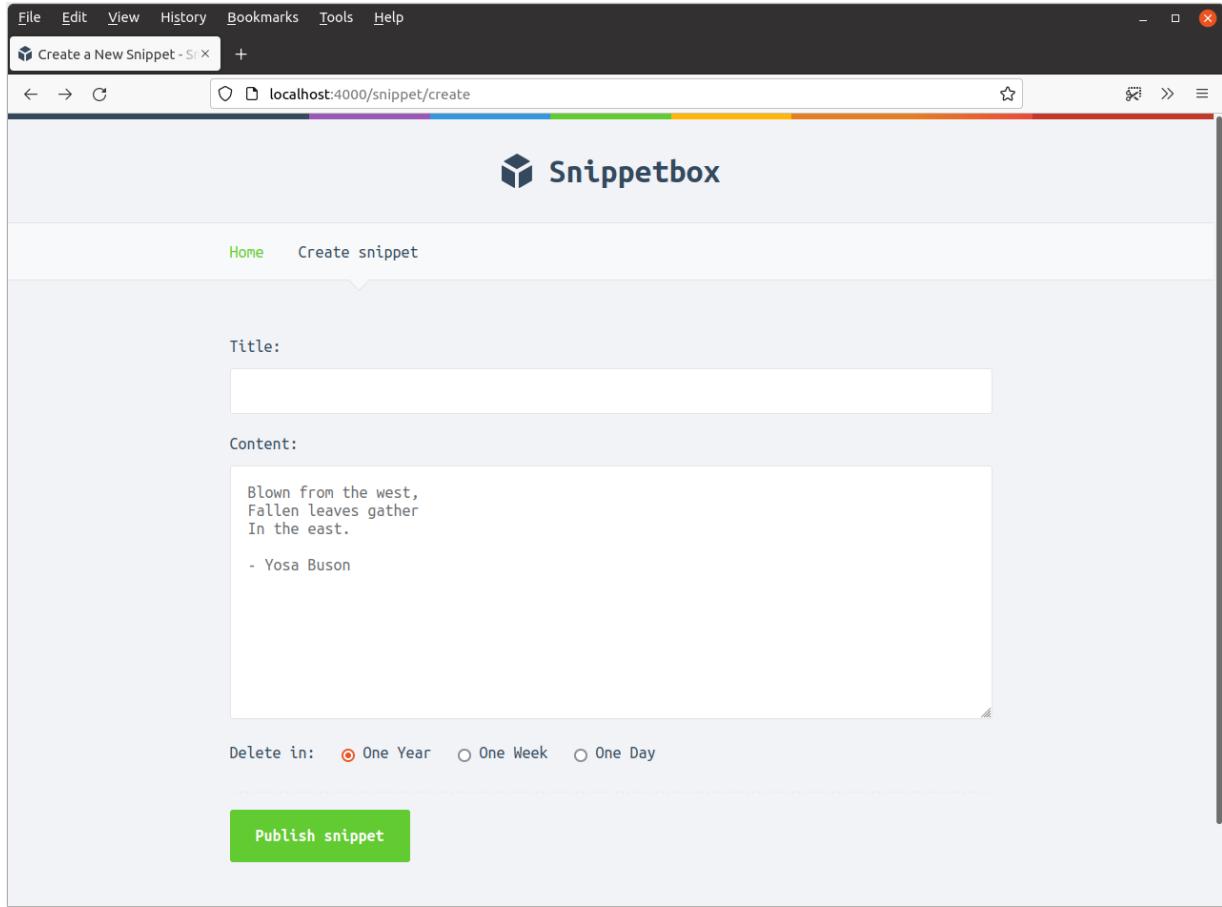
    app.render(w, r, http.StatusOK, "create tmpl", data)
}

...
```

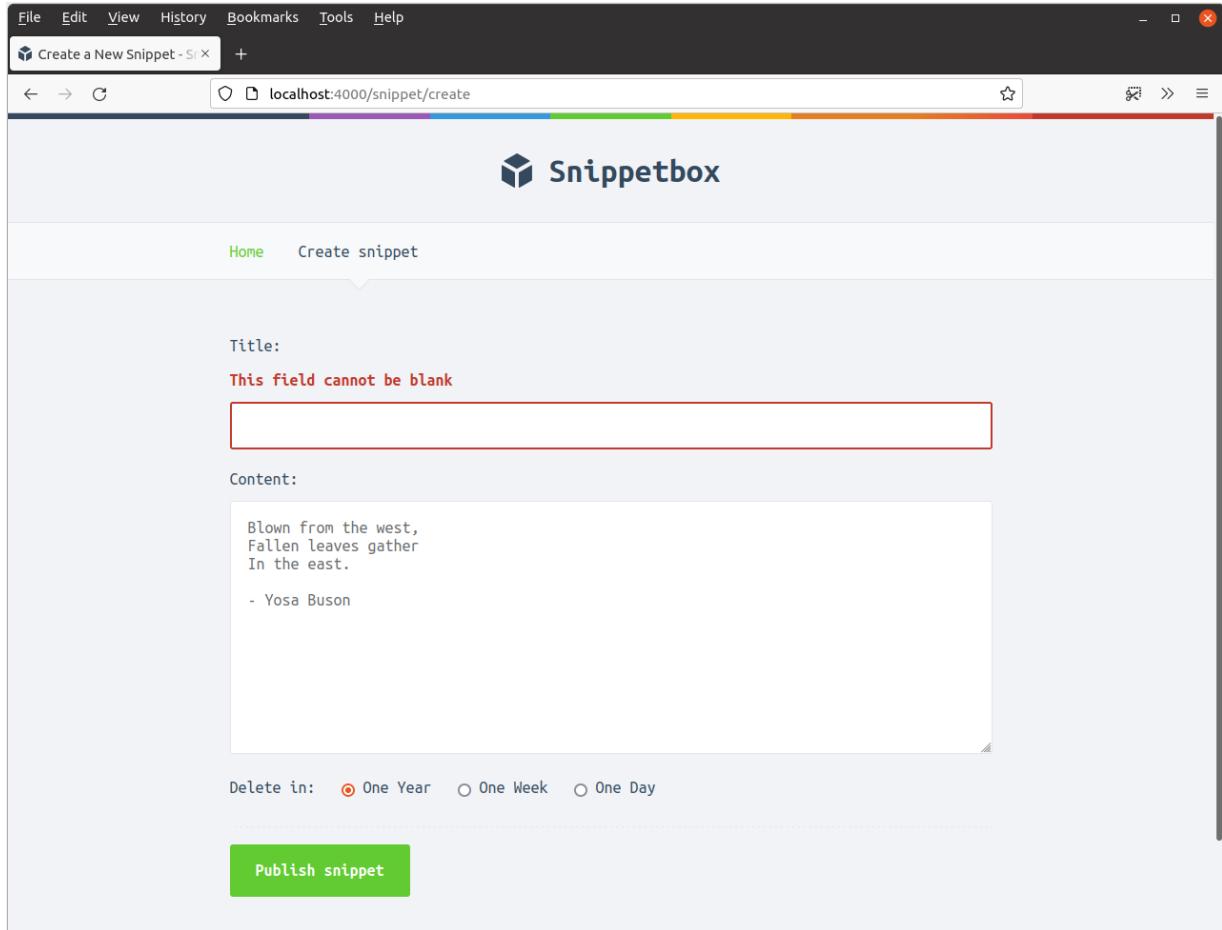
Now that's done, please restart the application and visit

<http://localhost:4000/snippet/create> in your browser. You should find that the page renders correctly without any errors.

The try adding some content and changing the default expiry time, but *leave the title field blank* like so:



After submission you should now see the form re-displayed, with the correctly re-populated snippet content and expiry option, and a “This field cannot be blank” error message alongside the title field:



Before we continue, feel free to spend some time playing around with the form and validation rules until you're confident that everything is working as you expect it to.

Additional information

Restful routing

If you've got a background in Ruby-on-Rails, Laravel or similar, you might be wondering why we haven't structured our routes and handlers to be more 'RESTful' and look like this:

Route pattern	Handler	Action
GET /snippets	snippetIndex	Display the home page
GET /snippets/{id}	snippetView	Display a specific snippet
GET /snippets/create	snippetCreate	Display a form for creating a new snippet
POST /snippets	snippetCreatePost	Save a new snippet

There are a couple of reasons.

The first reason is because of overlapping routes — a HTTP request to `/snippets/create` potentially matches both the `GET /snippets/{id}` and `GET /snippets/create` routes. In our application, the snippet ID values are always numeric so there will never be a ‘real’ overlap between these two routes — but imagine if our snippet ID values were user-generated, or a random 6-character string, and hopefully you can see the potential for a problem. Generally speaking, overlapping routes can be a source of bugs and unexpected behavior in your application, and it’s good practice to avoid them if you can — or use them with care and caution if you can’t.

The second reason is that the HTML form presented on `/snippets/create` would need to post to `/snippets` when submitted. This means that when we re-render the HTML form to show any validation errors, the URL in the user’s browser will also change to `/snippets`. YMMV on whether you consider this a problem or not — most users don’t look at URLs, but I think it’s a bit clunky and confusing in terms of UX... especially if a `GET` request to `/snippets` normally renders something else (like a list of all snippets).

Creating validation helpers

OK, so we're now in the position where our application is validating the form data according to our business rules and gracefully handling any validation errors. That's great, but it's taken quite a bit of work to get there.

And while the approach we've taken is fine as a one-off, if your application has *many forms* then you can end up with quite a lot of repetition in your code and validation rules. Not to mention, writing code for validating forms isn't exactly the most exciting way to spend your time.

So to help us with validation throughout the rest of this project, we'll create our own small `internal/validator` package to abstract some of this behavior and reduce the boilerplate code in our handlers. We won't actually change how the application works for the user at all; it's really just a refactoring of our codebase.

Adding a validator package

If you're coding-along, please go ahead and create the following directory and file on your machine:

```
$ mkdir internal/validator  
$ touch internal/validator/validator.go
```

Then in this new `internal/validator/validator.go` file add the following code:

File: internal/validator/validator.go

```
package validator

import (
    "slices"
    "strings"
    "unicode/utf8"
)

// Define a new Validator struct which contains a map of validation error messages
// for our form fields.
type Validator struct {
    FieldErrors map[string]string
}

// Valid() returns true if the FieldErrors map doesn't contain any entries.
func (v *Validator) Valid() bool {
    return len(v.FieldErrors) == 0
}

// AddFieldError() adds an error message to the FieldErrors map (so long as no
// entry already exists for the given key).
func (v *Validator) AddFieldError(key, message string) {
    // Note: We need to initialize the map first, if it isn't already
    // initialized.
    if v.FieldErrors == nil {
        v.FieldErrors = make(map[string]string)
    }

    if _, exists := v.FieldErrors[key]; !exists {
        v.FieldErrors[key] = message
    }
}

// CheckField() adds an error message to the FieldErrors map only if a
// validation check is not 'ok'.
func (v *Validator) CheckField(ok bool, key, message string) {
    if !ok {
        v.AddFieldError(key, message)
    }
}

// NotBlank() returns true if a value is not an empty string.
func NotBlank(value string) bool {
    return strings.TrimSpace(value) != ""
}

// MaxChars() returns true if a value contains no more than n characters.
func MaxChars(value string, n int) bool {
    return utf8.RuneCountInString(value) <= n
}

// PermittedValue() returns true if a value is in a list of specific permitted
// values.
func PermittedValue[T comparable](value T, permittedValues ...T) bool {
    return slices.Contains(permittedValues, value)
}
```

To summarize this:

In the code above we've defined a `Validator` struct type which contains a map of error

messages. The `Validator` type provides a `CheckField()` method for conditionally adding errors to the map, and a `Valid()` method which returns whether the errors map is empty or not. We've also added `NotBlank()`, `MaxChars()` and `PermittedValue()` functions to help us perform some specific validation checks

Note: the `PermittedValue()` function is a *generic function* which will work with values of different types. We'll talk about generics in more detail at the end of this chapter.

Conceptually this `Validator` type is quite basic, but that's not a bad thing. As we'll see over the course of this book, it's surprisingly powerful in practice and gives us a lot of flexibility and control over validation checks and how we perform them.

Using the helpers

Alright, let's start putting the `Validator` type to use!

We'll head back to our `cmd/web/handlers.go` file and update it to embed a `Validator` struct in our `snippetcreateForm` struct, and then use this to perform the necessary validation checks on the form data.

Tip: If you're not familiar with the concept of struct embedding in Go, Eli Bendersky has written a [good introduction](#) on the topic and I recommend quickly reading it before you continue.

Like so:

```
File: cmd/web/handlers.go

package main

import (
    "errors"
    "fmt"
    "net/http"
    "strconv"

    "snippetbox.alexedwards.net/internal/models"
    "snippetbox.alexedwards.net/internal/validator" // New import
)

...

// Remove the explicit FieldErrors struct field and instead embed the Validator
// struct. Embedding this means that our snippetcreateForm "inherits" all the
// fields and methods of our Validator struct (including the FieldErrors field).
```

```

type snippetcreateForm struct {
    Title          string
    Content        string
    Expires        int
    validator.Validator
}

func (app *application) snippetCreatePost(w http.ResponseWriter, r *http.Request) {
    err := r.ParseForm()
    if err != nil {
        app.clientError(w, http.StatusBadRequest)
        return
    }

    expires, err := strconv.Atoi(r.PostForm.Get("expires"))
    if err != nil {
        app.clientError(w, http.StatusBadRequest)
        return
    }

    form := snippetcreateForm{
        Title:  r.PostForm.Get("title"),
        Content: r.PostForm.Get("content"),
        Expires: expires,
        // Remove the FieldErrors assignment from here.
    }

    // Because the Validator struct is embedded by the snippetcreateForm struct,
    // we can call CheckField() directly on it to execute our validation checks.
    // CheckField() will add the provided key and error message to the
    // FieldErrors map if the check does not evaluate to true. For example, in
    // the first line here we "check that the form.Title field is not blank". In
    // the second, we "check that the form.Title field has a maximum character
    // length of 100" and so on.
    form.CheckFieldvalidator.NotBlank(form.Title), "title", "This field cannot be blank")
    form.CheckFieldvalidator.MaxChars(form.Title, 100), "title", "This field cannot be more than 100 characters long")
    form.CheckFieldvalidator.NotBlank(form.Content), "content", "This field cannot be blank")
    form.CheckFieldvalidator.PermittedValue(form.Expires, 1, 7, 365), "expires", "This field must equal 1, 7 or 365")

    // Use the Valid() method to see if any of the checks failed. If they did,
    // then re-render the template passing in the form in the same way as
    // before.
    if !form.Valid() {
        data := app.newTemplateData(r)
        data.Form = form
        app.render(w, r, http.StatusUnprocessableEntity, "create tmpl", data)
        return
    }

    id, err := app.snippets.Insert(form.Title, form.Content, form.Expires)
    if err != nil {
        app.serverError(w, r, err)
        return
    }

    http.Redirect(w, r, fmt.Sprintf("/snippet/view/%d", id), http.StatusSeeOther)
}

```

So this is shaping up really nicely.

We've now got an `internal/validator` package with validation rules and logic that can be reused across our application, and it can easily be extended to include additional rules in

the future. Both form data and errors are neatly encapsulated in a single `snippetcreateForm` struct — which we can easily pass to our templates — and the syntax for displaying error messages and re-populating the data in our templates is simple and consistent.

If you like, go ahead and re-run the application now. All being well, you should find that the form and validation rules are working correctly and in exactly the same manner as before.

Additional information

Generics

Go 1.18 was the first version of the language to support *generics* — also known by the more technical name of *parametric polymorphism*. Very broadly, generics allow you to write code that works with *different concrete types*.

For example, in older versions of Go, if you wanted to count how many times a particular value appears in a `[]string` slice and an `[]int` slice you would need to write two separate functions — one function for the `[]string` type and another for the `[]int`. A bit like this:

```
// Count how many times the value v appears in the slice s.
func countString(v string, s []string) int {
    count := 0
    for _, vs := range s {
        if v == vs {
            count++
        }
    }
    return count
}

func countInt(v int, s []int) int {
    count := 0
    for _, vs := range s {
        if v == vs {
            count++
        }
    }
    return count
}
```

Now, with generics, it's possible to write a single `count()` function that will work for `[]string`, `[]int`, or any other slice of a [comparable type](#). The code would look like this:

```
func count[T comparable](v T, s []T) int {
    count := 0
    for _, vs := range s {
        if v == vs {
            count++
        }
    }
    return count
}
```

If you're not familiar with the syntax for generic code in Go, there's a lot of great information available which explains how generics works and walks you through the syntax for writing generic code.

To get up to speed, I highly recommend reading the [official Go generics tutorial](#), and also watching the first 15 minutes of [this video](#) to help consolidate what you've learnt.

Rather than duplicating that same information here, instead I'd like to talk briefly about a less common (but just as important!) topic: *when* to use generics.

For now at least, you should aim to use generics *judiciously and cautiously*.

I know that might sound a bit boring, but generics are a relatively new language feature and best-practices around writing generic code are still being established. If you work on a team, or write code in public, it's also worth keeping in mind that not all other Go developers will necessarily be familiar with how generic code works.

You don't *need* to use generics, and it's OK not to.

But even with those caveats, writing generic code can be really useful in certain scenarios. Very generally speaking, you might want to consider it:

- If you find yourself writing repeated boilerplate code for different data types. Examples of this might be common operations on slices, maps or channels — or helpers for carrying out validation checks or test assertions on different data types.
- When you are writing code and find yourself reaching for the `any` (empty `interface{}`) type. An example of this might be when you are creating a data structure (like a queue, cache or linked list) which needs to operate on different types.

In contrast, you probably don't want to use generics:

- If it makes your code harder to understand or less clear.
- If all the types that you need to work with have a common set of methods — in which case it's better to define and use a normal `interface` type instead.
- Just *because you can*. Prefer instead write non-generic code by default, and switch to a

generic version later *only if it is actually needed*.

Automatic form parsing

We can simplify our `snippetCreatePost` handler further by using a third-party package like `go-playground/form` or `gorilla/schema` to automatically decode the form data into the `createSnippetForm` struct. Using an automatic decoder is *totally* optional, but it can help to save you time and typing — especially if your application has lots of forms, or you need to process a very large form.

In this chapter we'll look at how to use the `go-playground/form` package. If you're following along, please go ahead and install it like so:

```
$ go get github.com/go-playground/form/v4@v4
go get: added github.com/go-playground/form/v4 v4.2.1
```

Using the form decoder

To get this working, the first thing that we need to do is initialize a new `*form.Decoder` instance in our `main.go` file and make it available to our handlers as a dependency. Like this:

```
File: cmd/web/main.go

package main

import (
    "database/sql"
    "flag"
    "html/template"
    "log/slog"
    "net/http"
    "os"

    "snippetbox.alexedwards.net/internal/models"

    "github.com/go-playground/form/v4" // New import
    _ "github.com/go-sql-driver/mysql"
)

// Add a formDecoder field to hold a pointer to a form.Decoder instance.
type application struct {
    logger      *slog.Logger
    snippets   *models.SnippetModel
    templateCache map[string]*template.Template
    formDecoder *form.Decoder
}

func main() {
```

```

addr := flag.String("addr", ":4000", "HTTP network address")
dsn := flag.String("dsn", "web:pass@snippetbox?parseTime=true", "MySQL data source name")
flag.Parse()

logger := slog.New(slog.NewTextHandler(os.Stdout, nil))

db, err := openDB(*dsn)
if err != nil {
    logger.Error(err.Error())
    os.Exit(1)
}
defer db.Close()

templateCache, err := newTemplateCache()
if err != nil {
    logger.Error(err.Error())
    os.Exit(1)
}

// Initialize a decoder instance...
formDecoder := form.NewReader()

// And add it to the application dependencies.
app := &application{
    logger:      logger,
    snippets:    &models.SnippetModel{DB: db},
    templateCache: templateCache,
    formDecoder:   formDecoder,
}

logger.Info("starting server", "addr", *addr)

err = http.ListenAndServe(*addr, app.routes())
logger.Error(err.Error())
os.Exit(1)
}

...

```

Next let's go to our `cmd/web/handlers.go` file and update it to use this new decoder, like so:

File: cmd/web/handlers.go

```
package main

...

// Update our snippetCreateForm struct to include struct tags which tell the
// decoder how to map HTML form values into the different struct fields. So, for
// example, here we're telling the decoder to store the value from the HTML form
// input with the name "title" in the Title field. The struct tag `form:"-"`
// tells the decoder to completely ignore a field during decoding.
type snippetcreateForm struct {
    Title      string `form:"title"`
    Content    string `form:"content"`
    Expires    int    `form:"expires"`
    validator.Validator `form:"-"`
}

func (app *application) snippetCreatePost(w http.ResponseWriter, r *http.Request) {
    err := r.ParseForm()
    if err != nil {
        app.clientError(w, http.StatusBadRequest)
        return
    }

    // Declare a new empty instance of the snippetCreateForm struct.
    var form snippetcreateForm

    // Call the Decode() method of the form decoder, passing in the current
    // request and *a pointer* to our snippetCreateForm struct. This will
    // essentially fill our struct with the relevant values from the HTML form.
    // If there is a problem, we return a 400 Bad Request response to the client.
    err = app.formDecoder.Decode(&form, r.PostForm)
    if err != nil {
        app.clientError(w, http.StatusBadRequest)
        return
    }

    // Then validate and use the data as normal...
    form.CheckFieldvalidator.NotBlank(form.Title), "title", "This field cannot be blank")
    form.CheckFieldvalidator.MaxChars(form.Title, 100), "title", "This field cannot be more than 100 characters long")
    form.CheckFieldvalidator.NotBlank(form.Content), "content", "This field cannot be blank")
    form.CheckFieldvalidator.PermittedValue(form.Expires, 1, 7, 365), "expires", "This field must equal 1, 7 or 365")

    if !form.Valid() {
        data := app.newTemplateData(r)
        data.Form = form
        app.render(w, r, http.StatusUnprocessableEntity, "create tmpl", data)
        return
    }

    id, err := app.snippets.Insert(form.Title, form.Content, form.Expires)
    if err != nil {
        app.serverError(w, r, err)
        return
    }

    http.Redirect(w, r, fmt.Sprintf("/snippet/view/%d", id), http.StatusSeeOther)
}
```

Hopefully you can see the benefit of this pattern. We can use simple struct tags to define a mapping between our HTML form and the ‘destination’ struct fields, and unpacking the

form data to the destination now only requires us to write a few lines of code — irrespective of how large the form is.

Importantly, type conversions are handled automatically too. We can see that in the code above, where the `expires` value is automatically mapped to an `int` data type.

So that's really good. But there is one problem.

When we call `app.formDecoder.Decode()` it requires a *non-nil pointer* as the target decode destination. If we try to pass in something that *isn't* a non-nil pointer, then `Decode()` will return a `form.InvalidDecoderError` error.

If this ever happens, it's a critical problem with our application code (rather than a client error due to bad input). So we need to check for this error specifically and manage it as a special case, rather than just returning a `400 Bad Request` response.

Creating a `decodePostForm` helper

To assist with this, let's create a new `decodePostForm()` helper which does three things:

- Calls `r.ParseForm()` on the current request.
- Calls `app.formDecoder.Decode()` to unpack the HTML form data to a target destination.
- Checks for a `form.InvalidDecoderError` error and triggers a panic if we ever see it.

If you're following along, please go ahead and add this to your `cmd/web/helpers.go` file like so:

File: cmd/web/helpers.go

```
package main

import (
    "bytes"
    "errors" // New import
    "fmt"
    "net/http"
    "time"

    "github.com/go-playground/form/v4" // New import
)

...

// Create a new decodePostForm() helper method. The second parameter here, dst,
// is the target destination that we want to decode the form data into.
func (app *application) decodePostForm(r *http.Request, dst any) error {
    // Call ParseForm() on the request, in the same way that we did in our
    // snippetCreatePost handler.
    err := r.ParseForm()
    if err != nil {
        return err
    }

    // Call Decode() on our decoder instance, passing the target destination as
    // the first parameter.
    err = app.formDecoder.Decode(dst, r.PostForm)
    if err != nil {
        // If we try to use an invalid target destination, the Decode() method
        // will return an error with the type *form.InvalidDecoderError. We use
        // errors.As() to check for this and raise a panic rather than returning
        // the error.
        var invalidDecoderError *form.InvalidDecoderError

        if errors.As(err, &invalidDecoderError) {
            panic(err)
        }

        // For all other errors, we return them as normal.
        return err
    }

    return nil
}
```

And with that done, we can make the final simplification to our `snippetCreatePost` handler. Go ahead and update it to use the `decodePostForm()` helper and remove the `r.ParseForm()` call, so that the code looks like this:

```
File: cmd/web/handlers.go
```

```
package main

...

func (app *application) snippetCreatePost(w http.ResponseWriter, r *http.Request) {
    var form snippetcreateForm

    err := app.decodePostForm(r, &form)
    if err != nil {
        app.clientError(w, http.StatusBadRequest)
        return
    }

    form.CheckFieldvalidator.NotBlank(form.Title), "title", "This field cannot be blank")
    form.CheckFieldvalidator.MaxChars(form.Title, 100), "title", "This field cannot be more than 100 characters long")
    form.CheckFieldvalidator.NotBlank(form.Content), "content", "This field cannot be blank")
    form.CheckFieldvalidator.PermittedValue(form.Expires, 1, 7, 365), "expires", "This field must equal 1, 7 or 365")

    if !form.Valid() {
        data := app.newTemplateData(r)
        data.Form = form
        app.render(w, r, http.StatusUnprocessableEntity, "create tmpl", data)
        return
    }

    id, err := app.snippets.Insert(form.Title, form.Content, form.Expires)
    if err != nil {
        app.serverError(w, r, err)
        return
    }

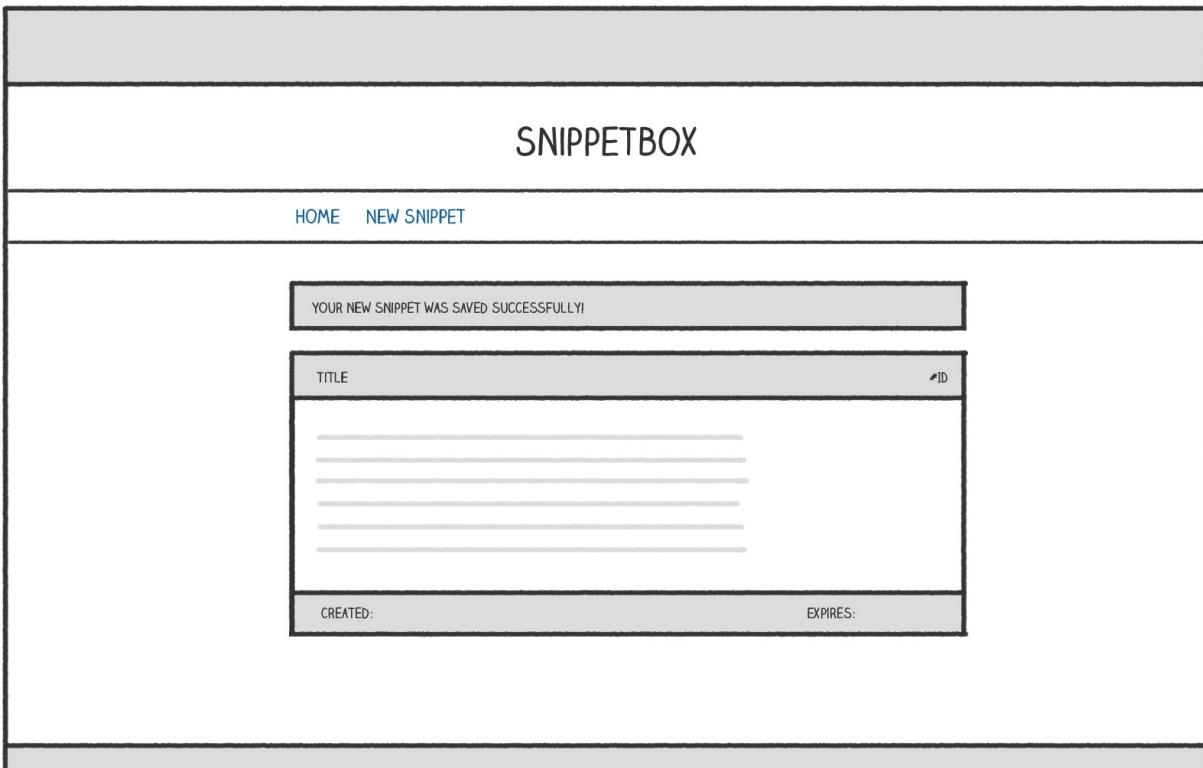
    http.Redirect(w, r, fmt.Sprintf("/snippet/view/%d", id), http.StatusSeeOther)
}
```

That's looking really good.

Our handler code is now nice and succinct, but still very clear in terms of it's behavior and what it is doing. And we have a general pattern in place for form processing and validation that we can easily re-use on other forms in our project — such as the user signup and login forms that we'll build shortly.

Stateful HTTP

A nice touch to improve our user experience would be to display a one-time confirmation message which the user sees *after* they've added a new snippet. Like so:



A confirmation message like this should only show up for the user once (immediately after creating the snippet) and no other users should ever see the message. If you've been programming for a while already, you might know this type of functionality as a flash message or a toast.

To make this work, we need to start sharing data (or state) between HTTP requests for the same user. The most common way to do that is to implement a session for the user.

In this section you'll learn:

- What [session managers](#) are available to help us implement sessions in Go.
- How to [use sessions](#) to safely and securely share data between requests for a particular user.
- How you can [customize session behavior](#) (including timeouts and cookie settings) based on your application's needs.

Choosing a session manager

There are a lot of [security considerations](#) when it comes to working with sessions, and proper implementation is not trivial. Unless you really need to roll your own implementation, it's a good idea to use an existing, well-tested, third-party package here.

I recommend using either [gorilla/sessions](#), or [alexewards/scs](#), depending on your project's needs.

- [gorilla/sessions](#) is the most established and well-known session management package for Go. It has a simple and easy-to-use API, and lets you store session data client-side (in signed and encrypted cookies) or server-side (in a database like MySQL, PostgreSQL or Redis).

However — importantly — it doesn't provide a mechanism to renew session IDs (which is necessary to reduce risks [associated with session fixation attacks](#) if you're using one of the server-side session stores).

- [alexewards/scs](#) lets you store session data server-side only. It supports automatic loading and saving of session data via middleware, has a nice interface for type-safe manipulation of data, and *does* allow renewal of session IDs. Like [gorilla/sessions](#), it also supports a variety of databases (including MySQL, PostgreSQL and Redis).

In summary, if you want to store session data client-side in a cookie then [gorilla/sessions](#) is a good choice, but otherwise [alexewards/scs](#) is generally the better option due to the ability to renew session IDs.

For this project we've already got a MySQL database set up, so we'll opt to use [alexewards/scs](#) and store the session data server-side in MySQL.

If you're following along, make sure that you're in your project directory and install the necessary packages like so:

```
$ go get github.com/alexewards/scs/v2@v2
go: downloading github.com/alexewards/scs/v2 v2.8.0
go get: added github.com/alexewards/scs/v2 v2.8.0

$ go get github.com/alexewards/scs/mysqlstore@latest
go: downloading github.com/alexewards/scs/mysqlstore v0.0.0-20240316133359-d7ab9d9831ec
go get: added github.com/alexewards/scs/mysqlstore v0.0.0-20240316133359-d7ab9d9831ec
```

Setting up the session manager

In this chapter I'll run through the basics of setting up and using the [alexewards/scs](#) package, but if you're going to use it in a production application I recommend reading the [documentation](#) and [API reference](#) to familiarize yourself with the full range of features.

The first thing we need to do is create a `sessions` table in our MySQL database to hold the session data for our users. Start by connecting to MySQL from your terminal window as the `root` user and execute the following SQL statement to setup the `sessions` table:

```
USE snippetbox;

CREATE TABLE sessions (
    token CHAR(43) PRIMARY KEY,
    data BLOB NOT NULL,
    expiry TIMESTAMP(6) NOT NULL
);

CREATE INDEX sessions_expiry_idx ON sessions (expiry);
```

In this table:

- The `token` field will contain a unique, randomly-generated, identifier for each session.
- The `data` field will contain the actual session data that you want to share between HTTP requests. This is stored as *binary data* in a `BLOB` (binary large object) type.
- The `expiry` field will contain an expiry time for the session. The `scs` package will automatically delete expired sessions from the `sessions` table so that it doesn't grow too large.

The next thing we need to do is establish a session manager in our `main.go` file and make it available to our handlers via the `application` struct. The session manager holds the configuration settings for our sessions, and also provides some middleware and helper methods to handle the loading and saving of session data.

Open your `main.go` file and update it as follows:

File: cmd/web/main.go

```
package main

import (
```

```

"database/sql"
"flag"
"html/template"
"log/slog"
"net/http"
"os"
"time" // New import

"snippetbox.alexedwards.net/internal/models"

"github.com/alexedwards/scs/mysqlstore" // New import
"github.com/alexedwards/scs/v2"         // New import
"github.com/go-playground/form/v4"
_ "github.com/go-sql-driver/mysql"

)

// Add a new sessionManager field to the application struct.
type application struct {
    logger      *slog.Logger
    snippets    *models.SnippetModel
    templateCache map[string]*template.Template
    formDecoder   *form.Decoder
    sessionManager *scs.SessionManager
}

func main() {
    addr := flag.String("addr", ":4000", "HTTP network address")
    dsn := flag.String("dsn", "web:pass@/snippetbox?parseTime=true", "MySQL data source name")
    flag.Parse()

    logger := slog.New(slog.NewTextHandler(os.Stdout, nil))

    db, err := openDB(*dsn)
    if err != nil {
        logger.Error(err.Error())
        os.Exit(1)
    }
    defer db.Close()

    templateCache, err := newTemplateCache()
    if err != nil {
        logger.Error(err.Error())
        os.Exit(1)
    }

    formDecoder := form.NewDecoder()

    // Use the scs.New() function to initialize a new session manager. Then we
    // configure it to use our MySQL database as the session store, and set a
    // lifetime of 12 hours (so that sessions automatically expire 12 hours
    // after first being created).
    sessionManager := scs.New()
    sessionManager.Store = mysqlstore.New(db)
    sessionManager.Lifetime = 12 * time.Hour

    // And add the session manager to our application dependencies.
    app := &application{
        logger:      logger,
        snippets:    &models.SnippetModel{DB: db},
        templateCache: templateCache,
        formDecoder:   formDecoder,
        sessionManager: sessionManager,
    }

    logger.Info("starting server", "addr", *addr)

    err = http.ListenAndServe(*addr, app.routes())
}

```

```
    ...  
    if err != nil {  
        logger.Error(err.Error())  
        os.Exit(1)  
    }  
  
    ...
```

Note: The `scs.New()` function returns a pointer to a `SessionManager` struct which holds the configuration settings for your sessions. In the code above we've set the `Store` and `Lifetime` fields of this struct, but there's a range of `other fields` that you can and should configure depending on your application's needs.

For the sessions to work, we also need to wrap our application routes with the middleware provided by the `SessionManager.LoadAndSave()` method. This middleware automatically loads and saves session data with every HTTP request and response.

It's important to note that we don't need this middleware to act on *all* our application routes. Specifically, we don't need it on the `GET /static/` route, because all this does is serve static files and there is no need for any stateful behavior.

So, because of that, it doesn't make sense to add the session middleware to our existing `standard` middleware chain.

Instead, let's create a new `dynamic` middleware chain containing the middleware appropriate for our dynamic application routes only.

Open the `routes.go` file and update it like so:

```
File: cmd/web/routes.go
```

```
package main

import (
    "net/http"

    "github.com/justinas/alice"
)

func (app *application) routes() http.Handler {
    mux := http.NewServeMux()

    // Leave the static files route unchanged.
    fileServer := http.FileServer(http.Dir("./ui/static/"))
    mux.Handle("GET /static/", http.StripPrefix("/static", fileServer))

    // Create a new middleware chain containing the middleware specific to our
    // dynamic application routes. For now, this chain will only contain the
    // LoadAndSave session middleware but we'll add more to it later.
    dynamic := alice.New(app.sessionManager.LoadAndSave)

    // Update these routes to use the new dynamic middleware chain followed by
    // the appropriate handler function. Note that because the alice ThenFunc()
    // method returns a http.Handler (rather than a http.HandlerFunc) we also
    // need to switch to registering the route using the mux.Handle() method.
    mux.Handle("GET /{$}", dynamic.ThenFunc(app.home))
    mux.Handle("GET /snippet/view/{id}", dynamic.ThenFunc(app.snippetView))
    mux.Handle("GET /snippet/create", dynamic.ThenFunc(app.snippetCreate))
    mux.Handle("POST /snippet/create", dynamic.ThenFunc(app.snippetCreatePost))

    standard := alice.New(app.recoverPanic, app.logRequest, commonHeaders)
    return standard.Then(mux)
}
```

If you run the application now you should find that it compiles all OK, and your application routes continue to work as normal.

Additional information

Without using alice

If you're not using the `justinas/alice` package to help manage your middleware chains, then you'd need to use the `http.HandlerFunc()` adapter to convert your handler functions like `app.home` to a `http.Handler`, and then wrap that with session middleware instead. Like this:

```
mux := http.NewServeMux()
mux.Handle("GET /{$}", app.sessionManager.LoadAndSave(http.Func(app.home)))
mux.Handle("GET /snippet/view/:id", app.sessionManager.LoadAndSave(http.HandlerFunc(app.snippetView)))
// ... etc
```

Working with session data

In this chapter let's put the session functionality to work and use it to persist the confirmation flash message between HTTP requests that we [discussed earlier](#).

We'll begin in our `cmd/web/handlers.go` file and update our `snippetCreatePost` method so that a flash message is added to the user's session data if — and only if — the snippet was created successfully. Like so:

```
File: cmd/web/handlers.go

package main

...

func (app *application) snippetCreatePost(w http.ResponseWriter, r *http.Request) {
    var form snippetcreateForm

    err := app.decodePostForm(r, &form)
    if err != nil {
        app.clientError(w, http.StatusBadRequest)
        return
    }

    form.CheckFieldvalidator.NotBlank(form.Title), "title", "This field cannot be blank")
    form.CheckFieldvalidator.MaxChars(form.Title, 100), "title", "This field cannot be more than 100 characters long")
    form.CheckFieldvalidator.NotBlank(form.Content), "content", "This field cannot be blank")
    form.CheckFieldvalidator.PermittedValue(form.Expires, 1, 7, 365), "expires", "This field must equal 1, 7 or 365")

    if !form.Valid() {
        data := app.newTemplateData(r)
        data.Form = form
        app.render(w, r, http.StatusUnprocessableEntity, "create tmpl", data)
        return
    }

    id, err := app.snippets.Insert(form.Title, form.Content, form.Expires)
    if err != nil {
        app.serverError(w, r, err)
        return
    }

    // Use the Put() method to add a string value ("Snippet successfully
    // created!") and the corresponding key ("flash") to the session data.
    app.sessionManager.Put(r.Context(), "flash", "Snippet successfully created!")

    http.Redirect(w, r, fmt.Sprintf("/snippet/view/%d", id), http.StatusSeeOther)
}
```

That's nice and simple, but there are a couple of things to point out:

- The first parameter that we pass to `app.sessionManager.Put()` is the *current request*

context. We'll talk properly about what the request context is and how to use it later in the book, but for now you can just think of it as somewhere that the session manager temporarily stores information while your handlers are dealing with the request.

- The second parameter (in our case the string "`flash`") is the *key* for the specific message that we are adding to the session data. We'll subsequently retrieve the message from the session data using this key too.
- If there's no existing session for the current user (or their session has expired) then a new, empty, session for them will automatically be created by the session middleware.

Next up we want our `snippetView` handler to retrieve the flash message (if one exists in the session for the current user) and pass it to the HTML template for subsequent display.

Because we want to display the flash message once only, we actually want to retrieve *and remove* the message from the session data. We can do both these operations at the same time by using the `PopString()` method.

I'll demonstrate:

File: cmd/web/handlers.go

```
package main

...

func (app *application) snippetView(w http.ResponseWriter, r *http.Request) {
    id, err := strconv.Atoi(r.PathValue("id"))
    if err != nil || id < 1 {
        http.NotFound(w, r)
        return
    }

    snippet, err := app.snippets.Get(id)
    if err != nil {
        if errors.Is(err, models.ErrNoRecord) {
            http.NotFound(w, r)
        } else {
            app.serverError(w, r, err)
        }
        return
    }

    // Use the PopString() method to retrieve the value for the "flash" key.
    // PopString() also deletes the key and value from the session data, so it
    // acts like a one-time fetch. If there is no matching key in the session
    // data this will return the empty string.
    flash := app.sessionManager.PopString(r.Context(), "flash")

    data := app.newTemplateData(r)
    data.Snippet = snippet

    // Pass the flash message to the template.
    data.Flash = flash

    app.render(w, r, http.StatusOK, "view tmpl", data)
}

...
```

Info: If you want to retrieve a value from the session data only (and leave it in there) you can use the `GetString()` method instead. The `scs` package also provides methods for retrieving other common data types, including `GetInt()`, `GetBool()`, `GetBytes()` and `GetTime()`.

If you try to run the application now, the compiler will (rightly) grumble that the `Flash` field isn't defined in our `templateData` struct. Go ahead and add it in like so:

```
File: cmd/web/templates.go
```

```
package main

import (
    "html/template"
    "path/filepath"
    "time"

    "snippetbox.alexedwards.net/internal/models"
)

type templateData struct {
    CurrentYear int
    Snippet     models.Snippet
    Snippets    []models.Snippet
    Form        any
    Flash       string // Add a Flash field to the templateData struct.
}

...
```

And now, we can update our `base tmpl` file to display the flash message, if one exists.

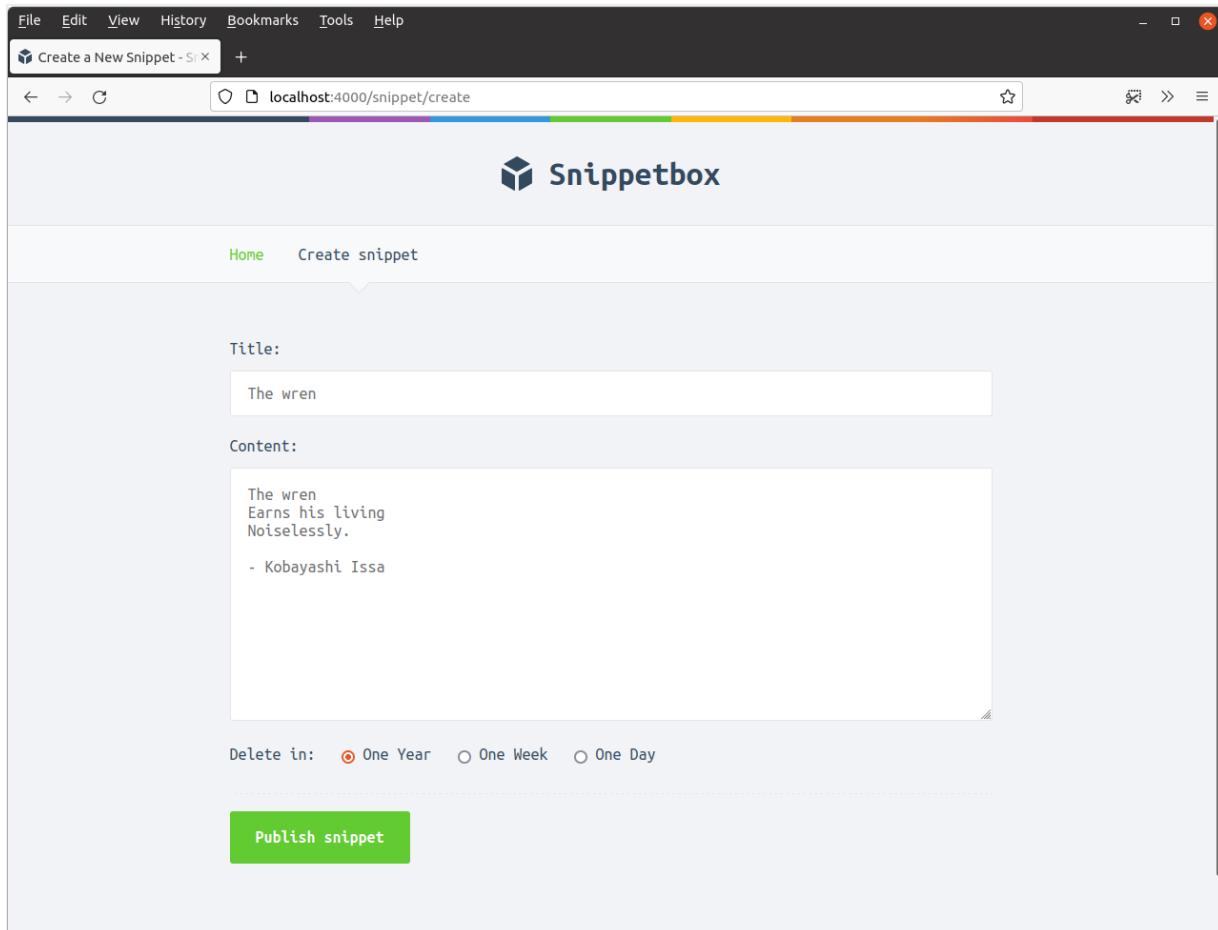
```
File: ui/html/base.tmpl
```

```
{{define "base"}}
<!doctype html>
<html lang='en'>
    <head>
        <meta charset='utf-8'>
        <title>{{template "title" .}} - Snippetbox</title>
        <link rel='stylesheet' href='/static/css/main.css'>
        <link rel='shortcut icon' href='/static/img/favicon.ico' type='image/x-icon'>
        <link rel='stylesheet' href='https://fonts.googleapis.com/css?family=Ubuntu+Mono:400,700'>
    </head>
    <body>
        <header>
            <h1><a href='/'>Snippetbox</a></h1>
        </header>
        {{template "nav" .}}
        <main>
            <!-- Display the flash message if one exists -->
            {{with .Flash}}
                <div class='flash'>{{.}}</div>
            {{end}}
            {{template "main" .}}
        </main>
        <footer>
            Powered by <a href='https://golang.org/'>Go</a> in {{.CurrentYear}}
        </footer>
        <script src='/static/js/main.js' type='text/javascript'></script>
    </body>
</html>
{{end}}
```

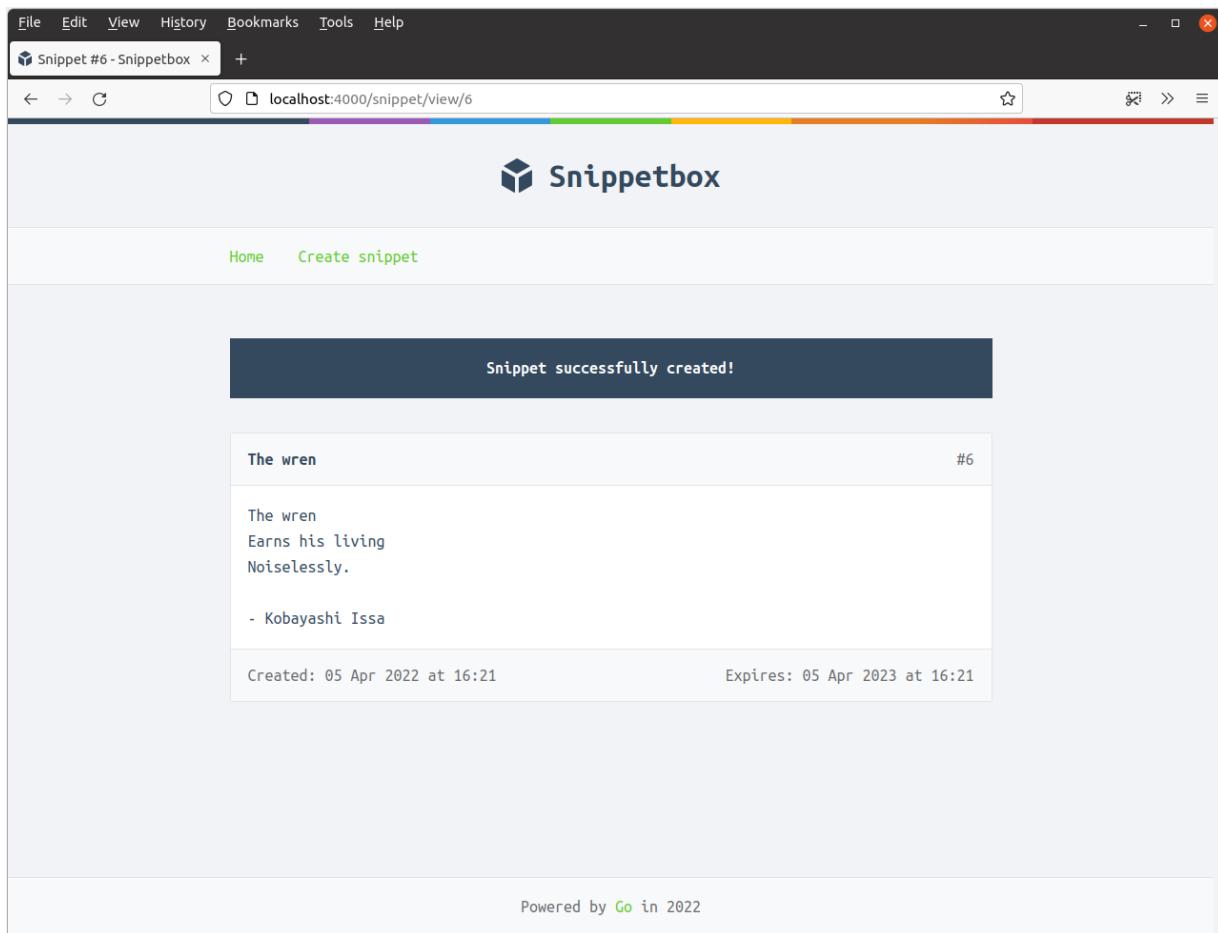
Remember, the `{{with .Flash}}` block will only be executed if the value of `.Flash` is not the empty string. So, if there's no "flash" key in the current user's session, the result is that

the chunk of new markup simply won't be displayed.

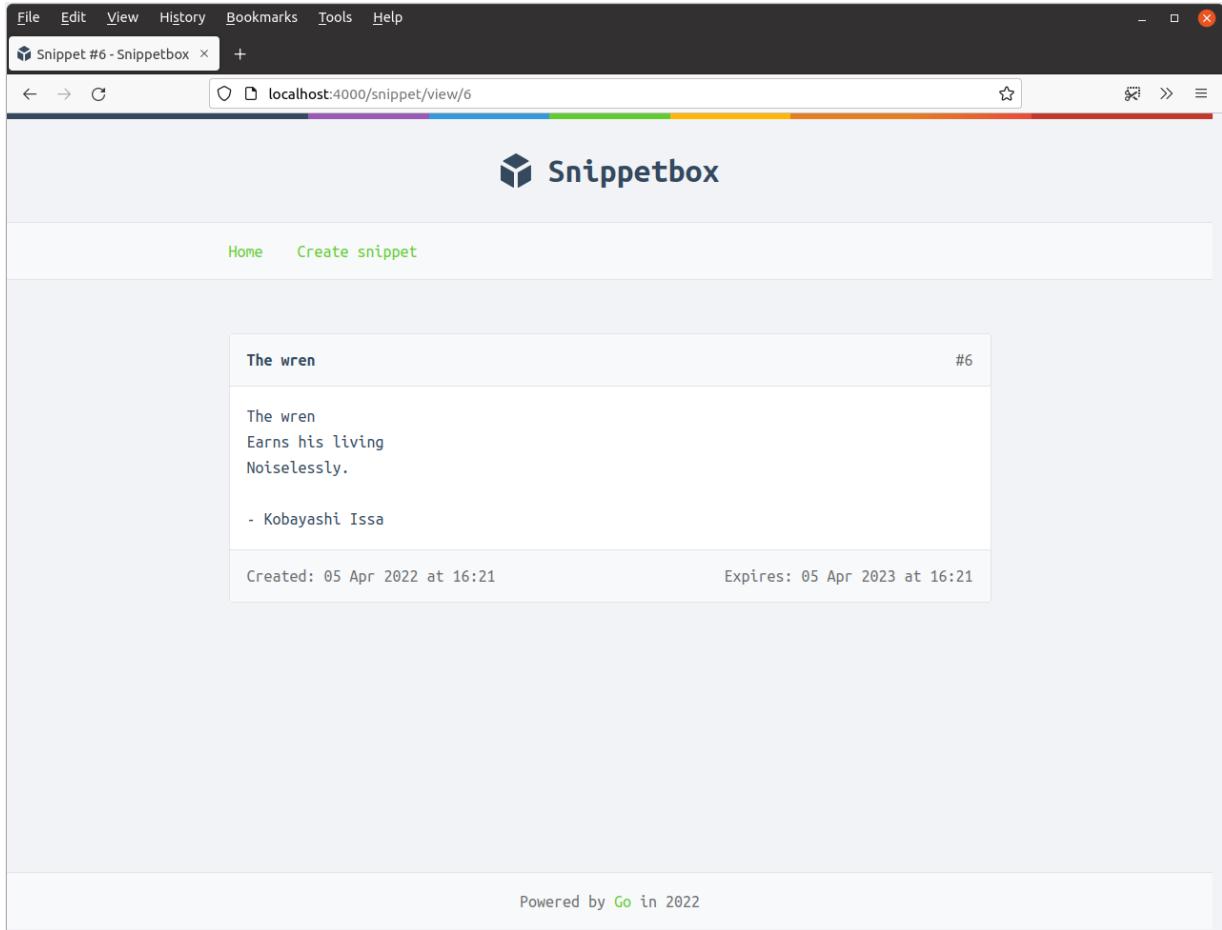
Once that's done, save all your files and restart the application. Try adding a new snippet like so...



And after redirection you should see the flash message now being displayed:



If you try refreshing the page, you can confirm that the flash message is no longer shown — it was a one-off message for the current user immediately after they created the snippet.



Auto-displaying flash messages

A little improvement we can make (which will save us some work later in the project) is to automate the display of flash messages, so that any message is automatically included the next time *any page is rendered*.

We can do this by adding any flash message to the template data via the `newTemplateData()` helper method that we made earlier, like so:

```
File: cmd/web/helpers.go
```

```
package main

...

func (app *application) newTemplateData(r *http.Request) templateData {
    return templateData{
        CurrentYear: time.Now().Year(),
        // Add the flash message to the template data, if one exists.
        Flash:       app.sessionManager.PopString(r.Context(), "flash"),
    }
}

...
```

Making that change means that we no longer need to check for the flash message within the `snippetView` handler, and the code can be reverted to look like this:

```
File: cmd/web/handlers.go
```

```
package main

...

func (app *application) snippetView(w http.ResponseWriter, r *http.Request) {
    id, err := strconv.Atoi(r.PathValue("id"))
    if err != nil || id < 1 {
        http.NotFound(w, r)
        return
    }

    snippet, err := app.snippets.Get(id)
    if err != nil {
        if errors.Is(err, models.ErrNoRecord) {
            http.NotFound(w, r)
        } else {
            app.serverError(w, r, err)
        }
        return
    }

    data := app.newTemplateData(r)
    data.Snippet = snippet

    app.render(w, r, http.StatusOK, "view tmpl", data)
}
```

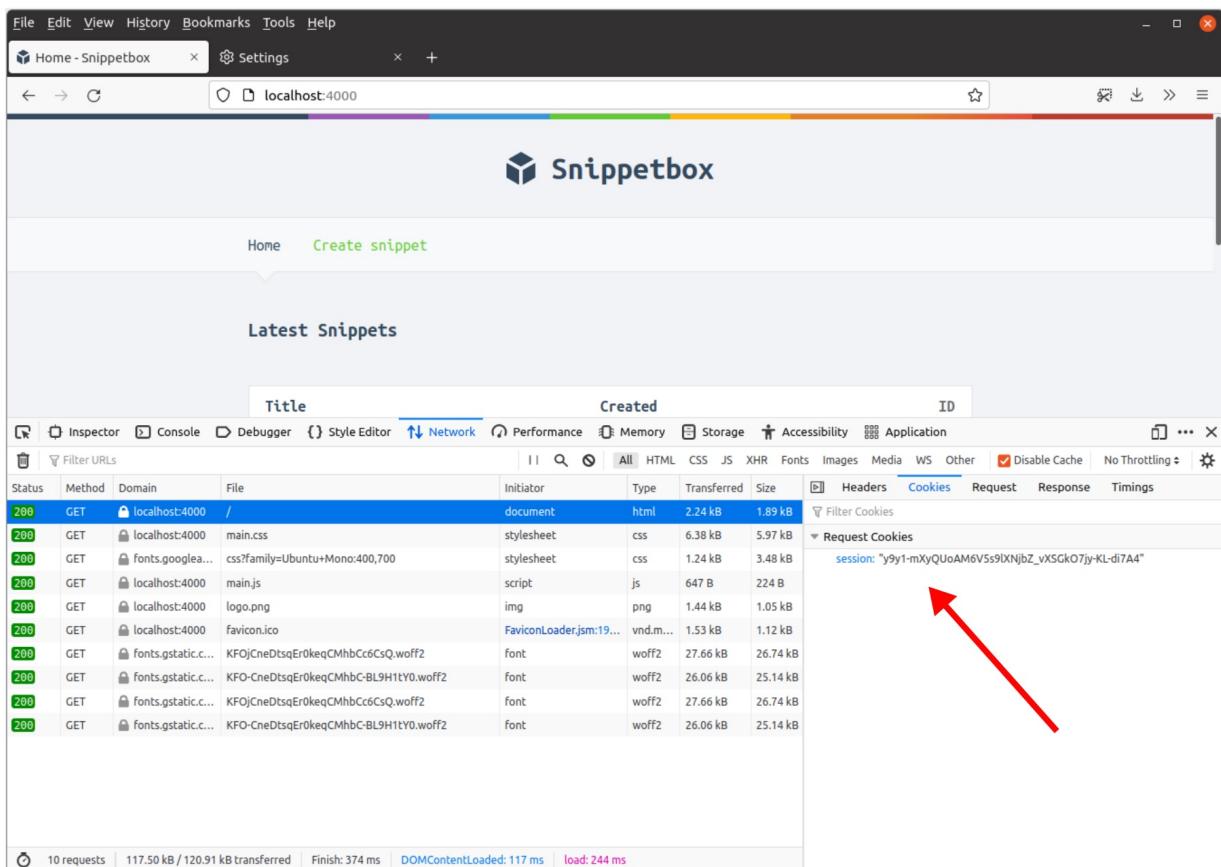
Feel free to try running the application again and creating another snippet. You should find that the flash message functionality still works as expected.

Additional information

Behind the scenes of session management

I'd like to take a moment to unpack some of the 'magic' behind session management and explain how it works behind the scenes.

If you like, open up the developer tools in your web browser and take a look at the cookie data for one of the pages. You should see a cookie named `session` in the request data, similar to this:



This is the *session cookie*, and it will be sent back to the Snippetbox application with every request that your browser makes.

The session cookie contains the *session token* — also sometimes known as the *session ID*. The session token is a high-entropy random string, which in my case is the value `y9y1-mXyQUoAM6V5s9lXNjbZ_vXSGk07jy-KL-di7A4` (yours will be different).

It's important to emphasize that the session token is just a random string. In itself, it doesn't carry or convey any *session data* (like the flash message that we set in this chapter).

Next, you might like to open up a terminal to MySQL and run a `SELECT` query against the

`sessions` table to lookup the session token that you see in your browser. Like so:

```
mysql> SELECT * FROM sessions WHERE token = 'y9y1-mXyQUoAM6V5s9lXNjbZ_vXSGk07jy-KL-di7A4';
+-----+-----+
| token          | data           |
+-----+-----+
| y9y1-mXyQUoAM6V5s9lXNjbZ_vXSGk07jy-KL-di7A4 | 0x26FF81030102FF820001020108446561646C696E6501FF8400010656616C75657301FF86000
+-----+-----+
1 row in set (0.00 sec)
```

This should return one record. The `data` value here is the thing that *actually contains the user's session data*. Specifically, what we're looking at is a MySQL **BLOB** (binary large object) containing a [gob-encoded](#) representation of the session data.

Each and every time we make a change to our session data, this `data` value will be updated to reflect the changes.

Lastly, the final column in the database is the `expiry` time, after which the session will no longer be considered valid.

So, what happens in our application is that the `LoadAndSave()` middleware checks each incoming request for a session cookie. If a session cookie is present, it reads the session token from the cookie and retrieves the corresponding session data from the database (while also checking that the session hasn't expired). It then adds the session data to the `request context` so it can be used in your handlers.

Any changes that you make to the session data in your handlers are updated in the request context, and then the `LoadAndSave()` middleware updates the database with any changes to the session data before it returns.

Server and security improvements

In this section of the book we're going to focus on making improvements to our application's HTTP server. You'll learn:

- How to customize server settings by using the [http.Server type](#).
- What the [server error log](#) is, and how to configure it to use your structured logger.
- How to quickly and easily [create a self-signed TLS certificate](#), using only Go.
- The fundamentals of setting up your application so that all requests and responses are [served securely over HTTPS](#).
- [Some sensible tweaks](#) to the default TLS settings to help keep user information secure and our server performing quickly.
- How to [set connection timeouts](#) to mitigate the impact of slow-client attacks.

The `http.Server` struct

So far in this book we've been using the `http.ListenAndServe()` shortcut function to start our server.

Although `http.ListenAndServe()` is very useful in short examples and tutorials, in real-world applications it's more common to manually create and use a `http.Server` struct instead. Doing this opens up the opportunity to customize the behavior of your server, which is exactly that we'll be doing in this section of the book.

So in preparation for that, let's quickly update our `main.go` file to stop using the `http.ListenAndServe()` shortcut, and manually create and use a `http.Server` struct instead.

```
File: cmd/web/main.go
```

```
package main

...

func main() {
    addr := flag.String("addr", ":4000", "HTTP network address")
    dsn := flag.String("dsn", "web:pass@/snippetbox?parseTime=true", "MySQL data source name")
    flag.Parse()

    logger := slog.New(slog.NewTextHandler(os.Stdout, nil))

    db, err := openDB(*dsn)
    if err != nil {
        logger.Error(err.Error())
        os.Exit(1)
    }
    defer db.Close()

    templateCache, err := newTemplateCache()
    if err != nil {
        logger.Error(err.Error())
        os.Exit(1)
    }

    formDecoder := form.NewReader()

    sessionManager := scs.New()
    sessionManager.Store = mysqlstore.New(db)
    sessionManager.Lifetime = 12 * time.Hour

    app := &application{
        logger:      logger,
        snippets:    &models.SnippetModel{DB: db},
        templateCache: templateCache,
        formDecoder:   formDecoder,
        sessionManager: sessionManager,
    }

    // Initialize a new http.Server struct. We set the Addr and Handler fields so
    // that the server uses the same network address and routes as before.
    srv := &http.Server{
        Addr:     *addr,
        Handler: app.routes(),
    }

    logger.Info("starting server", "addr", srv.Addr)

    // Call the ListenAndServe() method on our new http.Server struct to start
    // the server.
    err = srv.ListenAndServe()
    logger.Error(err.Error())
    os.Exit(1)
}

...
```

This is a small change which doesn't affect our application behavior (yet!), but it sets us up nicely for the work to come.

The server error log

It's important to be aware that Go's `http.Server` may write its own log entries relating to things like unrecovered panics, or problems accepting or writing to HTTP connections.

By default, it writes these entries using the standard logger — which means they will be written to the standard error stream (instead of standard out like our other log entries), and they won't be in the same format as the rest of our nice structured log entries.

To demonstrate this, let's add a deliberate error to our application and set a `Content-Length` header with an invalid value on our responses. Go ahead and update the `render()` helper like so:

```
File: cmd/web/helpers.go

package main

...

func (app *application) render(w http.ResponseWriter, r *http.Request, status int, page string, data templateData) {
    ts, ok := app.templateCache[page]
    if !ok {
        err := fmt.Errorf("the template %s does not exist", page)
        app.serverError(w, r, err)
        return
    }

    buf := new(bytes.Buffer)

    err := ts.ExecuteTemplate(buf, "base", data)
    if err != nil {
        app.serverError(w, r, err)
        return
    }

    // Deliberate error: set a Content-Length header with an invalid (non-integer)
    // value.
    w.Header().Set("Content-Length", "this isn't an integer!")

    w.WriteHeader(status)

    buf.WriteTo(w)
}

...
```

Then run the application, make a request to `http://localhost:4000`, and check the application log. You should see that it looks similar to this:

```
$ go run ./cmd/web/
time=2024-03-18T11:29:23.000+00:00 level=INFO msg="starting server" addr=:4000
time=2024-03-18T11:29:23.000+00:00 level=INFO msg="received request" ip=127.0.0.1:60824 proto=HTTP/1.1 method=GET uri=/
2024/03/18 11:29:23 http: invalid Content-Length of "this isn't an integer!"
time=2024-03-18T11:29:23.000+00:00 level=INFO msg="received request" ip=127.0.0.1:60824 proto=HTTP/1.1 method=GET uri=/static
time=2024-03-18T11:29:23.000+00:00 level=INFO msg="received request" ip=127.0.0.1:60830 proto=HTTP/1.1 method=GET uri=/static
```

We can see that the third log entry here, which informs us of the **Content-Length** problem, is in a very different format to the others. That's not great, especially if you want to filter your logs to look for errors, or use an external service to monitor them and send you alerts.

Unfortunately, it's not possible to configure `http.Server` to use our new structured logger directly. Instead, we have to convert our structured logger *handler* into a `*log.Logger` which writes log entries at a specific fixed level, and then register that with the `http.Server`. We can do this conversion using the `slog.NewLogLogger()` function, like so:

```
File: cmd/web/main.go

package main

...

func main() {
    addr := flag.String("addr", ":4000", "HTTP network address")
    dsn := flag.String("dsn", "web:pass@/snippetbox?parseTime=true", "MySQL data source name")
    flag.Parse()

    logger := slog.New(slog.NewTextHandler(os.Stdout, nil))

    ...

    srv := &http.Server{
        Addr:     *addr,
        Handler: app.routes(),
        // Create a *log.Logger from our structured logger handler, which writes
        // log entries at Error level, and assign it to the ErrorLog field. If
        // you would prefer to log the server errors at Warn level instead, you
        // could pass slog.LevelWarn as the final parameter.
        ErrorLog: slog.NewLogLogger(logger.Handler(), slog.LevelError),
    }

    logger.Info("starting server", "addr", srv.Addr)

    err = srv.ListenAndServe()
    logger.Error(err.Error())
    os.Exit(1)
}
```

With that in place, any log messages that `http.Server` automatically writes will now be written using our structured logger at `Error` level. If you restart the application and make another request to `http://localhost:4000`, you should see that the log entries now look

like this:

```
$ go run ./cmd/web/
time=2024-03-18T11:29:23.000+00:00 level=INFO msg="starting server" addr=:4000
time=2024-03-18T11:29:23.000+00:00 level=INFO msg="received request" ip=127.0.0.1:40854 proto=HTTP/1.1 method=GET uri=/
time=2024-03-18T11:29:23.000+00:00 level=ERROR msg="http: invalid Content-Length of \"this isn't an integer!\""
```

Before we go any further, head back to your `cmd/web/helpers.go` file and remove the deliberate error from the `render()` method:

```
File: cmd/web/helpers.go

package main

...

func (app *application) render(w http.ResponseWriter, r *http.Request, status int, page string, data templateData) {
    ts, ok := app.templateCache[page]
    if !ok {
        err := fmt.Errorf("the template %s does not exist", page)
        app.serverError(w, r, err)
        return
    }

    buf := new(bytes.Buffer)

    err := ts.ExecuteTemplate(buf, "base", data)
    if err != nil {
        app.serverError(w, r, err)
        return
    }

    w.WriteHeader(status)

    buf.WriteTo(w)
}

...
```

Generating a self-signed TLS certificate

Let's switch our attention to getting our application using HTTPS (rather than plain HTTP) for all requests and responses.

HTTPS is essentially HTTP sent across a TLS (*Transport Layer Security*) connection. The advantage to this is that HTTPS traffic is encrypted and signed, which helps ensure its privacy and integrity during transit.

Note: If you're not familiar with the term, TLS is essentially the modern version of SSL (*Secure Sockets Layer*). SSL now has been officially deprecated due to security concerns, but the name still lives on in the public consciousness and is often used interoperably with TLS. For clarity and accuracy, we'll stick with the term TLS throughout this book.

Before our server can start using HTTPS, we need to generate a *TLS certificate*.

For production servers I recommend using [Let's Encrypt](#) to create your TLS certificates, but for development purposes the simplest thing to do is to generate your own *self-signed certificate*.

A self-signed certificate is the same as a normal TLS certificate, except that it isn't cryptographically signed by a trusted certificate authority. This means that your web browser will raise a warning the first time it's used, but it will nonetheless encrypt HTTPS traffic correctly and is fine for development and testing purposes.

Handily, the `crypto/tls` package in Go's standard library includes a `generate_cert.go` tool that we can use to easily create our own self-signed certificate.

If you're following along, first create a new `tls` directory in the root of your project repository to hold the certificate and change into it:

```
$ cd $HOME/code/snippetbox
$ mkdir tls
$ cd tls
```

To run the `generate_cert.go` tool, you'll need to know the place on your computer where the source code for the Go standard library is installed. If you're using Linux, macOS or FreeBSD and followed the [official install instructions](#), then the `generate_cert.go` file should be located under `/usr/local/go/src/crypto/tls`.

If you're using macOS and installed Go using Homebrew, the file will probably be at `/usr/local/Cellar/go/<version>/libexec/src/crypto/tls/generate_cert.go` or a similar path.

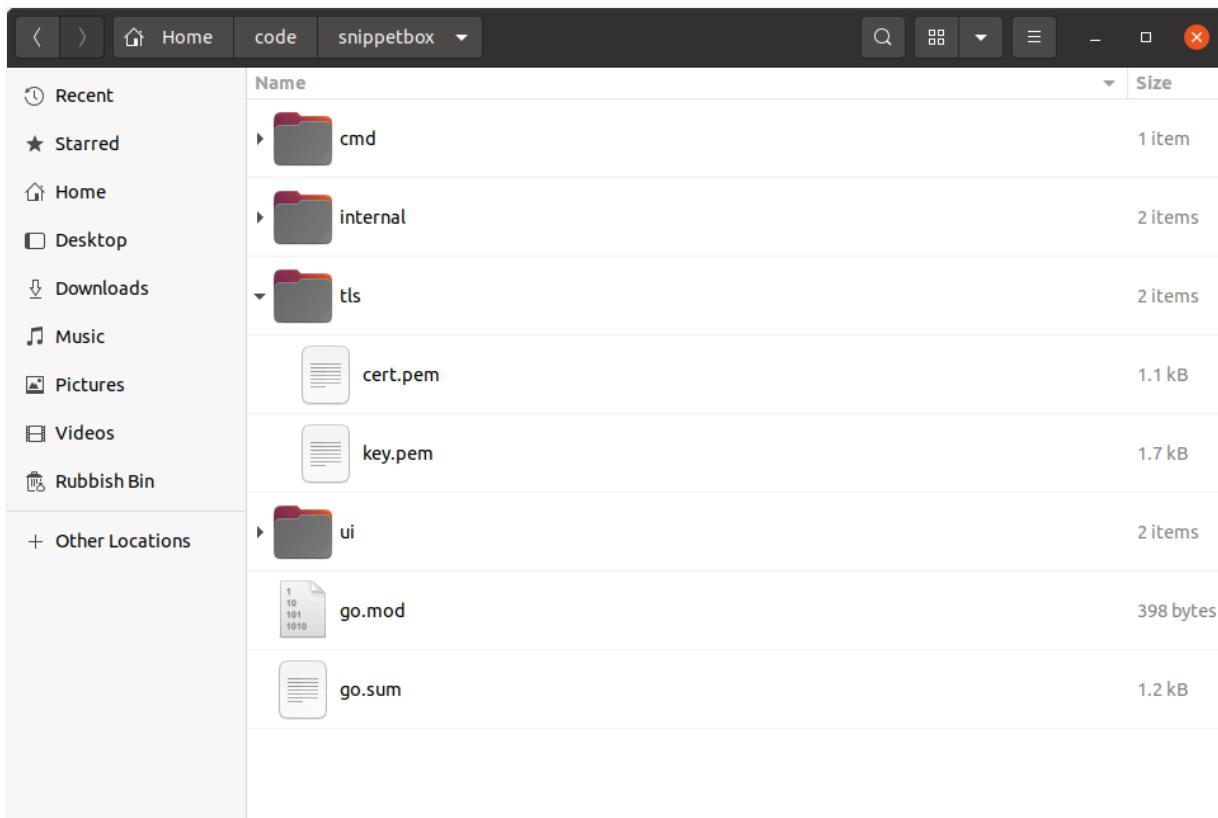
Once you know where it is located, you can then run the `generate_cert.go` tool like so:

```
$ go run /usr/local/go/src/crypto/tls/generate_cert.go --rsa-bits=2048 --host=localhost
2024/03/18 11:29:23 wrote cert.pem
2024/03/18 11:29:23 wrote key.pem
```

Behind the scenes, this `generate_cert.go` command works in two stages:

1. First it generates a [2048-bit](#) RSA key pair, which is a cryptographically secure [public key and private key](#).
2. It then stores the private key in a `key.pem` file, and generates a self-signed TLS certificate for the host `localhost` containing the public key — which it stores in a `cert.pem` file. Both the private key and certificate are PEM encoded, which is the standard format used by most TLS implementations.

Your project repository should now look something like this:



And that's it! We've now got a self-signed TLS certificate (and corresponding private key) that we can use during development.

Additional information

The `mkcert` tool

As an alternative to the `generate_cert.go` tool, you might want to consider using `mkcert` to generate the TLS certificates. Although this requires some extra set up, it has the advantage that the generated certificates are *locally trusted* — meaning that you can use them for testing and development without getting security warnings in your web browser.

Running a HTTPS server

Now that we have a self-signed TLS certificate and corresponding private key, starting a HTTPS web server is lovely and simple — we just need open the `main.go` file and swap the `srv.ListenAndServe()` method for `srv.ListenAndServeTLS()` instead.

Alter your `main.go` file to match the following code:

File: cmd/web/main.go

```
package main

...

func main() {
    addr := flag.String("addr", ":4000", "HTTP network address")
    dsn := flag.String("dsn", "web:pass@/snippetbox?parseTime=true", "MySQL data source name")
    flag.Parse()

    logger := slog.New(slog.NewTextHandler(os.Stdout, nil))

    db, err := openDB(*dsn)
    if err != nil {
        logger.Error(err.Error())
        os.Exit(1)
    }
    defer db.Close()

    templateCache, err := newTemplateCache()
    if err != nil {
        logger.Error(err.Error())
        os.Exit(1)
    }

    formDecoder := form.NewReader()

    sessionManager := scs.New()
    sessionManager.Store = mysqlstore.New(db)
    sessionManager.Lifetime = 12 * time.Hour
    // Make sure that the Secure attribute is set on our session cookies.
    // Setting this means that the cookie will only be sent by a user's web
    // browser when a HTTPS connection is being used (and won't be sent over an
    // unsecure HTTP connection).
    sessionManager.Cookie.Secure = true

    app := &application{
        logger:         logger,
        snippets:       &models.SnippetModel{DB: db},
        templateCache: templateCache,
        formDecoder:   formDecoder,
        sessionManager: sessionManager,
    }

    srv := &http.Server{
        Addr:     *addr,
        Handler: app.routes(),
        ErrorLog: slog.NewLogLogger(logger.Handler(), slog.LevelError),
    }

    logger.Info("starting server", "addr", srv.Addr)

    // Use the ListenAndServeTLS() method to start the HTTPS server. We
    // pass in the paths to the TLS certificate and corresponding private key as
    // the two parameters.
    err = srv.ListenAndServeTLS("./tls/cert.pem", "./tls/key.pem")
    logger.Error(err.Error())
    os.Exit(1)
}

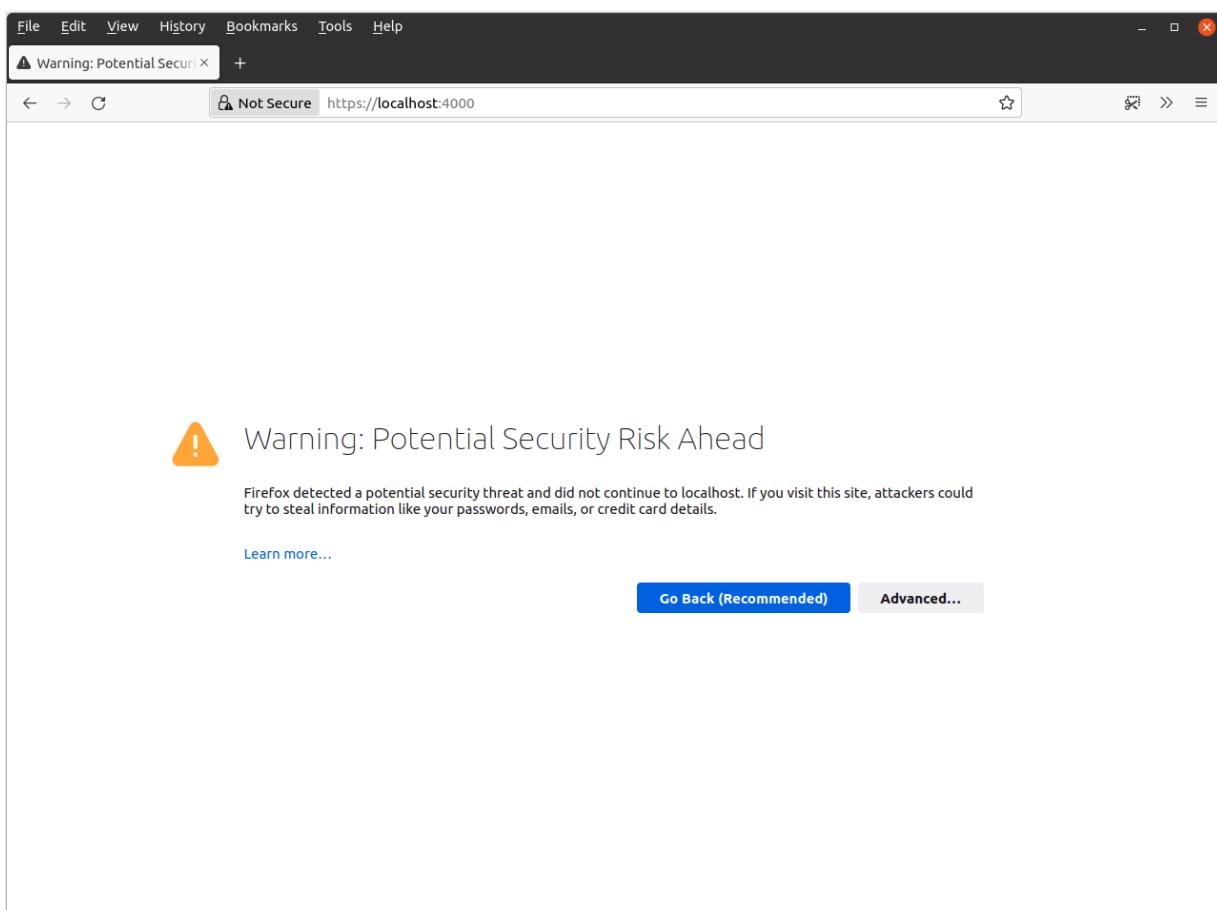
...
```

When we run this, our server will still be listening on port 4000 — the only difference is that it will now be talking HTTPS instead of HTTP.

Go ahead and run it as normal:

```
$ cd $HOME/code/snippetbox
$ go run ./cmd/web
time=2024-03-18T11:29:23.000+00:00 level=INFO msg="starting server" addr=:4000
```

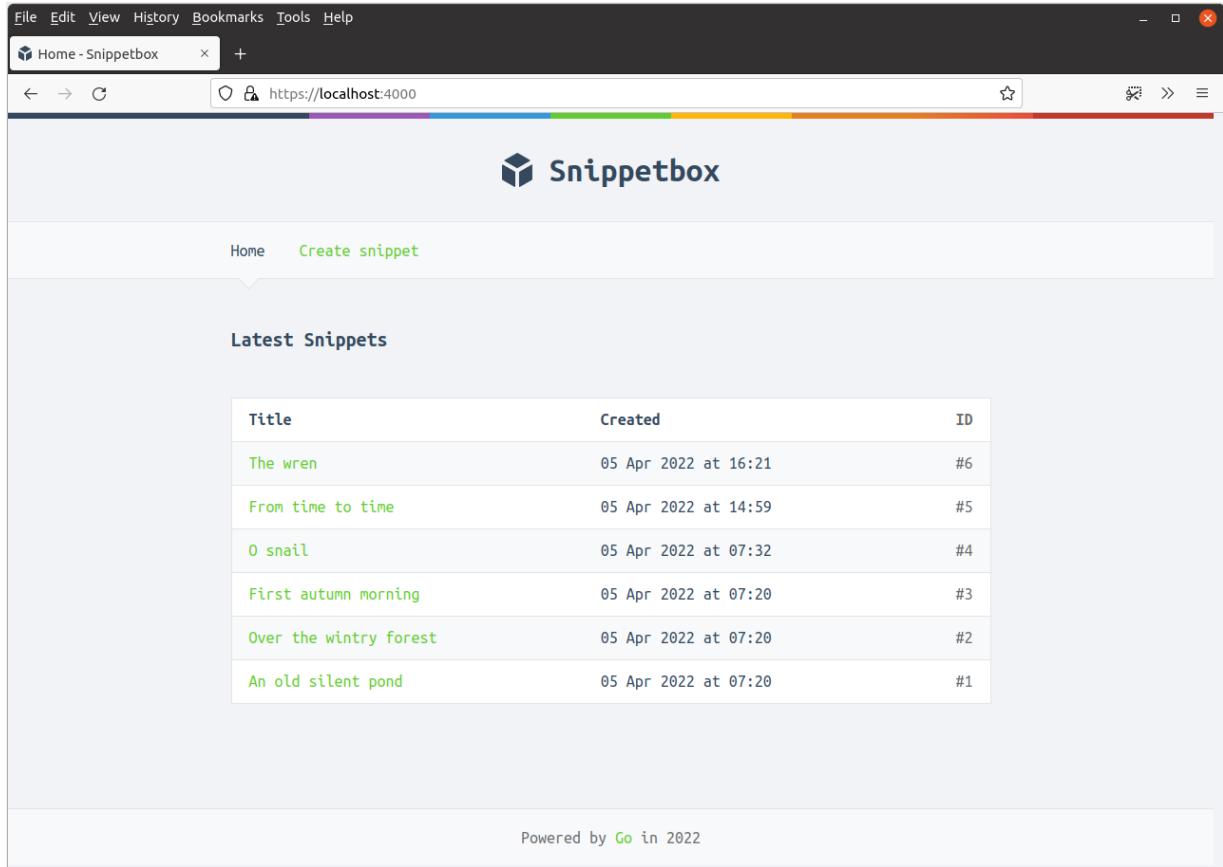
If you open up your web browser and visit <https://localhost:4000> you will probably get a browser warning because the TLS certificate is self-signed, similar to the screenshot below.



- If you're using Firefox like me, click “Advanced” then “Accept the Risk and Continue”.
- If you're using Chrome or Chromium, click “Advanced” and then the “Proceed to localhost (unsafe)” link.

After that the application homepage should appear (although it will still carry a warning in the URL bar).

In Firefox, it should look a bit like this:



Title	Created	ID
The wren	05 Apr 2022 at 16:21	#6
From time to time	05 Apr 2022 at 14:59	#5
O snail	05 Apr 2022 at 07:32	#4
First autumn morning	05 Apr 2022 at 07:20	#3
Over the wintry forest	05 Apr 2022 at 07:20	#2
An old silent pond	05 Apr 2022 at 07:20	#1

Powered by Go in 2022

If you're using Firefox, I recommend pressing **Ctrl+i** to inspect the Page Info for your homepage:

The screenshot shows the 'Page Info' dialog for a local HTTPS server. The 'Security' tab is active. In the 'Web Site Identity' section, it says the web site is 'localhost' and does not supply ownership information, verified by 'Acme Co' until April 2023. The 'Privacy & History' section shows 15,719 visits and cookie storage. The 'Technical Details' section confirms an encrypted connection using 'TLS_AES_128_GCM_SHA256' cipher suite and TLS 1.3.

The ‘Security > Technical Details’ section here confirms that our connection is encrypted and working as expected.

In my case, I can see that TLS version 1.3 is being used, and the cipher suite for my HTTPS connection is [TLS_AES_128_GCM_SHA256](#). We’ll talk more about cipher suites in the next chapter.

Aside: If you’re wondering who or what ‘Acme Co’ is in the Web Site Identity section of the screenshot above, it’s just a hard-coded placeholder name that the [generate_cert.go](#) tool uses.

Additional information

HTTP requests

It’s important to note that our HTTPS server *only supports HTTPS*. If you try making a regular HTTP request to it, the server will send the user a [400 Bad Request](#) status and the message

"Client sent an HTTP request to an HTTPS server" . Nothing will be logged.

```
$ curl -i http://localhost:4000/
HTTP/1.0 400 Bad Request

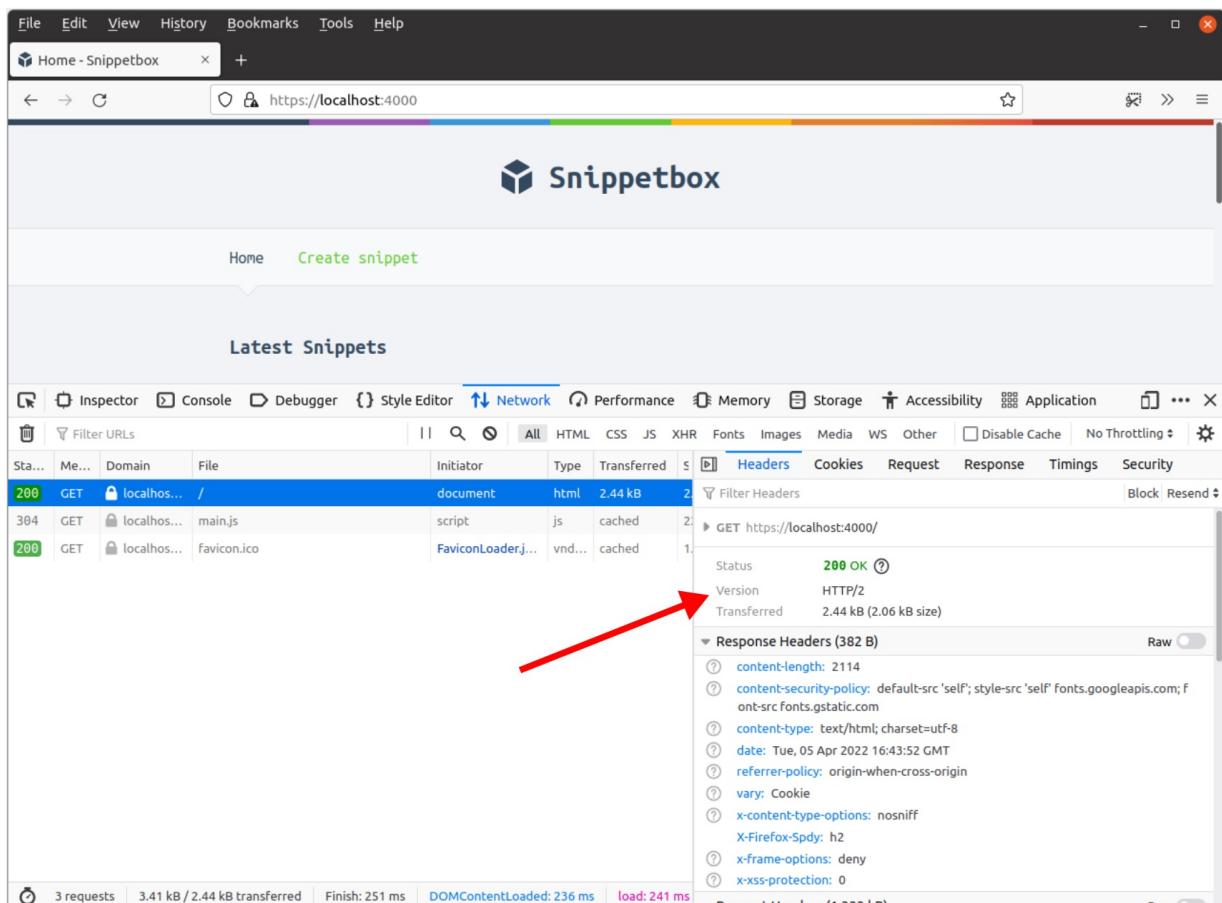
Client sent an HTTP request to an HTTPS server.
```

HTTP/2 connections

A big plus of using HTTPS is that Go's will automatically upgrade the connection to use HTTP/2 if the client supports it.

This is good because it means that, ultimately, our pages will load faster for users. If you're not familiar with HTTP/2 you can get a run-down of the basics and a flavor of how it has been implemented behind the scenes in Go in this [GoSF meetup talk](#) by Brad Fitzpatrick.

If you're using an up-to-date version of Firefox you should be able to see this in action. Press **Ctrl+Shift+E** to open the Developer Tools, and if you look at the headers for the homepage you should see that the protocol being used is HTTP/2.



Certificate permissions

It's important to note that the user that you're using to run your Go application must have read permissions for both the `cert.pem` and `key.pem` files, otherwise `ListenAndServeTLS()` will return a `permission denied` error.

By default, the `generate_cert.go` tool grants read permission to *all users* for the `cert.pem` file but read permission only to the *owner* of the `key.pem` file. In my case the permissions look like this:

```
$ cd $HOME/code/snippetbox/tls
$ ls -l
total 8
-rw-rw-r-- 1 alex alex 1090 Mar 18 16:24 cert.pem
-rw----- 1 alex alex 1704 Mar 18 16:24 key.pem
```

Generally, it's a good idea to keep the permissions of your private keys as tight as possible and allow them to be read only by the owner or a specific group.

Source control

If you're using a version control system (like Git or Mercurial) you may want to add an ignore rule so the contents of the `tls` directory are not accidentally committed. With Git, for instance:

```
$ cd $HOME/code/snippetbox
$ echo 'tls/' >> .gitignore
```

Configuring HTTPS settings

Go has good default settings for its HTTPS server, but it's possible to optimize and customize how the server behaves.

One change, which is almost always a good idea to make, is to restrict the *elliptic curves* that can potentially be used during the TLS handshake. Go supports a few elliptic curves, but as of Go 1.22 only `tls.CurveP256` and `tls.X25519` have assembly implementations. The others are very CPU intensive, so omitting them helps ensure that our server will remain performant under heavy loads.

To make this tweak, we can create a `tls.Config` struct containing our non-default TLS settings, and add it to our `http.Server` struct before we start the server.

I'll demonstrate:

File: cmd/web/main.go

```
package main

import (
    "crypto/tls" // New import
    "database/sql"
    "flag"
    "html/template"
    "log/slog"
    "net/http"
    "os"
    "time"

    "snippetbox.alexedwards.net/internal/models"

    "github.com/alexedwards/scs/mysqlstore"
    "github.com/alexedwards/scs/v2"
    "github.com/go-playground/form/v4"
    _ "github.com/go-sql-driver/mysql"
)

...

func main() {
    ...

    app := &application{
        logger:      logger,
        snippets:    &models.SnippetModel{DB: db},
        templateCache: templateCache,
        formDecoder:   formDecoder,
        sessionManager: sessionManager,
    }

    // Initialize a tls.Config struct to hold the non-default TLS settings we
    // want the server to use. In this case the only thing that we're changing
    // is the curve preferences value, so that only elliptic curves with
    // assembly implementations are used.
    tlsConfig := &tls.Config{
        CurvePreferences: []tls.CurveID{tls.X25519, tls.CurveP256},
    }

    // Set the server's TLSConfig field to use the tlsConfig variable we just
    // created.
    srv := &http.Server{
        Addr:      *addr,
        Handler:   app.routes(),
        ErrorLog:  slog.NewLogLogger(logger.Handler(), slog.LevelError),
        TLSConfig: tlsConfig,
    }

    logger.Info("starting server", "addr", srv.Addr)

    err = srv.ListenAndServeTLS("./tls/cert.pem", "./tls/key.pem")
    logger.Error(err.Error())
    os.Exit(1)
}

...
```

Additional information

TLS versions

By default, Go's HTTPS server is configured to support TLS 1.2 and 1.3. You can customize this and change the minimum and maximum TLS versions using the `tls.Config.MinVersion` and `MaxVersion` fields and [TLS versions constants](#) in the `crypto/tls` package.

For example, if you want the server to support TLS versions 1.0 to 1.2 only, you can use a configuration like so:

```
tlsConfig := &tls.Config{
    MinVersion: tls.VersionTLS10,
    MaxVersion: tls.VersionTLS12,
}
```

Restricting cipher suites

The cipher suites that Go supports are also defined in the [crypto/tls package constants](#).

For some applications, it may be desirable to limit your HTTPS server to only support a subset of the available cipher suites. For example, you might want to *only support* cipher suites which use ECDHE (forward secrecy) and *not support* weak cipher suites that use RC4, 3DES or CBC. You can do this via the `tls.Config.CipherSuites` field like so:

```
tlsConfig := &tls.Config{
    CipherSuites: []uint16{
        tls.TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384,
        tls.TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384,
        tls.TLS_ECDHE_ECDSA_WITH_CHACHA20_POLY1305,
        tls.TLS_ECDHE_RSA_WITH_CHACHA20_POLY1305,
        tls.TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256,
        tls.TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256,
    },
}
```

Go will automatically choose *which* of these cipher suites is actually used at runtime based on the cipher security, performance, and client/server hardware support.

Important: Restricting the supported cipher suites to only include strong, modern, ciphers can mean that users with certain older browsers won't be able to use your website. There's a balance to be struck between security and backwards-compatibility and the right decision for you will depend on the technology typically used by your user base. Mozilla's [recommended configurations](#) for modern, intermediate and old browsers may assist you in making a decision here.

It's also important (and interesting) to note that if a TLS 1.3 connection is negotiated, any `CipherSuites` field in your `tls.Config` will be ignored. The reason for this is that all the cipher suites that Go supports for TLS 1.3 connections are considered to be *safe*, so there isn't much point in providing a mechanism to configure them.

Basically, using `tls.Config` to set a custom list of supported cipher suites will affect TLS 1.0-1.2 connections only.

Connection timeouts

Let's take a moment to improve the resiliency of our server by adding some timeout settings, like so:

```
File: cmd/web/main.go

package main

...

func main() {
    ...

    tlsConfig := &tls.Config{
        CurvePreferences: []tls.CurveID{tls.X25519, tls.CurveP256},
    }

    srv := &http.Server{
        Addr:     *addr,
        Handler:  app.routes(),
        ErrorLog: slog.NewLogLogger(logger.Handler(), slog.LevelError),
        TLSConfig: tlsConfig,
        // Add Idle, Read and Write timeouts to the server.
        IdleTimeout: time.Minute,
        ReadTimeout: 5 * time.Second,
        WriteTimeout: 10 * time.Second,
    }

    logger.Info("starting server", "addr", srv.Addr)

    err = srv.ListenAndServeTLS("./tls/cert.pem", "./tls/key.pem")
    logger.Error(err.Error())
    os.Exit(1)
}

...
```

All three of these timeouts — `IdleTimeout`, `ReadTimeout` and `WriteTimeout` — are server-wide settings which act on the underlying connection and apply to all requests irrespective of their handler or URL.

The `IdleTimeout` setting

By default, Go enables [keep-alives](#) on all accepted connections. This helps reduce latency (especially for HTTPS connections) because a client can reuse the same connection for multiple requests without having to repeat the TLS handshake.

By default, keep-alive connections will be automatically closed after a couple of minutes (the exact time [depends on your operating system](#)). This helps to clear-up connections where the user has disappeared unexpectedly — e.g. due to a power cut on the client's end.

There is no way to *increase* this default (unless you roll your own `net.Listener`), but you can *reduce* it via the `IdleTimeout` setting. In our case, we've set `IdleTimeout` to 1 minute, which means that all keep-alive connections will be automatically closed after 1 minute of inactivity.

The ReadTimeout setting

In our code we've also set the `ReadTimeout` setting to 5 seconds. This means that if the request headers or body are still being read 5 seconds after the request is first accepted, then Go will close the underlying connection. Because this is a 'hard' closure on the connection, the user won't receive any HTTP(S) response.

Setting a short `ReadTimeout` period helps to mitigate the risk from slow-client attacks — such as [Slowloris](#) — which could otherwise keep a connection open indefinitely by sending partial, incomplete, HTTP(S) requests.

Important: If you set `ReadTimeout` but don't set `IdleTimeout`, then `IdleTimeout` will default to using the same setting as `ReadTimeout`. For instance, if you set `ReadTimeout` to 3 seconds, then there is the side-effect that all keep-alive connections will also be closed after 3 seconds of inactivity. Generally, my recommendation is to avoid any ambiguity and always set an explicit `IdleTimeout` value for your server.

The WriteTimeout setting

The `WriteTimeout` setting will close the underlying connection if our server attempts to write to the connection after a given period (in our code, 10 seconds). But this behaves slightly differently depending on the protocol being used.

- For HTTP connections, if some data is written to the connection more than 10 seconds after the *read of the request header* finished, Go will close the underlying connection instead of writing the data.
- For HTTPS connections, if some data is written to the connection more than 10 seconds after the request is *first accepted*, Go will close the underlying connection instead of

writing the data. This means that if you’re using HTTPS (like we are) it’s sensible to set `WriteTimeout` to a value greater than `ReadTimeout`.

It’s important to bear in mind that writes made by a handler are buffered and written to the connection as one when the handler returns. Therefore, the idea of `WriteTimeout` is generally *not* to prevent long-running handlers, but to prevent the data that the handler returns from taking too long to write.

Additional information

The `ReadHeaderTimeout` setting

`http.Server` also provides a `ReadHeaderTimeout` setting, which we haven’t used in our application. This works in a similar way to `ReadTimeout`, except that it applies to the read of the HTTP(S) headers only. So, if you set `ReadHeaderTimeout` to 3 seconds, a connection will be closed if the request headers are still being read 3 seconds after the request is accepted. However, reading of the request body can still take place after 3 seconds has passed, without the connection being closed.

This can be useful if you want to apply a server-wide limit to reading request headers, but want to implement different timeouts on different routes when it comes to reading the request body (possibly using the `http.TimeoutHandler()` middleware).

For our Snippetbox web application we don’t have any actions that warrant per-route read timeouts — reading the request headers and bodies for all our routes should be comfortably completed in 5 seconds, so we’ll stick to using `ReadTimeout`.

The `MaxHeaderBytes` setting

`http.Server` includes a `MaxHeaderBytes` field, which you can use to control the maximum number of bytes the server will read when parsing request headers. By default, Go allows a maximum header length of 1MB.

If you want to limit the maximum header length to 0.5MB, for example, you would write:

```
srv := &http.Server{
    Addr:          *addr,
    MaxHeaderBytes: 524288,
    ...
}
```

If `MaxHeaderBytes` is exceeded, then the user will automatically be sent a [431 Request Header Fields Too Large](#) response.

There's a gotcha to point out here: Go *always* adds an [additional 4096 bytes](#) of headroom to the figure you set. If you need `MaxHeaderBytes` to be a precise or very low number you'll need to factor this in.

User authentication

In this section of the book we're going to add some user authentication functionality to our application, so that only registered, logged-in users can create new snippets. Non-logged-in users will still be able to view the snippets, and will also be able to sign up for an account.

The workflow will look like this:

1. A user will register by visiting a form at `/user/signup` and entering their name, email address and password. We'll store this information in a new `users` database table (which we'll create in a moment).
2. A user will log in by visiting a form at `/user/login` and entering their email address and password.
3. We will then check the database to see if the email and password they entered match one of the users in the `users` table. If there's a match, the user has *authenticated* successfully and we add the relevant `id` value for the user to their session data, using the key "`authenticatedUserID`".
4. When we receive any subsequent requests, we can check the user's session data for a "`authenticatedUserID`" value. If it exists, we know that the user has already successfully logged in. We can keep checking this until the session expires, when the user will need to log in again. If there's no "`authenticatedUserID`" in the session, we know that the user is not logged in.

In many ways, a lot of the content in this section is just putting together the things that we've already learned in a different way. So it's a good litmus test of your understanding and a reminder of some key concepts.

You'll learn:

- How to implement basic `signup`, `login` and `logout` functionality for users.
- A secure approach to encrypting and `storing user passwords` in your database.
- A solid and straightforward approach to `verifying that a user is logged in` using middleware and sessions.
- How to `prevent cross-site request forgery` (CSRF) attacks.

Routes setup

Let's begin this section by adding five new routes to our application, so that it looks like this:

Route pattern	Handler	Action
GET /{\$}	home	Display the home page
GET /snippet/view/{id}	snippetView	Display a specific snippet
GET /snippet/create	snippetCreate	Display a form for creating a new snippet
POST /snippet/create	snippetCreatePost	Create a new snippet
GET /user/signup	userSignup	Display a form for signing up a new user
POST /user/signup	userSignupPost	Create a new user
GET /user/login	userLogin	Display a form for logging in a user
POST /user/login	userLoginPost	Authenticate and login the user
POST /user/logout	userLogoutPost	Logout the user
GET /static/	http.FileServer	Serve a specific static file

Note that the new state-changing handlers — `userSignupPost`, `userLoginPost` and `userLogoutPost` — are all using `POST` requests, not `GET`.

If you're following along, open up your `handlers.go` file and add placeholders for the five new handler functions as follows:

```

File: cmd/web/handlers.go

package main

...

func (app *application) userSignup(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintln(w, "Display a form for signing up a new user...")
}

func (app *application) userSignupPost(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintln(w, "Create a new user...")
}

func (app *application) userLogin(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintln(w, "Display a form for logging in a user...")
}

func (app *application) userLoginPost(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintln(w, "Authenticate and login the user...")
}

func (app *application) userLogoutPost(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintln(w, "Logout the user...")
}

```

Then when that's done, let's create the corresponding routes in the `routes.go` file:

```

File: cmd/web/routes.go

package main

...

func (app *application) routes() http.Handler {
    mux := http.NewServeMux()

    fileServer := http.FileServer(http.Dir("./ui/static/"))
    mux.Handle("GET /static/", http.StripPrefix("/static", fileServer))

    dynamic := alice.New(app.sessionManager.LoadAndSave)

    mux.Handle("GET /${}", dynamic.ThenFunc(app.home))
    mux.Handle("GET /snippet/view/{id}", dynamic.ThenFunc(app.snippetView))
    mux.Handle("GET /snippet/create", dynamic.ThenFunc(app.snippetCreate))
    mux.Handle("POST /snippet/create", dynamic.ThenFunc(app.snippetCreatePost))

    // Add the five new routes, all of which use our 'dynamic' middleware chain.
    mux.Handle("GET /user/signup", dynamic.ThenFunc(app.userSignup))
    mux.Handle("POST /user/signup", dynamic.ThenFunc(app.userSignupPost))
    mux.Handle("GET /user/login", dynamic.ThenFunc(app.userLogin))
    mux.Handle("POST /user/login", dynamic.ThenFunc(app.userLoginPost))
    mux.Handle("POST /user/logout", dynamic.ThenFunc(app.userLogoutPost))

    standard := alice.New(app.recoverPanic, app.logRequest, commonHeaders)
    return standard.Then(mux)
}

```

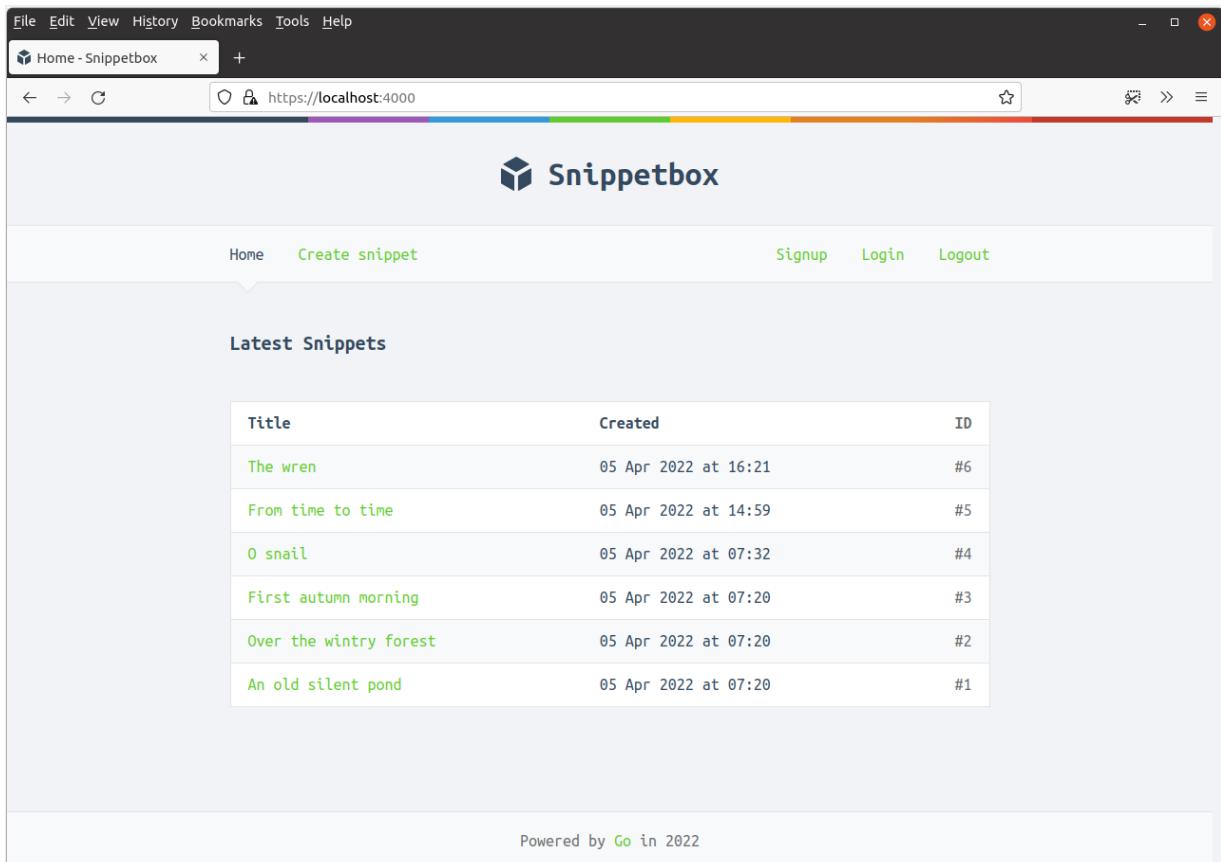
Finally, we'll also need to update the `nav.tmpl` partial to include navigation items for the

new pages:

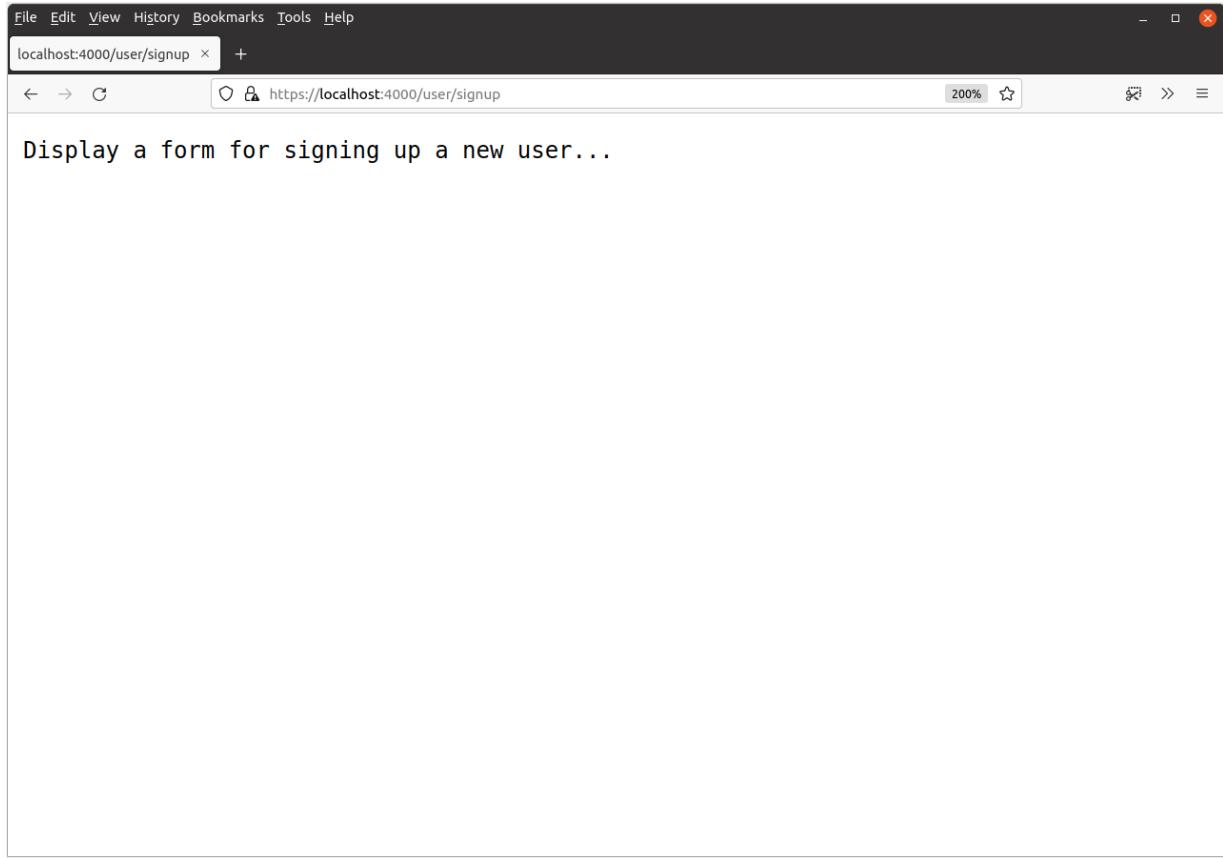
```
File: ui/html/partials/nav.tpl

{{define "nav"}}
<nav>
  <div>
    <a href='/'>Home</a>
    <a href='/snippet/create'>Create snippet</a>
  </div>
  <div>
    <a href='/user/signup'>Signup</a>
    <a href='/user/login'>Login</a>
    <form action='/user/logout' method='POST'>
      <button>Logout</button>
    </form>
  </div>
</nav>
{{end}}
```

If you like, you can run the application at this point and you should see the new items in the navigation bar like this:



If you click the new links, they should respond with the relevant placeholder plain-text response. For example, if you click the 'Signup' link you should see a response similar to this:



Display a form for signing up a new user...

Creating a users model

Now that the routes are set up, we need to create a new `users` database table and a database model to access it.

Start by connecting to MySQL from your terminal window as the `root` user and execute the following SQL statement to setup the `users` table:

```
USE snippetbox;

CREATE TABLE users (
    id INTEGER NOT NULL PRIMARY KEY AUTO_INCREMENT,
    name VARCHAR(255) NOT NULL,
    email VARCHAR(255) NOT NULL,
    hashed_password CHAR(60) NOT NULL,
    created DATETIME NOT NULL
);

ALTER TABLE users ADD CONSTRAINT users_uc_email UNIQUE (email);
```

There's a couple of things worth pointing out about this table:

- The `id` field is an autoincrementing integer field and the primary key for the table. This means that the user ID values are guaranteed to be unique positive integers (1, 2, 3... etc).
- The type of the `hashed_password` field is `CHAR(60)`. This is because we'll be storing bcrypt hashes of the user passwords in the database — not the passwords themselves — and the hashes are always exactly 60 characters long.
- We've also added a `UNIQUE` constraint on the `email` column and named it `users_uc_email`. This constraint ensures that we won't end up with two users who have the same email address. If we try to insert a record in this table with a duplicate email, MySQL will throw an `ERROR 1062: ER_DUP_ENTRY` error.

Building the model in Go

Next let's setup a model so that we can easily work with the new `users` table. We'll follow the same pattern that we used earlier in the book for modeling access to the `snippets` table, so hopefully this should feel familiar and straightforward.

First, open up the `internal/models/errors.go` file that you created earlier and define a couple of new error types:

```
File: internal/models/errors.go

package models

import (
    "errors"
)

var (
    ErrNoRecord = errors.New("models: no matching record found")

    // Add a new ErrInvalidCredentials error. We'll use this later if a user
    // tries to login with an incorrect email address or password.
    ErrInvalidCredentials = errors.New("models: invalid credentials")

    // Add a new ErrDuplicateEmail error. We'll use this later if a user
    // tries to signup with an email address that's already in use.
    ErrDuplicateEmail = errors.New("models: duplicate email")
)
```

Then create a new file at `internal/models/users.go`:

```
$ touch internal/models/users.go
```

...and define a new `User` struct (to hold the data for a specific user) and a `UserModel` struct (with some placeholder methods for interacting with our database). Like so:

File: internal/models/users.go

```
package models

import (
    "database/sql"
    "time"
)

// Define a new User struct. Notice how the field names and types align
// with the columns in the database "users" table?
type User struct {
    ID          int
    Name        string
    Email       string
    HashedPassword []byte
    Created     time.Time
}

// Define a new UserModel struct which wraps a database connection pool.
type UserModel struct {
    DB *sql.DB
}

// We'll use the Insert method to add a new record to the "users" table.
func (m *UserModel) Insert(name, email, password string) error {
    return nil
}

// We'll use the Authenticate method to verify whether a user exists with
// the provided email address and password. This will return the relevant
// user ID if they do.
func (m *UserModel) Authenticate(email, password string) (int, error) {
    return 0, nil
}

// We'll use the Exists method to check if a user exists with a specific ID.
func (m *UserModel) Exists(id int) (bool, error) {
    return false, nil
}
```

The final stage is to add a new field to our `application` struct so that we can make this model available to our handlers. Update the `main.go` file as follows:

```
File: cmd/web/main.go
```

```
package main

...

// Add a new users field to the application struct.
type application struct {
    logger      *slog.Logger
    snippets   *models.SnippetModel
    users       *models.UserModel
    templateCache map[string]*template.Template
    formDecoder  *form.Decoder
    sessionManager *scs.SessionManager
}

func main() {

    ...

    // Initialize a models.UserModel instance and add it to the application
    // dependencies.
    app := &application{
        logger:      logger,
        snippets:   &models.SnippetModel{DB: db},
        users:       &models.UserModel{DB: db},
        templateCache: templateCache,
        formDecoder:  formDecoder,
        sessionManager: sessionManager,
    }

    tlsConfig := &tls.Config{
        CurvePreferences: []tls.CurveID{tls.X25519, tls.CurveP256},
    }

    srv := &http.Server{
        Addr:      *addr,
        Handler:   app.routes(),
        ErrorLog:   slog.NewLogLogger(logger.Handler(), slog.LevelError),
        TLSConfig:  tlsConfig,
        IdleTimeout: time.Minute,
        ReadTimeout: 5 * time.Second,
        WriteTimeout: 10 * time.Second,
    }

    logger.Info("starting server", "addr", srv.Addr)

    err = srv.ListenAndServeTLS("./tls/cert.pem", "./tls/key.pem")
    logger.Error(err.Error())
    os.Exit(1)
}

...
```

Make sure that all the files are all saved, then go ahead and try to run the application. At this stage you should find that it compiles correctly without any problems.

User signup and password encryption

Before we can log in any users to our Snippetbox application, we need a way for them to sign up for an account. We'll cover how to do that in this chapter.

Go ahead and create a new `ui/html/pages/signup.tpl` file containing the following markup for the signup form.

```
$ touch ui/html/pages/signup.tpl
```

```
File: ui/html/pages/signup.tpl

{{define "title"}}Signup{{end}}

{{define "main"}}
<form action='/user/signup' method='POST' novalidate>
    <div>
        <label>Name:</label>
        {{with .Form.FieldErrors.name}}
            <label class='error'>{{.}}</label>
        {{end}}
        <input type='text' name='name' value='{{.Form.Name}}'>
    </div>
    <div>
        <label>Email:</label>
        {{with .Form.FieldErrors.email}}
            <label class='error'>{{.}}</label>
        {{end}}
        <input type='email' name='email' value='{{.Form.Email}}'>
    </div>
    <div>
        <label>Password:</label>
        {{with .Form.FieldErrors.password}}
            <label class='error'>{{.}}</label>
        {{end}}
        <input type='password' name='password'>
    </div>
    <div>
        <input type='submit' value='Signup'>
    </div>
</form>
{{end}}
```

Hopefully this should feel familiar so far. For the signup form we're using exactly the [same form structure](#) that we used earlier in the book, with three fields: `name`, `email` and `password` (which use the relevant HTML5 input types).

Important: Notice here that we're not re-displaying the password if the form fails validation. This is because we don't want there to be [any risk](#) of the browser (or other intermediary) caching the plain-text password entered by the user.

Then let's update our `cmd/web/handlers.go` file to include a new `userSignupForm` struct (which will represent and hold the form data), and hook it up to the `userSignup` handler.

Like so:

```
File: cmd/web/handlers.go

package main

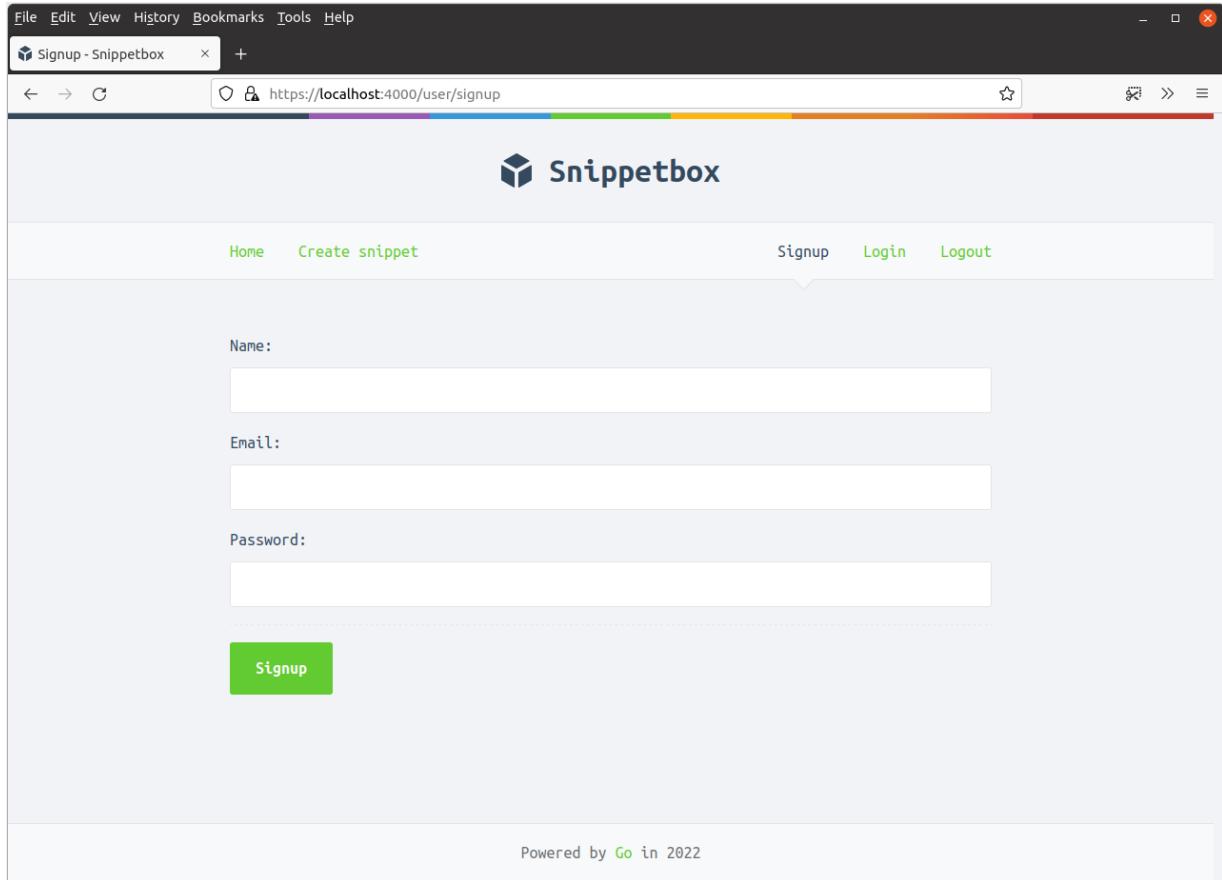
...

// Create a new userSignupForm struct.
type userSignupForm struct {
    Name          string `form:"name"`
    Email         string `form:"email"`
    Password      string `form:"password"`
    validator.Validator `form:"-"`
}

// Update the handler so it displays the signup page.
func (app *application) userSignup(w http.ResponseWriter, r *http.Request) {
    data := app.newTemplateData(r)
    data.Form = userSignupForm{}
    app.render(w, r, http.StatusOK, "signup tmpl", data)
}

...
```

If you run the application and visit <https://localhost:4000/user/signup>, you should now see a page which looks like this:



Validating the user input

When this form is submitted the data will end up being posted to the `userSignupPost` handler that we made earlier.

The first task of this handler will be to validate the data to make sure that it is sane and sensible before we insert it into the database. Specifically, we want to do four things:

1. Check that the provided name, email address and password are not blank.
2. Sanity check the format of the email address.
3. Ensure that the password is at least 8 characters long.
4. Make sure that the email address isn't already in use.

We can cover the first three checks by heading back to our `internal/validator/validator.go` file and creating two helper new methods — `MinChars()` and `Matches()` — along with a regular expression for sanity checking an email address.

Like this:

```
File: internal/validator/validator.go
```

```
package validator

import (
    "regexp" // New import
    "slices"
    "strings"
    "unicode/utf8"
)

// Use the regexp.MustCompile() function to parse a regular expression pattern
// for sanity checking the format of an email address. This returns a pointer to
// a 'compiled' regexp.Regexp type, or panics in the event of an error. Parsing
// this pattern once at startup and storing the compiled *regexp.Regexp in a
// variable is more performant than re-parsing the pattern each time we need it.
var EmailRX = regexp.MustCompile(`^[\w\.-]+\@[a-zA-Z0-9](?:[a-zA-Z0-9-]{0,61}[a-zA-Z0-9])?(?:\.\w+){0,2}`)

...

// MinChars() returns true if a value contains at least n characters.
func MinChars(value string, n int) bool {
    return utf8.RuneCountInString(value) >= n
}

// Matches() returns true if a value matches a provided compiled regular
// expression pattern.
func Matches(value string, rx *regexp.Regexp) bool {
    return rx.MatchString(value)
}
```

There are a couple of things about the `EmailRX` regular expression pattern I want to quickly mention:

- The pattern we're using is the one currently recommended by the W3C and Web Hypertext Application Technology Working Group for validating email addresses. For more information about this pattern, see [here](#). If you're reading this book in PDF format or on a narrow device, and can't see the entire line, then here it is broken up into multiple lines:

```
"^[\w\.-]+\@[a-zA-Z0-9](?:[a-zA-Z0-9-]{0,61}[a-zA-Z0-9])?(?:\.\w+){0,2}"
```

In your code, this regexp pattern should all be on a single line with no whitespace. If there's an alternative pattern that you prefer to use for email address sanity checking, then feel free to swap it in instead.

- Because the `EmailRX` regexp pattern is written as an interpreted string literal, we need to *double-escape special characters* in the regexp with `\\\` for it to work correctly (we can't use a raw string literal because the pattern contains a back quote character). If you're not familiar with the difference between string literal forms, then [this section](#) of the Go

spec is worth a read.

But anyway, I'm digressing. Let's get back to the task at hand.

Head over to your `handlers.go` file and add some code to process the form and run the validation checks like so:

```
File: cmd/web/handlers.go

package main

...

func (app *application) userSignupPost(w http.ResponseWriter, r *http.Request) {
    // Declare an zero-valued instance of our userSignupForm struct.
    var form userSignupForm

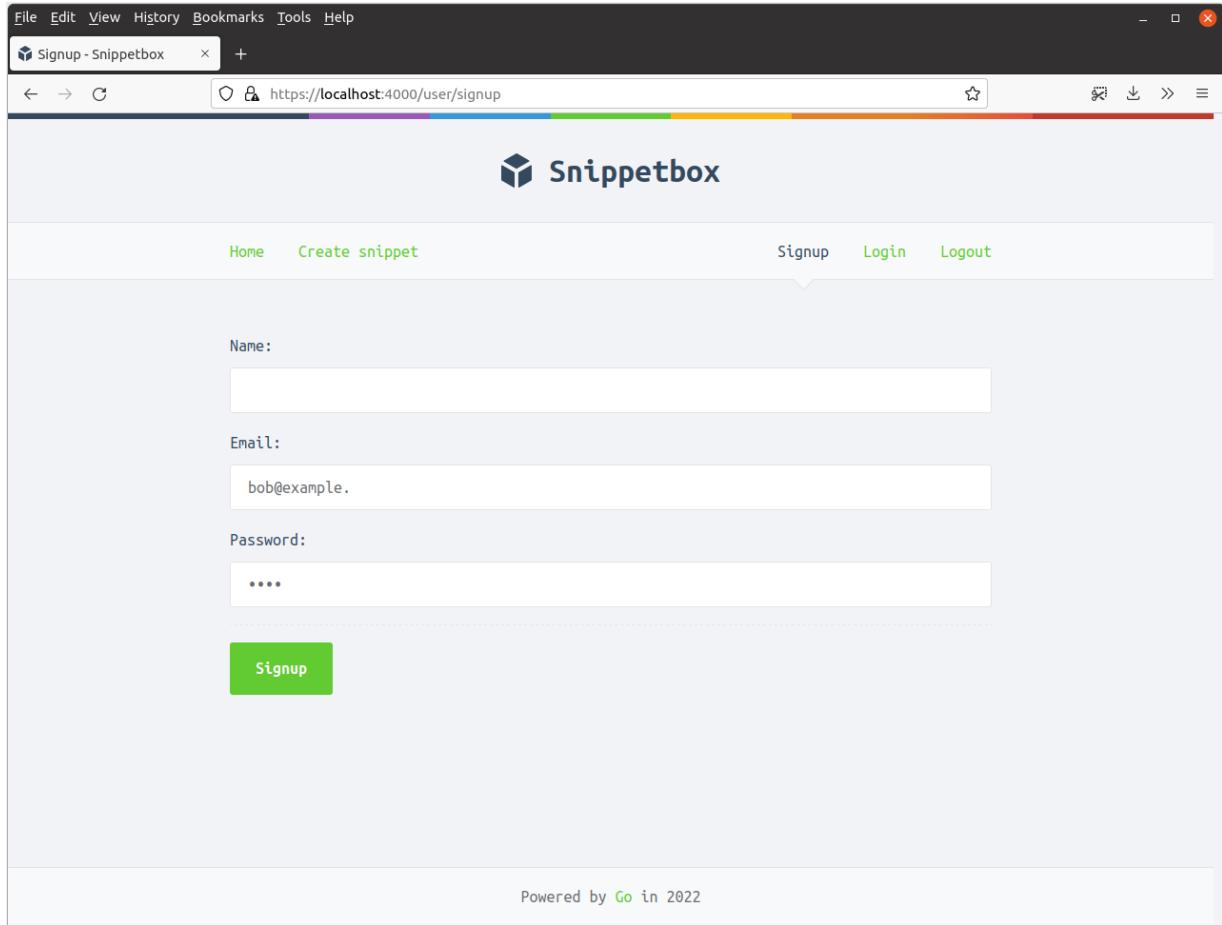
    // Parse the form data into the userSignupForm struct.
    err := app.decodePostForm(r, &form)
    if err != nil {
        app.clientError(w, http.StatusBadRequest)
        return
    }

    // Validate the form contents using our helper functions.
    form.CheckFieldvalidator.NotBlank(form.Name), "name", "This field cannot be blank")
    form.CheckFieldvalidator.NotBlank(form.Email), "email", "This field cannot be blank")
    form.CheckFieldvalidator.Matches(form.Email, validator.EmailRX), "email", "This field must be a valid email address")
    form.CheckFieldvalidator.NotBlank(form.Password), "password", "This field cannot be blank")
    form.CheckFieldvalidator.MinChars(form.Password, 8), "password", "This field must be at least 8 characters long")

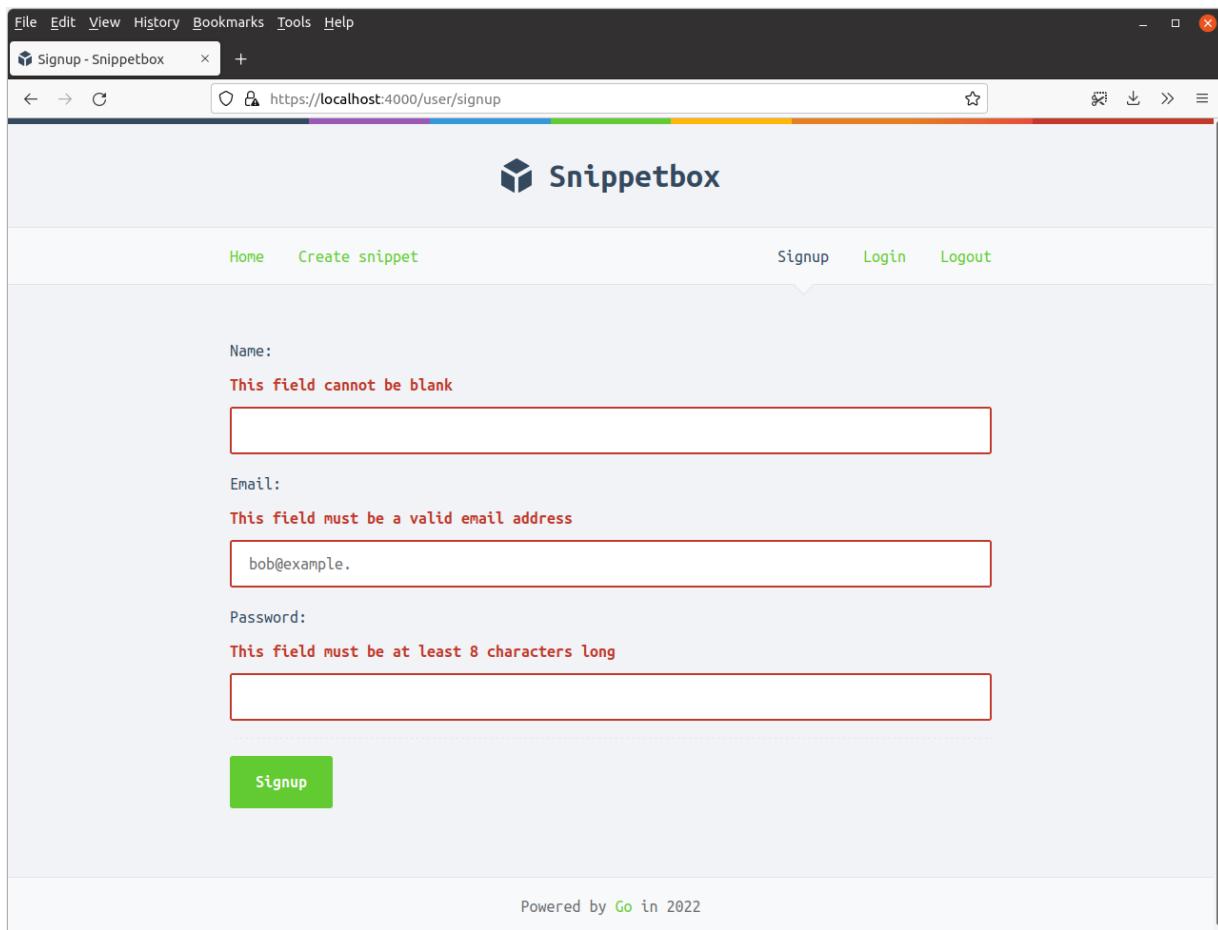
    // If there are any errors, redisplay the signup form along with a 422
    // status code.
    if !form.Valid() {
        data := app.newTemplateData(r)
        data.Form = form
        app.render(w, r, http.StatusUnprocessableEntity, "signup tmpl", data)
        return
    }

    // Otherwise send the placeholder response (for now!).
    fmt.Fprintln(w, "Create a new user...")
}
```

Try running the application now and putting some invalid data into the signup form, like this:



And if you try to submit it, you should see the appropriate validation failures returned like so:



All that remains now is the fourth validation check: *make sure that the email address isn't already in use*. This is a bit trickier to deal with.

Because we've got a `UNIQUE` constraint on the `email` field of our `users` table, it's already guaranteed that we won't end up with two users in our database who have the same email address. So from a business logic and data integrity point of view we are already OK. But the question remains about how we communicate any `email already in use` problem to a user. We'll tackle this at the end of the chapter.

A brief introduction to bcrypt

If your database is ever compromised by an attacker, it's hugely important that it doesn't contain the plain-text versions of your users' passwords.

It's good practice — well, essential, really — to store a one-way hash of the password, derived with a computationally expensive key-derivation function such as Argon2, scrypt or bcrypt. Go has implementations of all 3 algorithms in the golang.org/x/crypto package.

However a plus-point of the bcrypt implementation specifically is that it includes helper

functions specifically designed for hashing and checking passwords, and that's what we'll use here.

If you're following along, please go ahead and download the latest version of the golang.org/x/crypto/bcrypt package:

```
$ go get golang.org/x/crypto/bcrypt@latest  
go: downloading golang.org/x/crypto v0.21.0  
go get: added golang.org/x/crypto v0.21.0
```

There are two functions that we'll use in this book. The first is the `bcrypt.GenerateFromPassword()` function which lets us create a hash of a given plain-text password like so:

```
hash, err := bcrypt.GenerateFromPassword([]byte("my plain text password"), 12)
```

This function will return a 60-character long hash which looks a bit like this:

```
$2a$12$NuTjWXm3KKntReFwyBVHyuf/to.HEwTy.eS206TNfkGfr6HzGJSWG
```

The second parameter that we pass to `bcrypt.GenerateFromPassword()` indicates the cost, which is represented by an integer between 4 and 31. The example above uses a cost of 12, which means that 4096 (2^{12}) bcrypt iterations will be used to generate the password hash.

The higher the cost, the more expensive the hash will be for an attacker to crack (which is a good thing). But a higher cost also means that our application needs to do more work to create the password hash when a user signs up — and that means increased resource use by your application and additional latency for the end user. So choosing an appropriate cost value is a balancing act. A cost of 12 is a reasonable minimum, but if possible you should carry out load testing, and if you can set the cost higher without adversely affecting user experience then you should.

On the flip side, we can check that a plain-text password matches a particular hash using the `bcrypt.CompareHashAndPassword()` function like so:

```
hash := []byte("$2a$12$NuTjWXm3KKntReFwyBVHyuf/to.HEwTy.eS206TNfkGfr6HzGJSWG")  
err := bcrypt.CompareHashAndPassword(hash, []byte("my plain text password"))
```

The `bcrypt.CompareHashAndPassword()` function will return `nil` if the plain-text password matches a particular hash, or an error if they don't match.

Storing the user details

The next stage of our build is to update the `UserModel.Insert()` method so that it creates a new record in our `users` table containing the validated name, email and hashed password.

This will be interesting for two reasons: first we want to store the bcrypt hash of the password (not the password itself) and second, we also need to manage the potential error caused by a duplicate email violating the `UNIQUE` constraint that we added to the table.

All errors returned by MySQL have a particular code, which we can use to triage what has caused the error (a full list of the MySQL error codes and descriptions can be [found here](#)). In the case of a duplicate email, the error code used will be `1062 (ER_DUP_ENTRY)`.

Open the `internal/models/users.go` file and update it to include the following code:

File: internal/models/users.go

```
package models

import (
    "database/sql"
    "errors" // New import
    "strings" // New import
    "time"

    "github.com/go-sql-driver/mysql" // New import
    "golang.org/x/crypto/bcrypt" // New import
)

...

type UserModel struct {
    DB *sql.DB
}

func (m *UserModel) Insert(name, email, password string) error {
    // Create a bcrypt hash of the plain-text password.
    hashedPassword, err := bcrypt.GenerateFromPassword([]byte(password), 12)
    if err != nil {
        return err
    }

    stmt := `INSERT INTO users (name, email, hashed_password, created)
VALUES(?, ?, ?, UTC_TIMESTAMP())`

    // Use the Exec() method to insert the user details and hashed password
    // into the users table.
    _, err = m.DB.Exec(stmt, name, email, string(hashedPassword))
    if err != nil {
        // If this returns an error, we use the errors.As() function to check
        // whether the error has the type *mysql.MySQLError. If it does, the
        // error will be assigned to the mySQLError variable. We can then check
        // whether or not the error relates to our users_uc_email key by
        // checking if the error code equals 1062 and the contents of the error
        // message string. If it does, we return an ErrDuplicateEmail error.
        var mySQLError *mysql.MySQLError
        if errors.As(err, &mySQLError) {
            if mySQLError.Number == 1062 && strings.Contains(mySQLError.Message, "users_uc_email") {
                return ErrDuplicateEmail
            }
        }
        return err
    }

    return nil
}

...
```

We can then finish this all off by updating the `userSignup` handler like so:

File: cmd/web/handlers.go

```
package main

...

func (app *application) userSignupPost(w http.ResponseWriter, r *http.Request) {
    var form userSignupForm

    err := app.decodePostForm(r, &form)
    if err != nil {
        app.clientError(w, http.StatusBadRequest)
        return
    }

    form.CheckFieldvalidator.NotBlank(form.Name), "name", "This field cannot be blank")
    form.CheckFieldvalidator.NotBlank(form.Email), "email", "This field cannot be blank")
    form.CheckFieldvalidator.Matches(form.Email, validator.EmailRX), "email", "This field must be a valid email address")
    form.CheckFieldvalidator.NotBlank(form.Password), "password", "This field cannot be blank")
    form.CheckFieldvalidator.MinChars(form.Password, 8), "password", "This field must be at least 8 characters long")

    if !form.Valid() {
        data := app.newTemplateData(r)
        data.Form = form
        app.render(w, r, http.StatusUnprocessableEntity, "signup tmpl", data)
        return
    }

    // Try to create a new user record in the database. If the email already
    // exists then add an error message to the form and re-display it.
    err = app.users.Insert(form.Name, form.Email, form.Password)
    if err != nil {
        if errors.Is(err, models.ErrDuplicateEmail) {
            form.AddFieldError("email", "Email address is already in use")

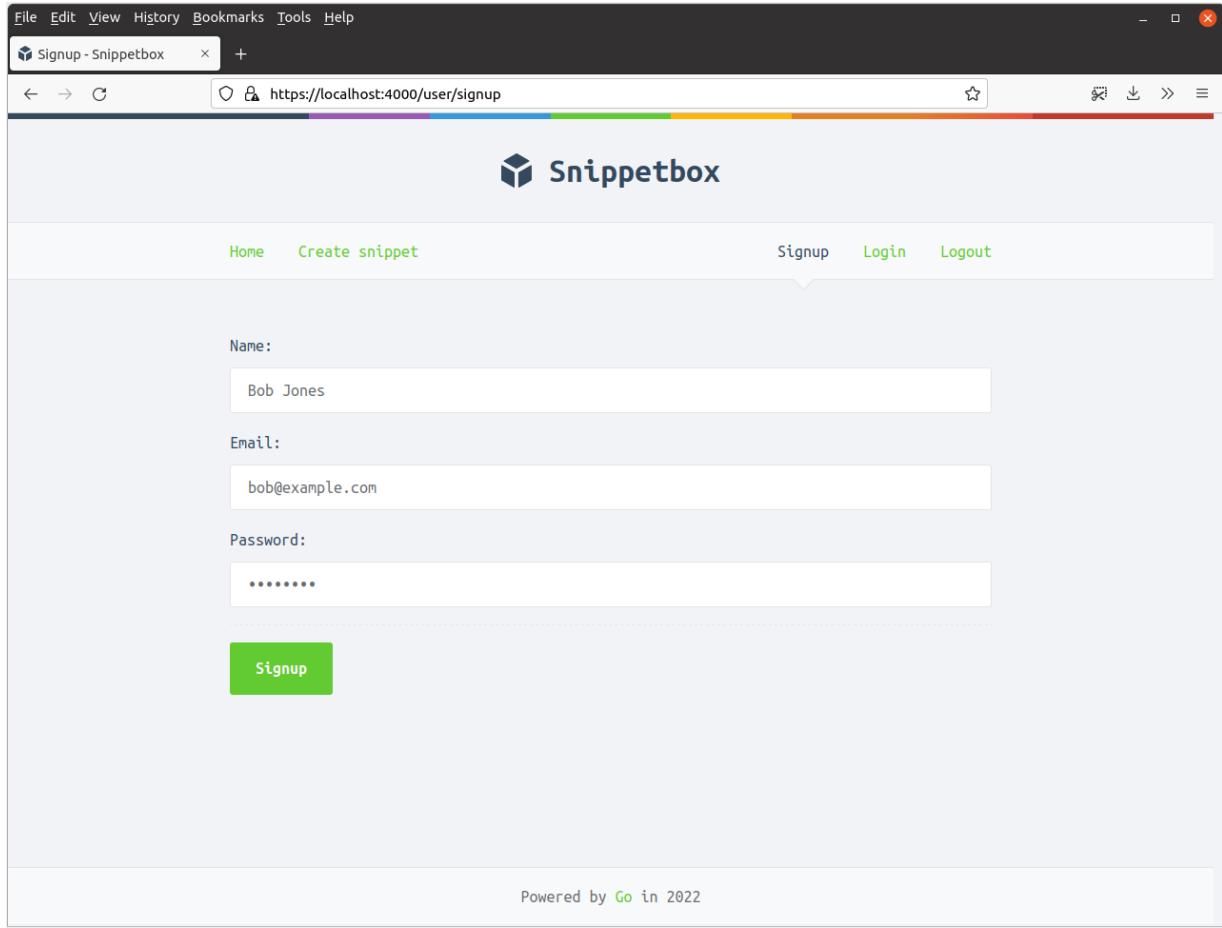
            data := app.newTemplateData(r)
            data.Form = form
            app.render(w, r, http.StatusUnprocessableEntity, "signup tmpl", data)
        } else {
            app.serverError(w, r, err)
        }
    }

    return
}

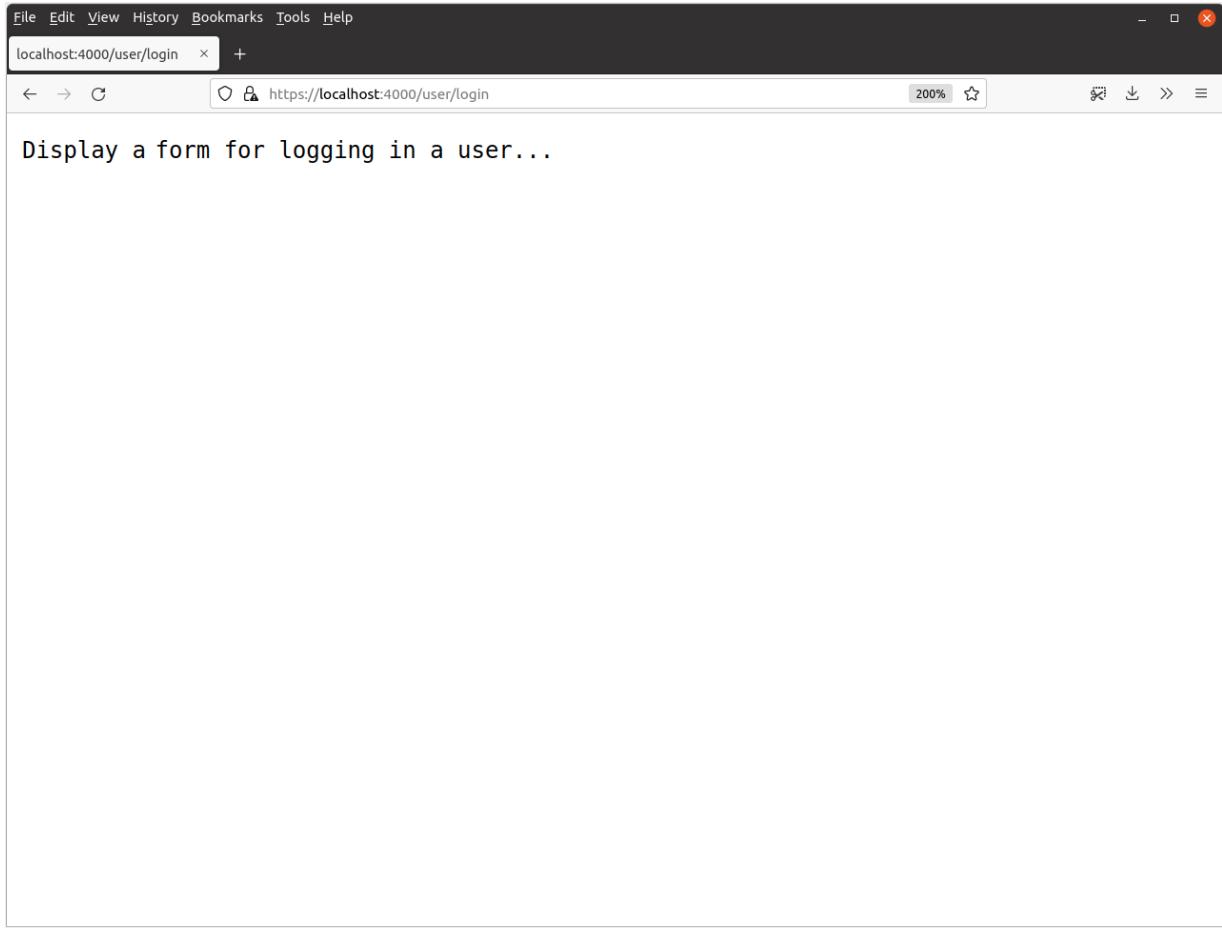
// Otherwise add a confirmation flash message to the session confirming that
// their signup worked.
app.sessionManager.Put(r.Context(), "flash", "Your signup was successful. Please log in.")

// And redirect the user to the login page.
http.Redirect(w, r, "/user/login", http.StatusSeeOther)
}
```

Save the files, restart the application and try signing up for an account. Make sure to remember the email address and password that you use... you'll need them in the next chapter!



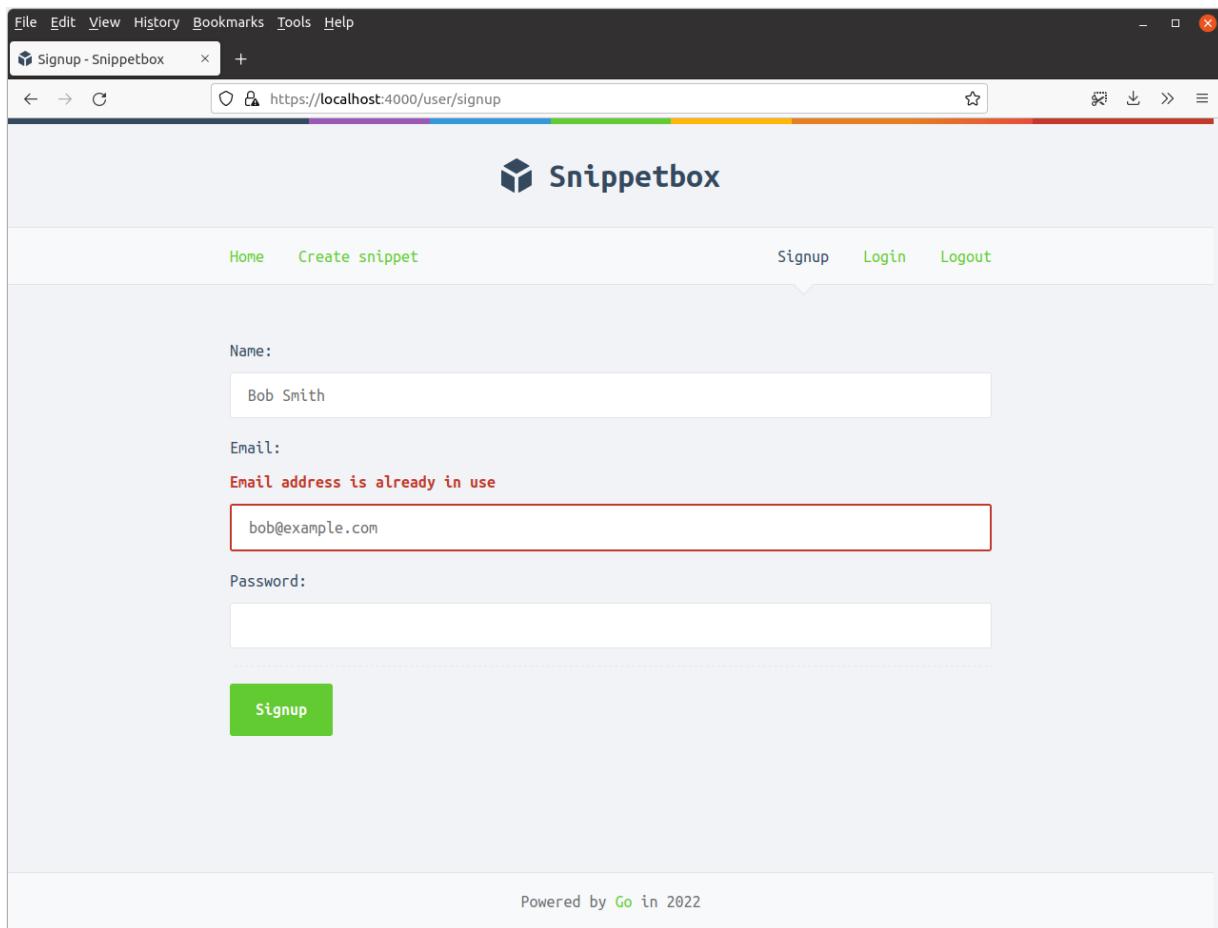
If everything works correctly, you should find that your browser redirects you to <https://localhost:4000/user/login> after you submit the form.



At this point it's worth opening your MySQL database and looking at the contents of the `users` table. You should see a new record with the details you just used to sign up and a bcrypt hash of the password.

```
mysql> SELECT * FROM users;
+----+-----+-----+-----+
| id | name      | email           | hashed_password          | created          |
+----+-----+-----+-----+
| 1  | Bob Jones | bob@example.com | $2a$12$NXQr0wVWp/TqAzCCyDoyegtpV40ExwrzVLnbFphPpWdvnmIoZ.Q. | 2024-03-18 11:29:23 |
+----+-----+-----+-----+
1 row in set (0.01 sec)
```

If you like, try heading back to the signup form and adding another account with the same email address. You should get a validation failure like so:



Additional information

Using database bcrypt implementations

Some databases provide built-in functions that you can use for password hashing and verification instead of implementing your own in Go, like we have in the code above.

But it's probably a good idea to avoid using these for two reasons:

- They tend to be vulnerable to [side-channel timing attacks](#) due to string comparison time not being constant, at least in [PostgreSQL](#) and [MySQL](#).
- Unless you're very careful, sending a plain-text password to your database risks the password being accidentally recorded in one of your database logs. A couple of high-profile examples of passwords being accidentally recorded in logs were the [GitHub](#) and [Twitter](#) incidents in 2018.

Alternatives for checking email duplicates

I understand that the code in our `UserModel.Insert()` method isn't very pretty, and that checking the error returned by MySQL feels a bit flaky. What if future versions of MySQL change their error numbers? Or the format of their error messages?

An alternative (but also imperfect) option would be to add an `UserModel.EmailTaken()` method to our model which checks to see if a user with a specific email already exists. We could call this *before* we try to insert a new record, and add a validation error message to the form as appropriate.

However, this would introduce a race condition to our application. If two users try to sign up with the same email address at *exactly* the same time, both submissions will pass the validation check but ultimately only one `INSERT` into the MySQL database will succeed. The other will violate our `UNIQUE` constraint and the user would end up receiving a `500 Internal Server Error` response.

The outcome of this particular race condition is fairly benign, and [some people](#) would advise you to simply not worry about it. But thinking critically about your application logic and writing code which avoids race conditions is a good habit to get into, and where there's a viable alternative — like there is in this case — it's better to avoid shipping with known race conditions in your codebase.

User login

In this chapter we're going to focus on creating the user login page for our application.

Before we get into the main part of this work, let's quickly revisit the `internal/validator` package that we made earlier and update it to support validation errors *which aren't associated with one specific form field*.

We'll use this later in the chapter to show the user a generic "*your email address or password is wrong*" message if their login fails, as this is considered **more secure** than explicitly indicating why the login failed.

Please go ahead and update your `internal/validator/validator.go` file like so:

```
File: internal/validator/validator.go

package validator

...

// Add a new NonFieldErrors []string field to the struct, which we will use to
// hold any validation errors which are not related to a specific form field.
type Validator struct {
    NonFieldErrors []string
    FieldErrors    map[string]string
}

// Update the Valid() method to also check that the NonFieldErrors slice is
// empty.
func (v *Validator) Valid() bool {
    return len(v.FieldErrors) == 0 && len(v.NonFieldErrors) == 0
}

// Create an AddNonFieldError() helper for adding error messages to the new
// NonFieldErrors slice.
func (v *Validator) AddNonFieldError(message string) {
    v.NonFieldErrors = append(v.NonFieldErrors, message)
}

...
```

Next let's create a new `ui/html/pages/login tmpl` template containing the markup for our login page. We'll follow the same pattern for showing validation errors and re-displaying data that we used for our signup page.

```
$ touch ui/html/pages/login.tmpl
```

```

File: ui/html/pages/login.tpl

{{define "title"}}Login{{end}}

{{define "main"}}
<form action='/user/login' method='POST' novalidate>
    <!-- Notice that here we are looping over the NonFieldErrors and displaying
        them, if any exist -->
    {{range .Form.NonFieldErrors}}
        <div class='error'>{{.}}</div>
    {{end}}
    <div>
        <label>Email:</label>
        {{with .Form.FieldErrors.email}}
            <label class='error'>{{.}}</label>
        {{end}}
        <input type='email' name='email' value='{{.Form.Email}}'>
    </div>
    <div>
        <label>Password:</label>
        {{with .Form.FieldErrors.password}}
            <label class='error'>{{.}}</label>
        {{end}}
        <input type='password' name='password'>
    </div>
    <div>
        <input type='submit' value='Login'>
    </div>
</form>
{{end}}

```

Then let's head to our `cmd/web/handlers.go` file and create a new `userLoginForm` struct (to represent and hold the form data), and adapt our `userLogin` handler to render the login page.

Like so:

```

File: cmd/web/handlers.go

package main

...

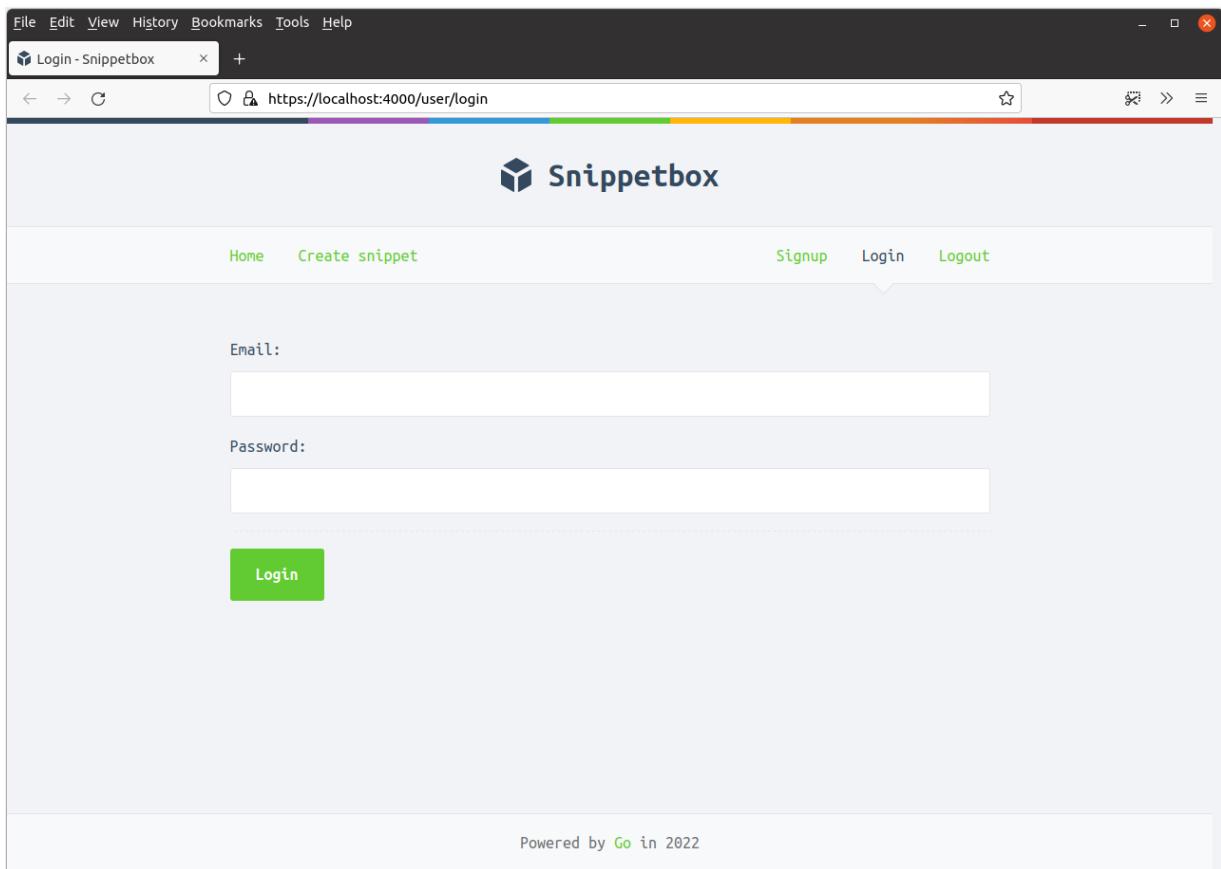
// Create a new userLoginForm struct.
type userLoginForm struct {
    Email          string `form:"email"`
    Password       string `form:"password"`
    validator.Validator `form:"-"`
}

// Update the handler so it displays the login page.
func (app *application) userLogin(w http.ResponseWriter, r *http.Request) {
    data := app.newTemplateData(r)
    data.Form = userLoginForm{}
    app.render(w, r, http.StatusOK, "login.tpl", data)
}

...

```

If you run the application and visit <https://localhost:4000/user/login>, you should now see the login page looking like this:



Verifying the user details

The next step is the interesting part: *how do we verify that the email and password submitted by a user are correct?*

The core part of this verification logic will take place in the `UserModel.Authenticate()` method of our user model. Specifically, we'll need it to do two things:

1. First it should retrieve the hashed password associated with the email address from our MySQL `users` table. If the email doesn't exist in the database, we will return the `ErrInvalidCredentials` error that we made earlier.
2. Otherwise, we want to compare the hashed password from the `users` table with the plain-text password that the user provided when logging in. If they don't match, we want to return the `ErrInvalidCredentials` error again. But if they do match, we want to return the user's `id` value from the database.

Let's do exactly that. Go ahead and add the following code to your

internal/models/users.go file:

```
File: internal/models/users.go

package models

...

func (m *UserModel) Authenticate(email, password string) (int, error) {
    // Retrieve the id and hashed password associated with the given email. If
    // no matching email exists we return the ErrInvalidCredentials error.
    var id int
    var hashedPassword []byte

    stmt := "SELECT id, hashed_password FROM users WHERE email = ?"

    err := m.DB.QueryRow(stmt, email).Scan(&id, &hashedPassword)
    if err != nil {
        if errors.Is(err, sql.ErrNoRows) {
            return 0, ErrInvalidCredentials
        } else {
            return 0, err
        }
    }

    // Check whether the hashed password and plain-text password provided match.
    // If they don't, we return the ErrInvalidCredentials error.
    err = bcrypt.CompareHashAndPassword(hashedPassword, []byte(password))
    if err != nil {
        if errors.Is(err, bcrypt.ErrMismatchedHashAndPassword) {
            return 0, ErrInvalidCredentials
        } else {
            return 0, err
        }
    }

    // Otherwise, the password is correct. Return the user ID.
    return id, nil
}
```

Our next step involves updating the `userLoginPost` handler so that it parses the submitted login form data and calls this `UserModel.Authenticate()` method.

If the login details are valid, we then want to add the user's `id` to their session data so that — for future requests — we know that they have authenticated successfully and which user they are.

Head over to your `handlers.go` file and update it as follows:

```
File: cmd/web/handlers.go

package main

...

func (app *application) userLoginPost(w http.ResponseWriter, r *http.Request) {
    // Decode the form data into the userLoginForm struct
```

```

// Decode the form data into the userLoginForm struct.
var form userLoginForm

err := app.decodePostForm(r, &form)
if err != nil {
    app.clientError(w, http.StatusBadRequest)
    return
}

// Do some validation checks on the form. We check that both email and
// password are provided, and also check the format of the email address as
// a UX-nicety (in case the user makes a typo).
form.CheckFieldvalidator.NotBlank(form.Email), "email", "This field cannot be blank")
form.CheckFieldvalidator.Matches(form.Email, validator.EmailRX), "email", "This field must be a valid email address")
form.CheckFieldvalidator.NotBlank(form.Password), "password", "This field cannot be blank")

if !form.Valid() {
    data := app.newTemplateData(r)
    data.Form = form
    app.render(w, r, http.StatusUnprocessableEntity, "login tmpl", data)
    return
}

// Check whether the credentials are valid. If they're not, add a generic
// non-field error message and re-display the login page.
id, err := app.users.Authenticate(form.Email, form.Password)
if err != nil {
    if errors.Is(err, models.ErrInvalidCredentials) {
        form.AddNonFieldError("Email or password is incorrect")

        data := app.newTemplateData(r)
        data.Form = form
        app.render(w, r, http.StatusUnprocessableEntity, "login tmpl", data)
    } else {
        app.serverError(w, r, err)
    }
    return
}

// Use the RenewToken() method on the current session to change the session
// ID. It's good practice to generate a new session ID when the
// authentication state or privilege levels changes for the user (e.g. login
// and logout operations).
err = app.sessionManager.RenewToken(r.Context())
if err != nil {
    app.serverError(w, r, err)
    return
}

// Add the ID of the current user to the session, so that they are now
// 'logged in'.
app.sessionManager.Put(r.Context(), "authenticatedUserID", id)

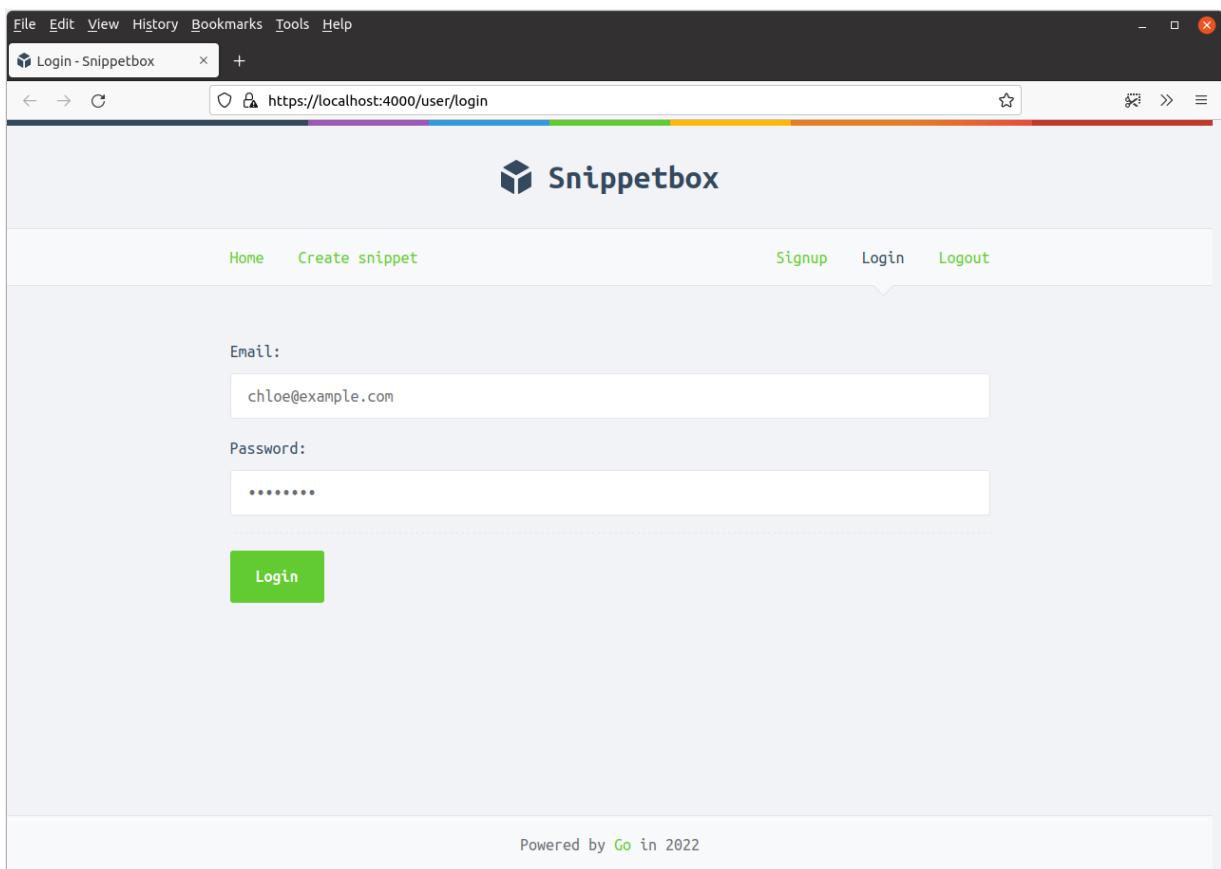
// Redirect the user to the create snippet page.
http.Redirect(w, r, "/snippet/create", http.StatusSeeOther)
}
...

```

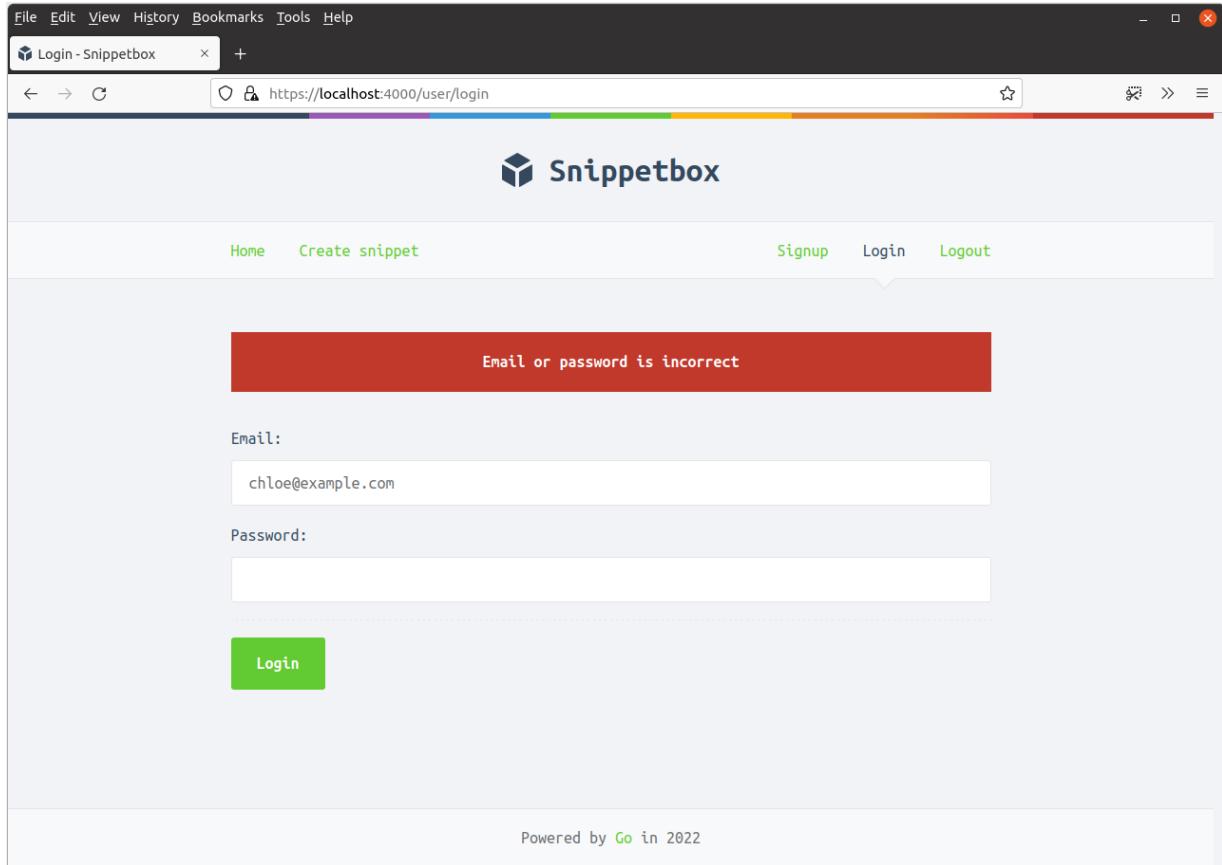
Note: The `SessionManager.RenewToken()` method that we're using in the code above will change the ID of the current user's session *but retain any data associated with the session*. It's good practice to do this before login to mitigate the risk of a session fixation attack. For more background and information on this, please see the [OWASP Session Management Cheat Sheet](#).

Alright, let's give this a try!

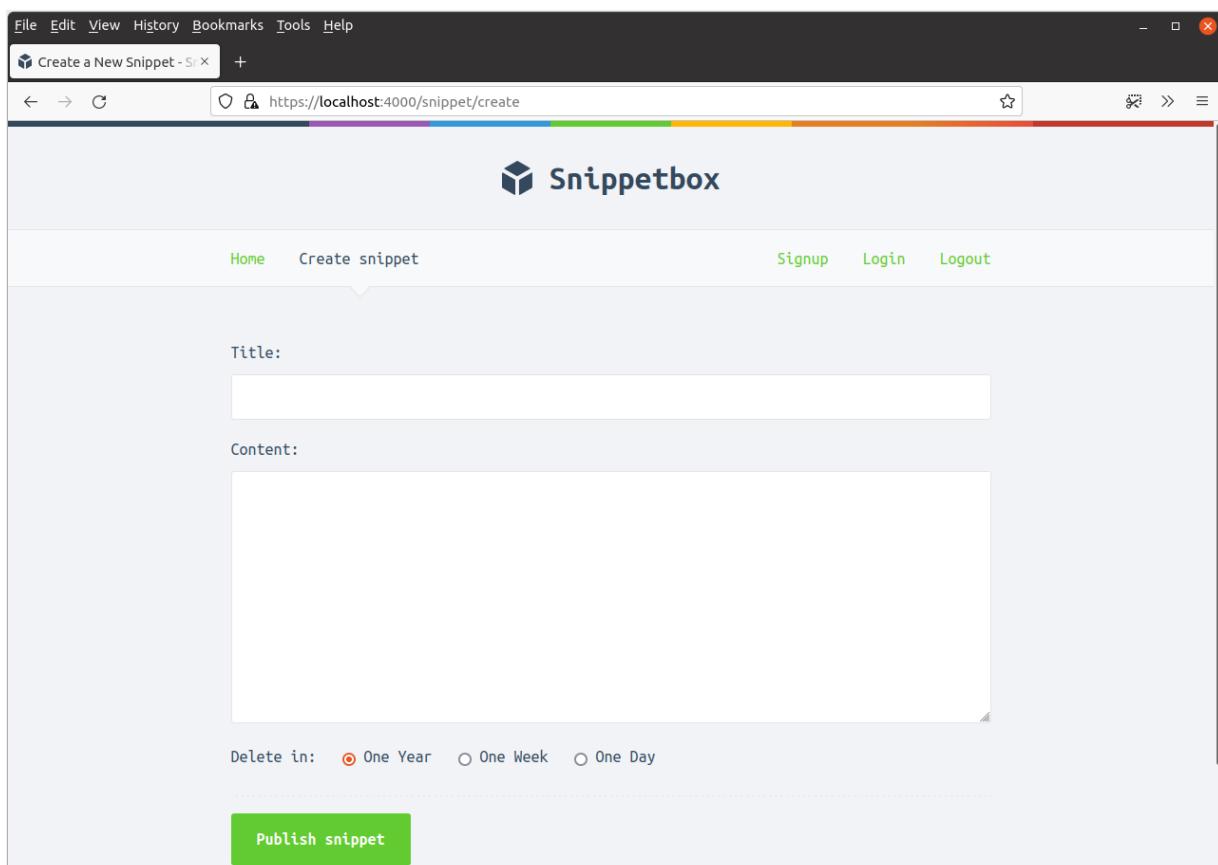
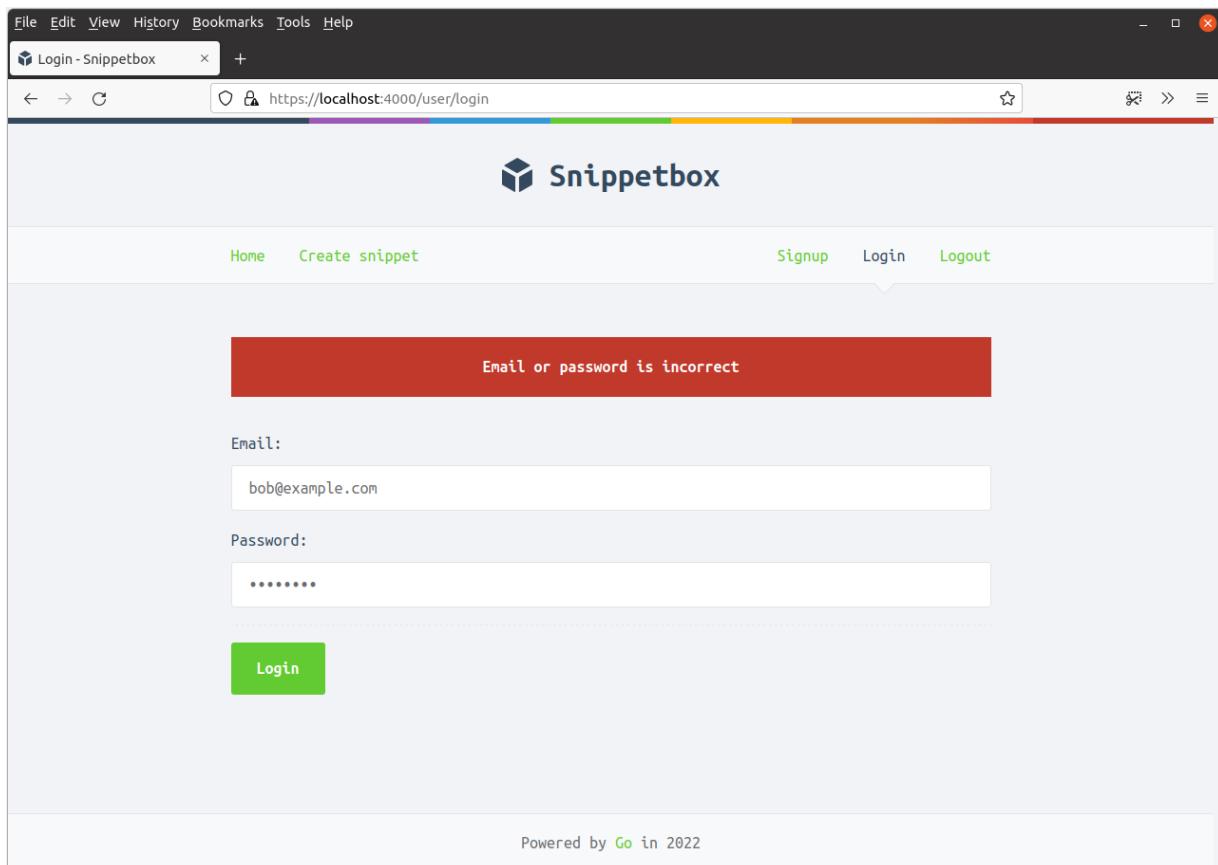
Restart the application and try submitting some invalid user credentials...



You should get a non-field validation error message which looks like this:



But when you input some correct credentials (use the email address and password for the user that you created in the previous chapter), the application should log you in and redirect you to the create snippet page, like so:



We've covered a lot of ground in the last two chapters, so let's quickly take stock of where

things are at.

- Users can now *register* with the site using the `GET /user/signup` form. We store the details of registered users (including a hashed version of their password) in the `users` table of our database.
- Registered users can then *authenticate* by using the `GET /user/login` form to provide their email address and password. If these match the details of a registered user, we deem them to have authenticated successfully and add the relevant `"authenticatedUserID"` value to their session data.

User logout

This brings us nicely to logging out a user. Implementing the user logout is straightforward in comparison to the signup and login — essentially all we need to do is remove the `"authenticatedUserID"` value from their session.

At the same time it's good practice to renew the session ID again, and we'll also add a flash message to the session data to confirm to the user that they've been logged out.

Let's update the `userLogoutPost` handler to do exactly that.

```
File: cmd/web/handlers.go

package main

...

func (app *application) userLogoutPost(w http.ResponseWriter, r *http.Request) {
    // Use the RenewToken() method on the current session to change the session
    // ID again.
    err := app.sessionManager.RenewToken(r.Context())
    if err != nil {
        app.serverError(w, r, err)
        return
    }

    // Remove the authenticatedUserID from the session data so that the user is
    // 'logged out'.
    app.sessionManager.Remove(r.Context(), "authenticatedUserID")

    // Add a flash message to the session to confirm to the user that they've been
    // logged out.
    app.sessionManager.Put(r.Context(), "flash", "You've been logged out successfully!")

    // Redirect the user to the application home page.
    http.Redirect(w, r, "/", http.StatusSeeOther)
}
```

Save the file and restart the application. If you now click the 'Logout' link in the navigation bar you should be logged out and redirected to the homepage like so:

File Edit View History Bookmarks Tools Help

Home - Snippetbox × +

← → ⌘ ⌘ https://localhost:4000 ☆ ⌘ ⌘ ⌘ ⌘ ⌘ ⌘

Snippetbox

Home Create snippet Signup Login Logout

You've been logged out successfully!

Latest Snippets

Title	Created	ID
The wren	05 Apr 2022 at 16:21	#6
From time to time	05 Apr 2022 at 14:59	#5
O snail	05 Apr 2022 at 07:32	#4
First autumn morning	05 Apr 2022 at 07:20	#3
Over the wintry forest	05 Apr 2022 at 07:20	#2
An old silent pond	05 Apr 2022 at 07:20	#1

User authorization

Being able to authenticate the users of our application is all well and good, but now we need to do something useful with that information. In this chapter we'll introduce some *authorization* checks so that:

1. Only authenticated (i.e. logged in) users can create a new snippet; and
2. The contents of the navigation bar changes depending on whether a user is authenticated (logged in) or not. Specifically:
 - Authenticated users should see links to 'Home', 'Create snippet' and 'Logout'.
 - Unauthenticated users should see links to 'Home', 'Signup' and 'Login'.

As I mentioned briefly in the previous chapter, we can check whether a request is being made by an authenticated user or not by checking for the existence of an "`authenticatedUserID`" value in their session data.

So let's start with that. Open the `cmd/web/helpers.go` file and add an `isAuthenticated()` helper function to return the authentication status like so:

```
File: cmd/web/helpers.go

package main

...

// Return true if the current request is from an authenticated user, otherwise
// return false.
func (app *application) isAuthenticated(r *http.Request) bool {
    return app.sessionManager.Exists(r.Context(), "authenticatedUserID")
}
```

That's neat. We can now check whether or not the request is coming from an authenticated (logged in) user by simply calling this `isAuthenticated()` helper.

The next step is to find a way to pass this information to our HTML templates, so that we can toggle the contents of the navigation bar appropriately.

There are two parts to this. First, we'll need to add a new `IsAuthenticated` field to our `templateData` struct:

```

File: cmd/web/templates.go

package main

import (
    "html/template"
    "path/filepath"
    "time"

    "snippetbox.alexedwards.net/internal/models"
)

type templateData struct {
    CurrentYear      int
    Snippet          models.Snippet
    Snippets         []models.Snippet
    Form             any
    Flash            string
    IsAuthenticated bool // Add an IsAuthenticated field to the templateData struct.
}

...

```

And the second step is to update our `newTemplateData()` helper so that this information is automatically added to the `templateData` struct every time we render a template. Like so:

```

File: cmd/web/helpers.go

package main

...

func (app *application) newTemplateData(r *http.Request) templateData {
    return templateData{
        CurrentYear:     time.Now().Year(),
        Flash:          app.sessionManager.PopString(r.Context(), "flash"),
        // Add the authentication status to the template data.
        IsAuthenticated: app.isAuthenticated(r),
    }
}

...

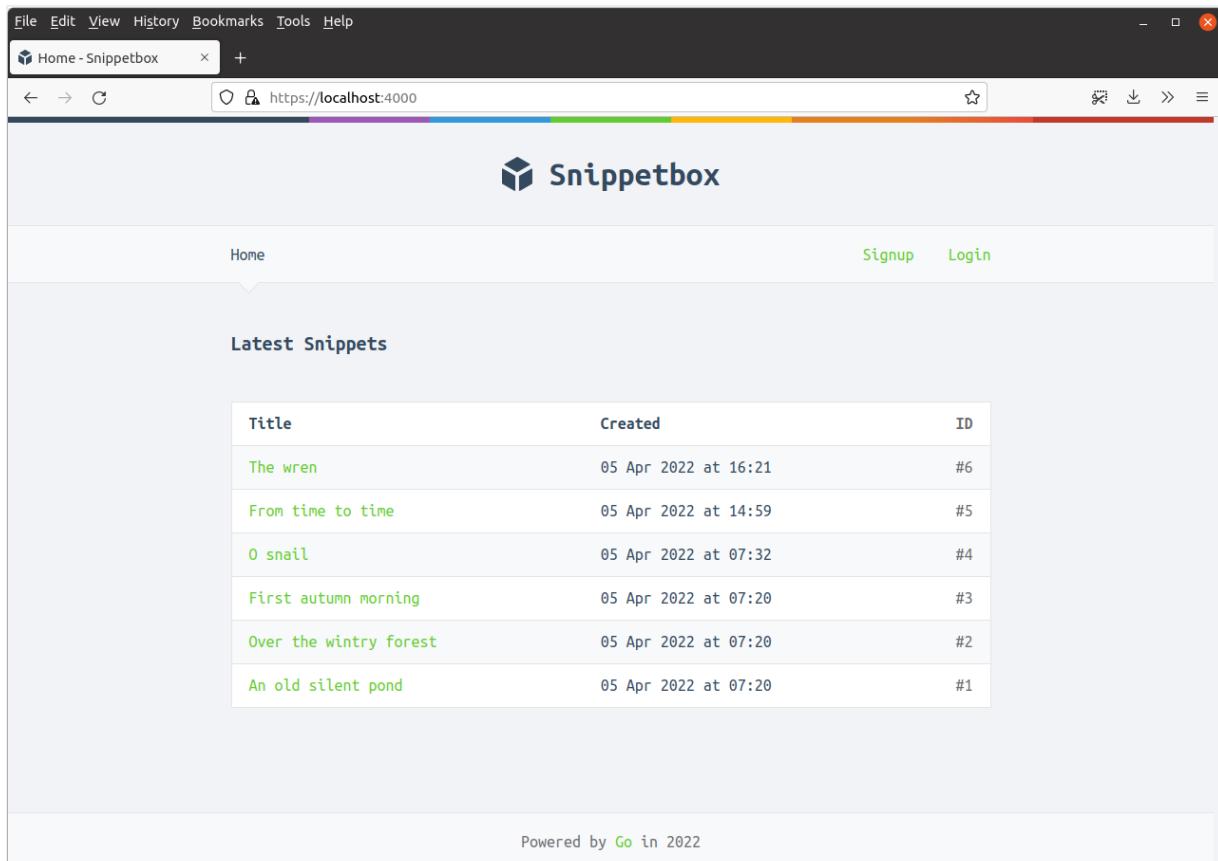
```

Once that's done, we can update the `ui/html/partials/nav.tpl` file to toggle the navigation links using the `{{if .IsAuthenticated}}` action like so:

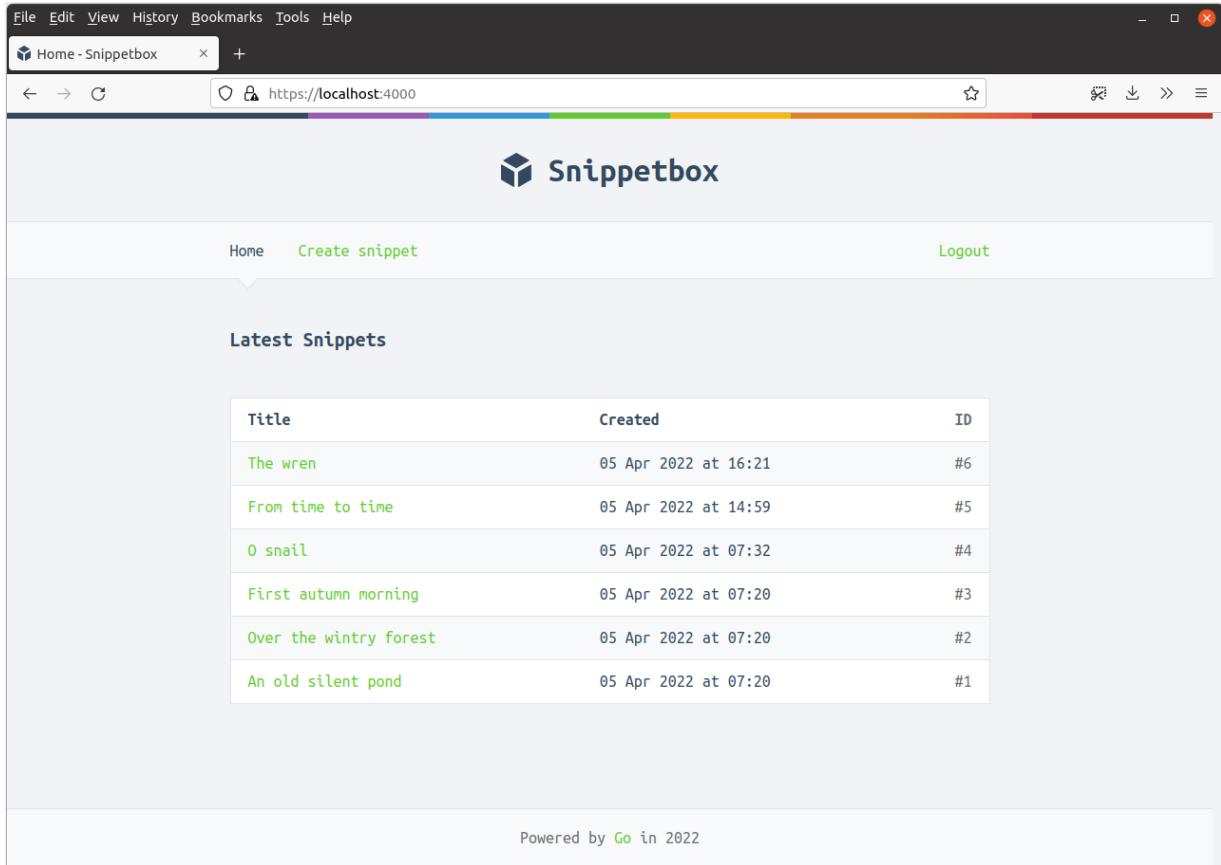
```
File: ui/html/partials/nav.tpl
```

```
 {{define "nav"}}
<nav>
  <div>
    <a href='/'>Home</a>
    <!-- Toggle the link based on authentication status -->
    {{if .IsAuthenticated}}
      <a href='/snippet/create'>Create snippet</a>
    {{end}}
  </div>
  <div>
    <!-- Toggle the links based on authentication status -->
    {{if .IsAuthenticated}}
      <form action='/user/logout' method='POST'>
        <button>Logout</button>
      </form>
    {{else}}
      <a href='/user/signup'>Signup</a>
      <a href='/user/login'>Login</a>
    {{end}}
  </div>
</nav>
{{end}}
```

Save all the files and try running the application now. If you're not currently logged in, your application homepage should look like this:



Otherwise — if you are logged in — your homepage should look like this:



Feel free to have a play around with this, and try logging in and out until you're confident that the navigation bar is being changed as you would expect.

Restricting access

As it stands, we're hiding the 'Create snippet' navigation link for any user that isn't logged in. But an unauthenticated user could still create a new snippet by visiting the <https://localhost:4000/snippet/create> page directly.

Let's fix that, so that if an unauthenticated user tries to visit any routes with the URL path `/snippet/create` they are redirected to `/user/login` instead.

The simplest way to do this is via some middleware. Open the `cmd/web/middleware.go` file and create a new `requireAuthentication()` middleware function, following the same pattern that we used [earlier in the book](#):

```
File: cmd/web/middleware.go
```

```
package main

...

func (app *application) requireAuthentication(next http.Handler) http.Handler {
    return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
        // If the user is not authenticated, redirect them to the login page and
        // return from the middleware chain so that no subsequent handlers in
        // the chain are executed.
        if !app.isAuthenticated(r) {
            http.Redirect(w, r, "/user/login", http.StatusSeeOther)
            return
        }

        // Otherwise set the "Cache-Control: no-store" header so that pages
        // requiring authentication are not stored in the users browser cache (or
        // other intermediary cache).
        w.Header().Add("Cache-Control", "no-store")

        // And call the next handler in the chain.
        next.ServeHTTP(w, r)
    })
}
```

We can now add this middleware to our [cmd/web/routes.go](#) file to protect specific routes.

In our case we'll want to protect the `GET /snippet/create` and `POST /snippet/create` routes. And there's not much point logging out a user if they're not logged in, so it makes sense to use it on the `POST /user/logout` route as well.

To help with this, let's rearrange our application routes into two 'groups'.

The first group will contain our 'unprotected' routes and use our existing `dynamic` middleware chain. The second group will contain our 'protected' routes and will use a new `protected` middleware chain — consisting of the `dynamic` middleware chain *plus* our new `requireAuthentication()` middleware.

Like this:

```
File: cmd/web/routes.go
```

```
package main

...

func (app *application) routes() http.Handler {
    mux := http.NewServeMux()

    fileServer := http.FileServer(http.Dir("./ui/static/"))
    mux.Handle("GET /static/", http.StripPrefix("/static", fileServer))

    // Unprotected application routes using the "dynamic" middleware chain.
    dynamic := alice.New(app.sessionManager.LoadAndSave)

    mux.Handle("GET /${}", dynamic.ThenFunc(app.home))
    mux.Handle("GET /snippet/view/{id}", dynamic.ThenFunc(app.snippetView))
    mux.Handle("GET /user/signup", dynamic.ThenFunc(app.userSignup))
    mux.Handle("POST /user/signup", dynamic.ThenFunc(app.userSignupPost))
    mux.Handle("GET /user/login", dynamic.ThenFunc(app.userLogin))
    mux.Handle("POST /user/login", dynamic.ThenFunc(app.userLoginPost))

    // Protected (authenticated-only) application routes, using a new "protected"
    // middleware chain which includes the requireAuthentication middleware.
    protected := dynamic.Append(app.requireAuthentication)

    mux.Handle("GET /snippet/create", protected.ThenFunc(app.snippetCreate))
    mux.Handle("POST /snippet/create", protected.ThenFunc(app.snippetCreatePost))
    mux.Handle("POST /user/logout", protected.ThenFunc(app.userLogoutPost))

    standard := alice.New(app.recoverPanic, app.logRequest, commonHeaders)
    return standard.Then(mux)
}
```

Save the files, restart the application and make sure that you're logged out.

Then try visiting <https://localhost:4000/snippet/create> directly in your browser. You should find that you get immediately redirected to the login form instead.

If you like, you can also confirm with curl that unauthenticated users are redirected for the `POST /snippet/create` route too:

```
$ curl -ki -d "" https://localhost:4000/snippet/create
HTTP/2 303
content-security-policy: default-src 'self'; style-src 'self' fonts.googleapis.com; font-src fonts.gstatic.com
location: /user/login
referrer-policy: origin-when-cross-origin
server: Go
vary: Cookie
x-content-type-options: nosniff
x-frame-options: deny
x-xss-protection: 0
content-length: 0
date: Wed, 18 Mar 2024 11:29:23 GMT
```

Additional information

Without using alice

If you're not using the [justinas/alice](#) package to manage your middleware that's OK — you can manually wrap your handlers like this instead:

```
mux.Handle("POST /snippet/create", app.sessionManager.LoadAndSave(app.requireAuthentication(http.HandlerFunc(app.snippetCreat
```

CSRF protection

In this chapter we'll look at how to protect our application from [cross-site request forgery \(CSRF\)](#) attacks.

If you're not familiar with the principles of CSRF, it's a type of attack where a malicious third-party website sends state-changing HTTP requests to your website. A great explanation of the basic CSRF attack can be [found here](#).

In our application, the main risk is this:

- A user logs into our application. Our session cookie is set to persist for 12 hours, so they will remain logged in even if they navigate away from the application.
- The user then goes to another website, which contains some malicious code that sends a cross-site request to our `POST /snippet/create` endpoint to add a new snippet to our database. The user's session cookie for our application will be sent along with this request.
- Because the request includes the session cookie, our application will interpret the request as coming from a logged-in user and it will process the request with that user's privileges. So completely unknown to the user, a new snippet will be added to our database.

As well as 'traditional' CSRF attacks like the above (where a request is processed with a logged-in user's privileges) your application may also be at risk from [login and logout](#) CSRF attacks.

SameSite cookies

One mitigation that we can take to prevent CSRF attacks is to make sure that the [SameSite](#) attribute is appropriately set on our session cookie.

By default the `alexedwards/scs` package that we're using always sets `SameSite=Lax` on the session cookie. This means that the session cookie won't be sent by the user's browser for any cross-site requests with the HTTP methods `POST`, `PUT` or `DELETE`.

So long as our application uses the `POST` method for any state-changing HTTP requests (like we are for our `login`, `signup`, `logout` and `create snippet` form submissions), it means that the

session cookie won't be sent for these requests if they come from another website — thereby preventing the CSRF attack.

However, the `SameSite` attribute is still relatively new and only fully supported by [96% of browsers](#) worldwide. So, although it's something that we can (and should) use as a defensive measure, we can't rely on it for all users.

Token-based mitigation

To mitigate the risk of CSRF for all users we'll also need to implement some form of [token check](#). Like session management and password hashing, when it comes to this there's a lot that you can get wrong... so it's probably safest to use a tried-and-tested third-party package instead of rolling your own implementation.

The two most popular packages for stopping CSRF attacks in Go web applications are [gorilla/csrf](#) and [justinas/nosurf](#). They both do roughly the same thing, using the [double-submit cookie](#) pattern to prevent attacks. In this pattern a random CSRF token is generated and sent to the user in a CSRF cookie. This CSRF token is then added to a hidden field in each HTML form that is potentially vulnerable to CSRF. When the form is submitted, both packages use some middleware to check that the hidden field value and cookie value match.

Out of the two packages, we'll opt to use [justinas/nosurf](#) in this book. I prefer it primarily because it's self-contained and doesn't have any additional dependencies. If you're following along, you can install the latest version like so:

```
$ go get github.com/justinas/nosurf@v1
go: downloading github.com/justinas/nosurf v1.1.1
go get: added github.com/justinas/nosurf v1.1.1
```

Using the nosurf package

To use [justinas/nosurf](#), open up your [cmd/web/middleware.go](#) file and create a new `noSurf()` middleware function like so:

```
File: cmd/web/middleware.go
```

```
package main

import (
    "fmt"
    "net/http"

    "github.com/justinas/nosurf" // New import
)

...

// Create a NoSurf middleware function which uses a customized CSRF cookie with
// the Secure, Path and HttpOnly attributes set.
func noSurf(next http.Handler) http.Handler {
    csrfHandler := nosurf.New(next)
    csrfHandler.SetBaseCookie(http.Cookie{
        HttpOnly: true,
        Path:     "/",
        Secure:   true,
    })

    return csrfHandler
}
```

One of the forms that we need to protect from CSRF attacks is our logout form, which is included in our `nav tmpl` partial and could potentially appear on any page of our application. So, because of this, we need to use our `noSurf()` middleware on *all* of our application routes (apart from `GET /static/`).

So, let's update the `cmd/web/routes.go` file to add this `noSurf()` middleware to the `dynamic` middleware chain that we made earlier:

```
File: cmd/web/routes.go
```

```
package main

...

func (app *application) routes() http.Handler {
    mux := http.NewServeMux()

    fileServer := http.FileServer(http.Dir("./ui/static/"))
    mux.Handle("GET /static/", http.StripPrefix("/static", fileServer))

    // Use the nosurf middleware on all our 'dynamic' routes.
    dynamic := alice.New(app.sessionManager.LoadAndSave, noSurf)

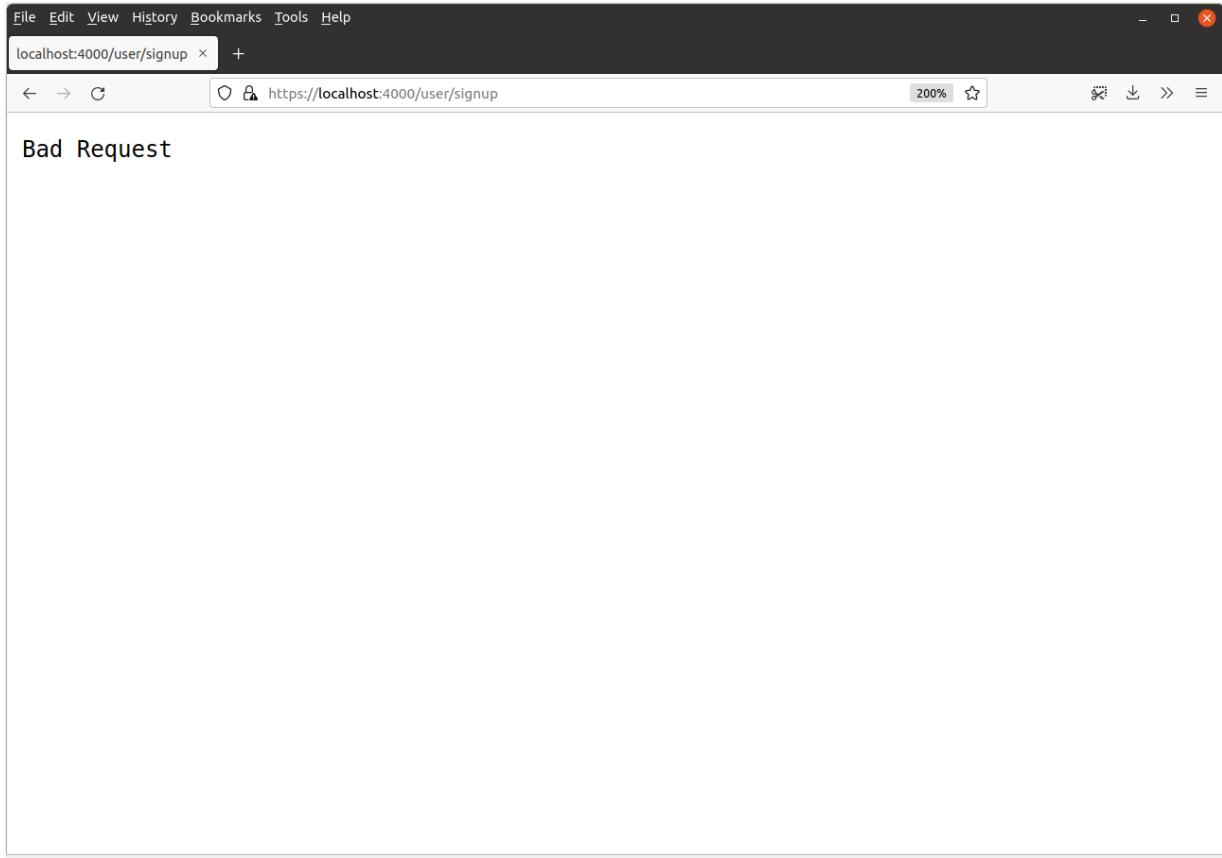
    mux.Handle("GET /${}", dynamic.ThenFunc(app.home))
    mux.Handle("GET /snippet/view/{id}", dynamic.ThenFunc(app.snippetView))
    mux.Handle("GET /user/signup", dynamic.ThenFunc(app.userSignup))
    mux.Handle("POST /user/signup", dynamic.ThenFunc(app.userSignupPost))
    mux.Handle("GET /user/login", dynamic.ThenFunc(app.userLogin))
    mux.Handle("POST /user/login", dynamic.ThenFunc(app.userLoginPost))

    protected := dynamic.Append(app.requireAuthentication)

    mux.Handle("GET /snippet/create", protected.ThenFunc(app.snippetCreate))
    mux.Handle("POST /snippet/create", protected.ThenFunc(app.snippetCreatePost))
    mux.Handle("POST /user/logout", protected.ThenFunc(app.userLogoutPost))

    standard := alice.New(app.recoverPanic, app.logRequest, commonHeaders)
    return standard.Then(mux)
}
```

At this point, you might like to fire up the application and try submitting one of the forms. When you do, the request should be intercepted by the `noSurf()` middleware and you should receive a `400 Bad Request` response.



To make the form submissions work, we need to use the `nosurf.Token()` function to get the CSRF token and add it to a hidden `csrf_token` field in each of our forms. So the next step is to add a new `CSRFToken` field to our `templateData` struct:

```
File: cmd/web/templates.go

package main

import (
    "html/template"
    "path/filepath"
    "time"

    "snippetbox.alexandedwards.net/internal/models"
)

type templateData struct {
    CurrentYear      int
    Snippet          models.Snippet
    Snippets         []models.Snippet
    Form             any
    Flash            string
    IsAuthenticated bool
    CSRFToken        string // Add a CSRFToken field.
}

...
```

And because the logout form can potentially appear on every page, it makes sense to add

the CSRF token to the template data automatically via our `newTemplateData()` helper. This will mean that it will be available to our templates each time we render a page.

Please go ahead and update the `cmd/web/helpers.go` file as follows:

```
File: cmd/web/helpers.go

package main

import (
    "bytes"
    "errors"
    "fmt"
    "net/http"
    "time"

    "github.com/go-playground/form/v4"
    "github.com/justinas/nosurf" // New import
)

...

func (app *application) newTemplateData(r *http.Request) templateData {
    return templateData{
        CurrentYear:     time.Now().Year(),
        Flash:           app.sessionManager.PopString(r.Context(), "flash"),
        IsAuthenticated: app.isAuthenticated(r),
        CSRFToken:       nosurf.Token(r), // Add the CSRF token.
    }
}

...
```

Finally, we need to update all the forms in our application to include this CSRF token in a hidden field.

Like so:

File: ui/html/pages/create.tpl

```
 {{define "title"}}Create a New Snippet{{end}}
```

```
 {{define "main"}}
<form action='/snippet/create' method='POST'>
    <!-- Include the CSRF token -->
    <input type='hidden' name='csrf_token' value='{{.CSRFToken}}'>
    <div>
        <label>Title:</label>
        {{with .Form.FieldErrors.title}}
            <label class='error'>{{.}}</label>
        {{end}}
        <input type='text' name='title' value='{{.Form.Title}}'>
    </div>
    <div>
        <label>Content:</label>
        {{with .Form.FieldErrors.content}}
            <label class='error'>{{.}}</label>
        {{end}}
        <textarea name='content'>{{.Form.Content}}</textarea>
    </div>
    <div>
        <label>Delete in:</label>
        {{with .Form.FieldErrors.expires}}
            <label class='error'>{{.}}</label>
        {{end}}
        <input type='radio' name='expires' value='365' {{if (eq .Form.Expires 365)}}checked{{end}}> One Year
        <input type='radio' name='expires' value='7' {{if (eq .Form.Expires 7)}}checked{{end}}> One Week
        <input type='radio' name='expires' value='1' {{if (eq .Form.Expires 1)}}checked{{end}}> One Day
    </div>
    <div>
        <input type='submit' value='Publish snippet'>
    </div>
</form>
{{end}}
```

File: ui/html/pages/login.tpl

```
 {{define "title"}}Login{{end}}
```

```
 {{define "main"}}
<form action='/user/login' method='POST' novalidate>
    <!-- Include the CSRF token -->
    <input type='hidden' name='csrf_token' value='{{.CSRFToken}}'>
    {{range .Form.NonFieldErrors}}
        <div class='error'>{{.}}</div>
    {{end}}
    <div>
        <label>Email:</label>
        {{with .Form.FieldErrors.email}}
            <label class='error'>{{.}}</label>
        {{end}}
        <input type='email' name='email' value='{{.Form.Email}}'>
    </div>
    <div>
        <label>Password:</label>
        {{with .Form.FieldErrors.password}}
            <label class='error'>{{.}}</label>
        {{end}}
        <input type='password' name='password'>
    </div>
    <div>
        <input type='submit' value='Login'>
    </div>
</form>
{{end}}
```

File: ui/html/pages/signup.tpl

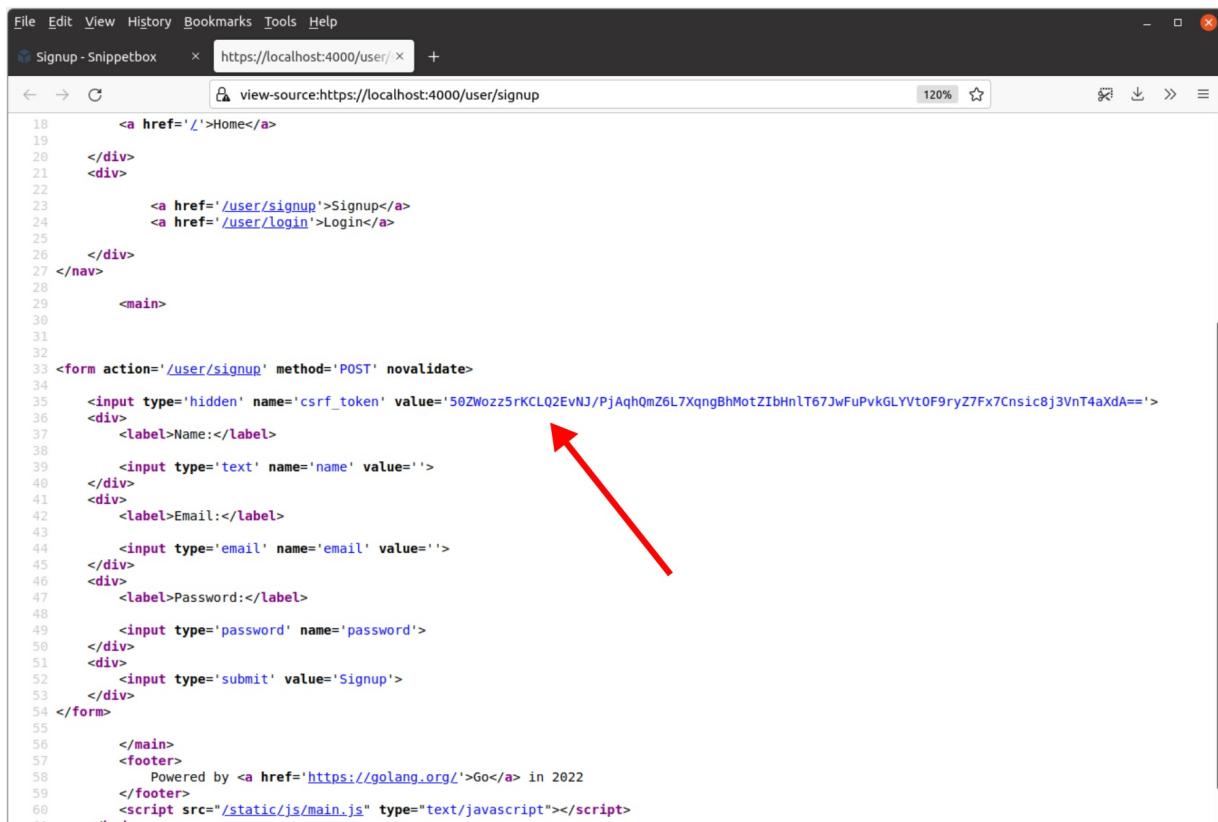
```
 {{define "title"}}Signup{{end}}
```

```
 {{define "main"}}
<form action='/user/signup' method='POST' novalidate>
    <!-- Include the CSRF token -->
    <input type='hidden' name='csrf_token' value='{{.CSRFToken}}'>
    <div>
        <label>Name:</label>
        {{with .Form.FieldErrors.name}}
            <label class='error'>{{.}}</label>
        {{end}}
        <input type='text' name='name' value='{{.Form.Name}}'>
    </div>
    <div>
        <label>Email:</label>
        {{with .Form.FieldErrors.email}}
            <label class='error'>{{.}}</label>
        {{end}}
        <input type='email' name='email' value='{{.Form.Email}}'>
    </div>
    <div>
        <label>Password:</label>
        {{with .Form.FieldErrors.password}}
            <label class='error'>{{.}}</label>
        {{end}}
        <input type='password' name='password'>
    </div>
    <div>
        <input type='submit' value='Signup'>
    </div>
</form>
{{end}}
```

File: ui/html/partials/nav.tpl

```
 {{define "nav"}}
<nav>
    <div>
        <a href='/'>Home</a>
        {{if .IsAuthenticated}}
            <a href='/snippet/create'>Create snippet</a>
        {{end}}
    </div>
    <div>
        {{if .IsAuthenticated}}
            <form action='/user/logout' method='POST'>
                <!-- Include the CSRF token -->
                <input type='hidden' name='csrf_token' value='{{.CSRFToken}}'>
                <button>Logout</button>
            </form>
        {{else}}
            <a href='/user/signup'>Signup</a>
            <a href='/user/login'>Login</a>
        {{end}}
    </div>
</nav>
{{end}}
```

Go ahead and run the application again, then view source of one of the forms. You should see that it now has a CSRF token included in a hidden field, like so.



```
File Edit View History Bookmarks Tools Help
Signup - Snippetbox https://localhost:4000/user/signup +
18     <a href='/L'>Home</a>
19
20     </div>
21     <div>
22         <a href='/user/signup'>Signup</a>
23         <a href='/user/login'>Login</a>
24
25     </div>
26 </nav>
27
28     <main>
29
30
31
32
33 <form action='/user/signup' method='POST' novalidate>
34
35     <input type='hidden' name='csrf_token' value='50ZWozz5rKCLQ2EvNJPjAqhQmZ6L7XqngBhMotZIBhnlT67JwFuPvkGLYVt0F9ryZ7Fx7Cnsic8j3VnT4aXdA=='>
36     <div>
37         <label>Name:</label>
38
39         <input type='text' name='name' value='>
40     </div>
41     <div>
42         <label>Email:</label>
43
44         <input type='email' name='email' value='>
45     </div>
46     <div>
47         <label>Password:</label>
48
49         <input type='password' name='password'>
50     </div>
51     <div>
52         <input type='submit' value='Signup'>
53     </div>
54 </form>
55
56     <main>
57     <footer>
58         Powered by <a href='https://golang.org/'>Go</a> in 2022
59     </footer>
60     <script src="/static/js/main.js" type="text/javascript"></script>
```

And if you try submitting the forms, it should now work correctly again.

Additional information

SameSite ‘Strict’ setting

If you want, you can change the session cookie to use the `SameSite=Strict` setting instead of (the default) `SameSite=Lax`. Like this:

```
sessionManager := scs.New()
sessionManager.Cookie.SameSite = http.SameSiteStrictMode
```

But it’s important to be aware that using `SameSite=Strict` will block the session cookie being sent by the user’s browser for *all* cross-site usage — including `safe` requests with HTTP methods like `GET` and `HEAD`.

While that might sound even safer (and it is!) the downside is that the session cookie won't be sent when a user clicks on a link to your application from another website. In turn, that means that your application would initially treat the user as 'not logged in' even if they have an active session containing their "`authenticatedUserID`" value.

So if your application will potentially have other websites linking to it (or links to it shared in emails or private messaging services), then `SameSite=Lax` is generally the more appropriate setting.

SameSite cookies and TLS 1.3

Earlier in this chapter I said that we can't solely rely on the `SameSite` cookie attribute to prevent CSRF attacks, because it isn't fully supported by all browsers.

But there is an [exception to this](#) rule, due to the fact that no browser exists *which supports TLS 1.3 and does not support SameSite cookies*.

In other words, if you were to make TLS 1.3 the minimum supported version in the TLS config for your server, then all browsers able to use your application will support `SameSite` cookies.

```
tlsConfig := &tls.Config{
    MinVersion: tls.VersionTLS13,
}
```

So long as you only allow HTTPS requests to your application and enforce TLS 1.3 as the minimum TLS version, you don't need to make any additional mitigation against CSRF attacks (like using the `justinas/nosurf` package). Just make sure that you always:

- Set `SameSite=Lax` or `SameSite=Strict` on the session cookie; and
- Use the `POST`, `PUT` or `DELETE` HTTP methods for any state-changing requests.

Using request context

At the moment our logic for authenticating a user consists of simply checking whether a "`authenticatedUserID`" value exists in their session data, like so:

```
func (app *application) isAuthenticated(r *http.Request) bool {
    return app.sessionManager.Exists(r.Context(), "authenticatedUserID")
}
```

We could make this check more robust by querying our `users` database table to make sure that the "`authenticatedUserID`" value is a real, valid, value (i.e we haven't deleted the user's account since they last logged in).

But there is a slight problem with doing this additional database check.

Our `isAuthenticated()` helper can potentially be called multiple times in each request cycle. Currently we use it twice — once in the `requireAuthentication()` middleware and again in the `newTemplateData()` helper. So, if we query the database from the `isAuthenticated()` helper directly, we would end up making duplicated round-trips to the database during every request. And that's not very efficient.

A better approach would be to carry out this check in some middleware to determine whether the current request is from an authenticated user or not, and then pass that information down to all subsequent handlers in the chain.

So how do we do this? Enter request context.

In this section you'll learn:

- [What request context is](#), how to use it, and when it is appropriate to use it.
- How to [use request context in practice](#) to pass information about the current user between your handlers.

How request context works

Every `http.Request` that our middleware and handlers process has a `context.Context` object embedded in it, which we can use to store information during the lifetime of the request.

As I've already hinted at, in a web application a common use-case for this is to pass information between your pieces of middleware and other handlers.

In our case, we want to use it to check if a user is authenticated once in some middleware, and if they are, then make this information available to all our other middleware and handlers.

Let's start with some theory and explain the syntax for working with request context. Then, in the next chapter, we'll get a bit more concrete again and demonstrate how to practically use it in our application.

The request context syntax

The basic code for adding information to a request's context looks like this:

```
// Where r is a *http.Request...
ctx := r.Context()
ctx = context.WithValue(ctx, "isAuthenticated", true)
r = r.WithContext(ctx)
```

Let's step through this line-by-line.

- First, we use the `r.Context()` method to retrieve the *existing* context from a request and assign it to the `ctx` variable.
- Then we use the `contextWithValue()` method to create a *new copy* of the existing context, containing the key "`isAuthenticated`" and a value of `true`.
- Then finally we use the `r.WithContext()` method to create a *copy* of the request containing our new context.

Important: Notice that we don't actually update the context for a request directly. What we're doing is *creating a new copy* of the `http.Request` object with our new context in it.

I should also point out that, for clarity, I made that code snippet a bit more verbose than it needs to be. It's more typical to write it like this:

```
ctx = contextWithValue(r.Context(), "isAuthenticated", true)
r = r.WithContext(ctx)
```

So that's how you add data to a request's context. But what about retrieving it again?

The important thing to explain is that, behind the scenes, request context values are stored with the type `any`. And that means that, after retrieving them from the context, you'll need to assert them to their original type before you use them.

To retrieve a value we need to use the `r.Context().Value()` method, like so:

```
isAuthenticated, ok := r.Context().Value("isAuthenticated").(bool)
if !ok {
    return errors.New("could not convert value to bool")
}
```

Avoiding key collisions

In the code samples above, I've used the string `"isAuthenticated"` as the key for storing and retrieving the data from a request's context. But this isn't recommended because there's a risk that other third-party packages used by your application will also want to store data using the key `"isAuthenticated"` — and that would cause a naming collision.

To avoid this, it's good practice to create your own custom type which you can use for your context keys. Extending our sample code, it's much better to do something like this:

```
// Declare a custom "contextKey" type for your context keys.  
type contextKey string  
  
// Create a constant with the type contextKey that we can use.  
const isAuthenticatedContextKey = contextKey("isAuthenticated")  
  
...  
  
// Set the value in the request context, using our isAuthenticatedContextKey  
// constant as the key.  
ctx := r.Context()  
ctx = contextWithValue(ctx, isAuthenticatedContextKey, true)  
r = r.WithContext(ctx)  
  
...  
  
// Retrieve the value from the request context using our constant as the key.  
isAuthenticated, ok := r.Context().Value(isAuthenticatedContextKey).(bool)  
if !ok {  
    return errors.New("could not convert value to bool")  
}
```

Request context for authentication/authorization

So, with those explanations out of the way, let's start to use the request context functionality in our application.

We'll begin by heading back to our `internal/models/users.go` file and fleshing out the `UserModel.Exists()` method, so that it returns `true` if a user with a specific ID exists in our `users` table, and `false` otherwise. Like so:

```
File: internal/models/users.go

package models

...

func (m *UserModel) Exists(id int) (bool, error) {
    var exists bool

    stmt := "SELECT EXISTS(SELECT true FROM users WHERE id = ?)"

    err := m.DB.QueryRow(stmt, id).Scan(&exists)
    return exists, err
}
```

Then let's create a new `cmd/web/context.go` file. In this file we'll define a custom `contextKey` type and an `isAuthenticatedContextKey` variable, so that we have a unique key we can use to store and retrieve the authentication status from a request context (without the risk of naming collisions).

```
$ touch cmd/web/context.go
```

```
File: cmd/web/context.go

package main

type contextKey string

const isAuthenticatedContextKey = contextKey("isAuthenticated")
```

And now for the exciting part. Let's create a new `authenticate()` middleware method which:

1. Retrieves the user's ID from their session data.
2. Checks the database to see if the ID corresponds to a valid user using the `UserModel.Exists()` method.
3. Updates the request context to include an `isAuthenticatedContextKey` key with the value `true`.

Here's the code:

```
File: cmd/web/middleware.go

package main

import (
    "context" // New import
    "fmt"
    "net/http"

    "github.com/justinas/nosurf"
)

...

func (app *application) authenticate(next http.Handler) http.Handler {
    return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
        // Retrieve the authenticatedUserID value from the session using the
        // GetInt() method. This will return the zero value for an int (0) if no
        // "authenticatedUserID" value is in the session -- in which case we
        // call the next handler in the chain as normal and return.
        id := app.sessionManager.GetInt(r.Context(), "authenticatedUserID")
        if id == 0 {
            next.ServeHTTP(w, r)
            return
        }

        // Otherwise, we check to see if a user with that ID exists in our
        // database.
        exists, err := app.users.Exists(id)
        if err != nil {
            app.serverError(w, r, err)
            return
        }

        // If a matching user is found, we know that the request is
        // coming from an authenticated user who exists in our database. We
        // create a new copy of the request (with an isAuthenticatedContextKey
        // value of true in the request context) and assign it to r.
        if exists {
            ctx := context.WithValue(r.Context(), isAuthenticatedContextKey, true)
            r = r.WithContext(ctx)
        }

        // Call the next handler in the chain.
        next.ServeHTTP(w, r)
    })
}
```

The important thing to emphasize here is the following difference:

- When we *don't* have a valid authenticated user, we pass the original and unchanged `*http.Request` to the next handler in the chain.
- When we *do* have a valid authenticated user, we create a copy of the request with a `isAuthenticatedContextKey` key and `true` value stored in the request context. We then pass this copy of `*http.Request` to the next handler in the chain.

Alright, let's update the `cmd/web/routes.go` file to include the `authenticate()` middleware in our `dynamic` middleware chain:

```
File: cmd/web/routes.go

package main

...

func (app *application) routes() http.Handler {
    mux := http.NewServeMux()

    fileServer := http.FileServer(http.Dir("./ui/static/"))
    mux.Handle("GET /static/", http.StripPrefix("/static", fileServer))

    // Add the authenticate() middleware to the chain.
    dynamic := alice.New(app.sessionManager.LoadAndSave, noSurf, app.authenticate)

    mux.Handle("GET /${}", dynamic.ThenFunc(app.home))
    mux.Handle("GET /snippet/view/{id}", dynamic.ThenFunc(app.snippetView))
    mux.Handle("GET /user/signup", dynamic.ThenFunc(app.userSignup))
    mux.Handle("POST /user/signup", dynamic.ThenFunc(app.userSignupPost))
    mux.Handle("GET /user/login", dynamic.ThenFunc(app.userLogin))
    mux.Handle("POST /user/login", dynamic.ThenFunc(app.userLoginPost))

    protected := dynamic.Append(app.requireAuthentication)

    mux.Handle("GET /snippet/create", protected.ThenFunc(app.snippetCreate))
    mux.Handle("POST /snippet/create", protected.ThenFunc(app.snippetCreatePost))
    mux.Handle("POST /user/logout", protected.ThenFunc(app.userLogoutPost))

    standard := alice.New(app.recoverPanic, app.logRequest, commonHeaders)
    return standard.Then(mux)
}
```

The last thing that we need to do is update our `isAuthenticated()` helper, so that instead of checking the session data it now checks the request context to determine if a user is authenticated or not.

We can do this like so:

```
File: cmd/web/helpers.go

package main

...

func (app *application) isAuthenticated(r *http.Request) bool {
    isAuthenticated, ok := r.Context().Value(isAuthenticatedContextKey).(bool)
    if !ok {
        return false
    }

    return isAuthenticated
}
```

It's important to point out here that if there isn't a value in the request context with the `isAuthenticatedContextKey` key, or the underlying value isn't a `bool`, then this type assertion will fail. In that case we take a 'safe' fall back and return false (i.e we assume that the user isn't authenticated).

If you like, try running the application again. It should compile correctly and if you log in as a certain user and browse around the application should work exactly as before.

Then, if you want, open MySQL and delete the record for the user that you're logged in as from the database. For example:

```
mysql> USE snippetbox;
mysql> DELETE FROM users WHERE email = 'bob@example.com';
```

And when you go back to your browser and refresh the page, the application is now smart enough to recognize that the user has been deleted, and you'll find yourself treated as an unauthenticated (logged-out) user.

Additional information

Misusing request context

It's important to emphasize that request context should only be used to store information relevant to the lifetime of a specific request. The Go documentation for `context.Context` warns:

Use context Values only for request-scoped data that transits processes and APIs.

That means you should not use it to pass dependencies that exist *outside of the lifetime of a request* — like loggers, template caches and your database connection pool — to your middleware and handlers.

For reasons of type-safety and clarity of code, it's almost always better to make these dependencies available to your handlers explicitly, by either making your handlers methods against an [application](#) struct (like we have in this book) or passing them in a closure (like in [this Gist](#)).

File embedding

The Go standard library includes an [embed](#) package, which makes it possible to *embed external files into your Go program itself*.

In this section of the book, we'll update our application so that it embeds the files from our [ui](#) directory — starting with the static CSS, JavaScript and image files, and then moving on to the HTML templates.

Although using the [embed](#) package is completely optional (and our application works fine as-is), it opens up the opportunity to create Go programs that are self-contained and have everything that they need to run *as part of the compiled binary executable*. In turn, that makes it easier to deploy or distribute your web application.

Let's jump straight in and explain how to use it.

Embedding static files

If you're following along, the first thing to do is create a new `ui/efs.go` file:

```
$ touch ui/efs.go
```

Then add the following code:

```
File: ui/efs.go

package ui

import (
    "embed"
)

//go:embed "static"
var Files embed.FS
```

The important line here is `//go:embed "static"`.

This looks like a comment, but it is actually a special *comment directive*. When our application is compiled (as part of either `go build` or `go run`), this comment directive instructs Go to store the files from our `ui/static` folder in an *embedded filesystem* referenced by the global variable `Files`.

There are a few important details about this which we need to explain.

- The comment directive must be placed *immediately above* the variable in which you want to store the embedded files.
- The directive has the format `go:embed "<path>"`. The path is relative to the `.go` file containing the directive, so — in our case — `go:embed "static"` embeds the directory `ui/static` from our project.
- You can only use the `go:embed` directive on global variables at package level, not within functions or methods. If you try to use it within a function or method, you'll get the error `"go:embed cannot apply to var inside func"` at compile time.
- Paths cannot contain `.` or `..` elements, nor may they begin or end with a `/`. This essentially restricts you to only embedding files or directories that are within the same

directory as the `.go` file containing the `go:embed` directive.

- The embedded file system is *always* rooted in the directory which contains the `go:embed` directive. So, in the example above, our `Files` variable contains an `embed.FS` embedded filesystem and the root of that filesystem is our `ui` directory.

Using the embedded static files

Now let's switch up our application so that it serves our static CSS, JavaScript and image files from the embedded file system — instead of reading them from the disk at runtime.

Open your `cmd/web/routes.go` file and update it as follows:

```
File: cmd/web/routes.go
```

```
package main

import (
    "net/http"

    "snippetbox.alexedwards.net/ui" // New import
    "github.com/justinas/alice"
)

func (app *application) routes() http.Handler {
    mux := http.NewServeMux()

    // Use the http.FileServerFS() function to create a HTTP handler which
    // serves the embedded files in ui.Files. It's important to note that our
    // static files are contained in the "static" folder of the ui.Files
    // embedded filesystem. So, for example, our CSS stylesheet is located at
    // "static/css/main.css". This means that we no longer need to strip the
    // prefix from the request URL -- any requests that start with /static/ can
    // just be passed directly to the file server and the corresponding static
    // file will be served (so long as it exists).
    mux.Handle("GET /static/", http.FileServerFS(ui.Files))

    dynamic := alice.New(app.sessionManager.LoadAndServe, noSurf, app.authenticate)

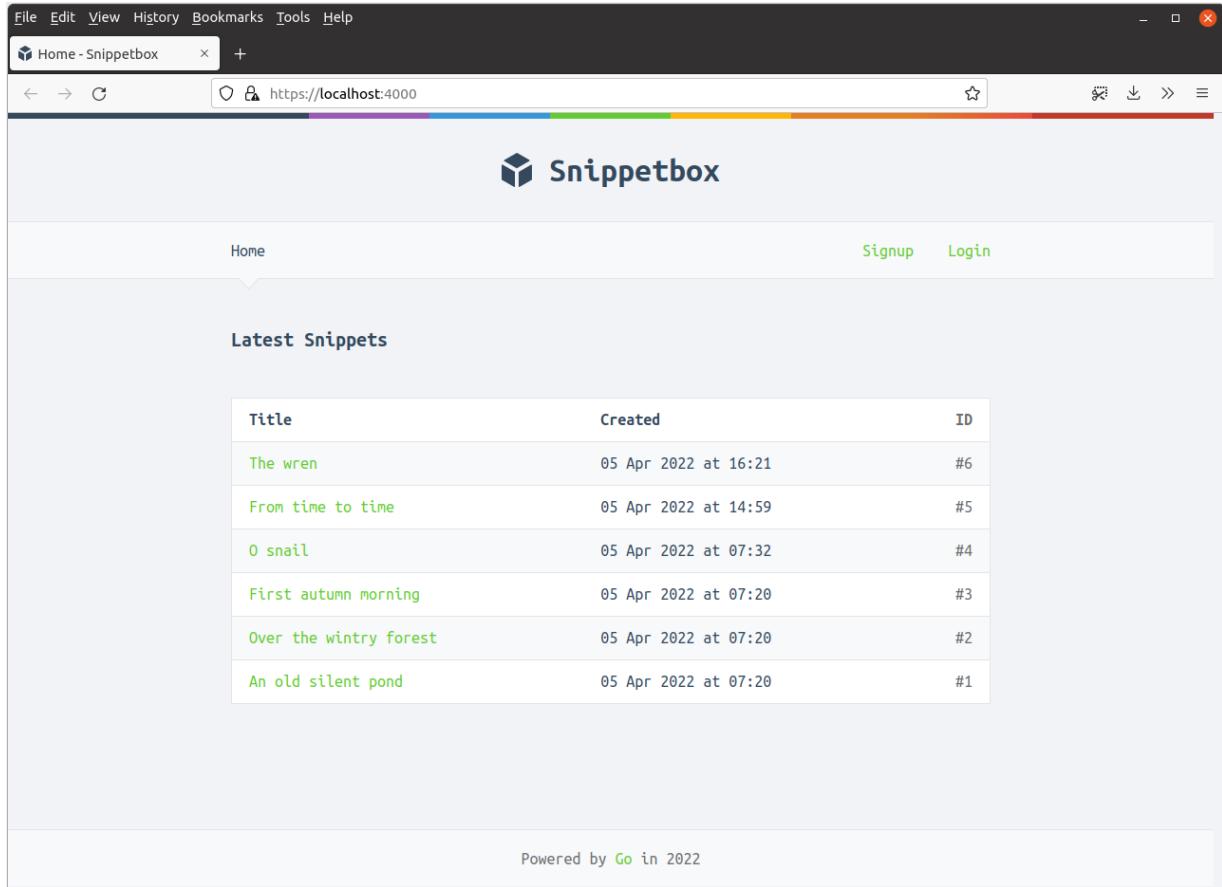
    mux.Handle("GET {$}", dynamic.ThenFunc(app.home))
    mux.Handle("GET /snippet/view/{id}", dynamic.ThenFunc(app.snippetView))
    mux.Handle("GET /user/signup", dynamic.ThenFunc(app.userSignup))
    mux.Handle("POST /user/signup", dynamic.ThenFunc(app.userSignupPost))
    mux.Handle("GET /user/login", dynamic.ThenFunc(app.userLogin))
    mux.Handle("POST /user/login", dynamic.ThenFunc(app.userLoginPost))

    protected := dynamic.Append(app.requireAuthentication)

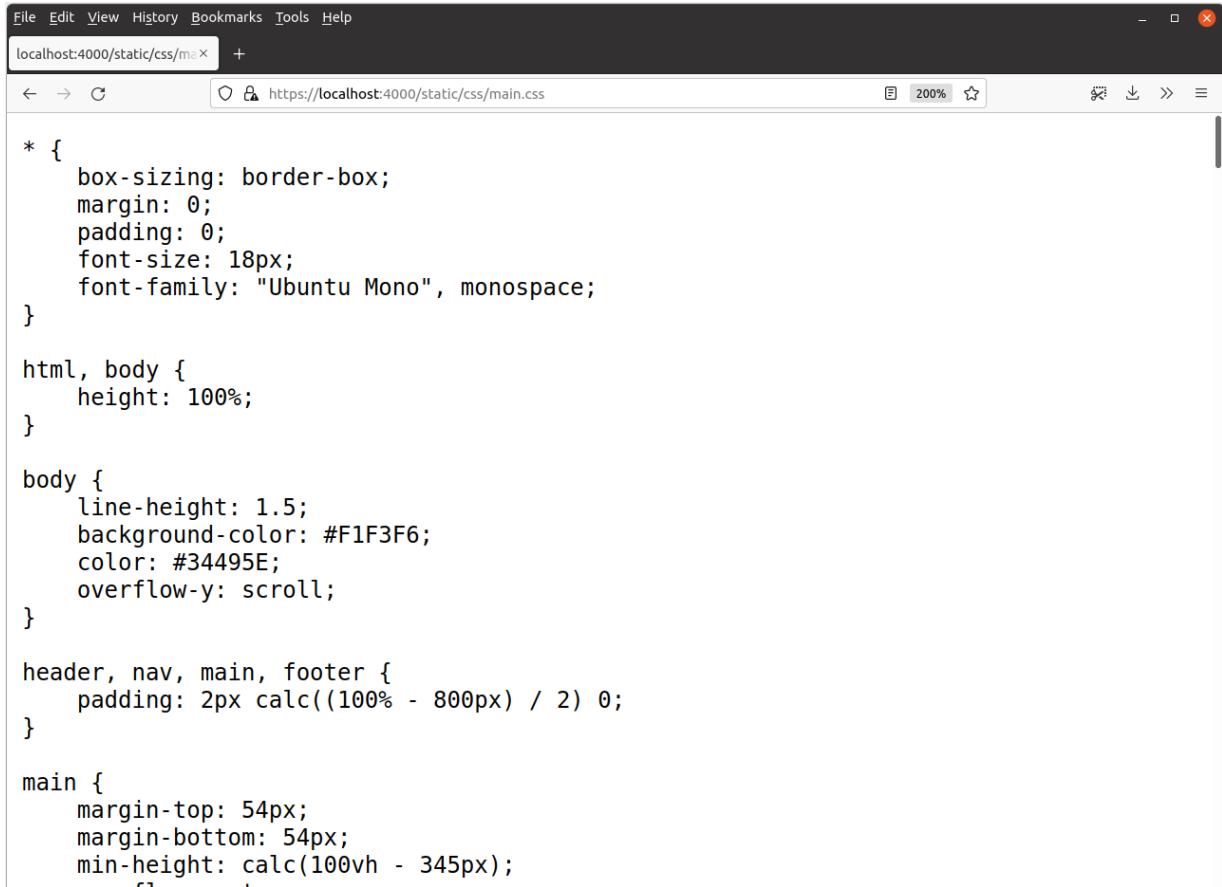
    mux.Handle("GET /snippet/create", protected.ThenFunc(app.snippetCreate))
    mux.Handle("POST /snippet/create", protected.ThenFunc(app.snippetCreatePost))
    mux.Handle("POST /user/logout", protected.ThenFunc(app.userLogoutPost))

    standard := alice.New(app.recoverPanic, app.logRequest, commonHeaders)
    return standard.Then(mux)
}
```

If you save these changes and then restart the application, you should find that everything compiles and runs correctly. When you visit <https://localhost:4000> in your browser, the static files should be served from the embedded filesystem and everything should look normal.



If you want, you can also navigate directly to the static files to check that they are still available. For example, visiting <https://localhost:4000/static/css/main.css> should display the CSS stylesheet for the webpage from the embedded filesystem.



A screenshot of a web browser window displaying the CSS code for a file named 'main.css'. The browser's address bar shows 'localhost:4000/static/css/main.css'. The code itself is a standard CSS reset followed by styling for the main content area.

```
* {  
    box-sizing: border-box;  
    margin: 0;  
    padding: 0;  
    font-size: 18px;  
    font-family: "Ubuntu Mono", monospace;  
}  
  
html, body {  
    height: 100%;  
}  
  
body {  
    line-height: 1.5;  
    background-color: #F1F3F6;  
    color: #34495E;  
    overflow-y: scroll;  
}  
  
header, nav, main, footer {  
    padding: 2px calc((100% - 800px) / 2) 0;  
}  
  
main {  
    margin-top: 54px;  
    margin-bottom: 54px;  
    min-height: calc(100vh - 345px);  
}
```

Additional information

Multiple paths

It's totally OK to specify multiple paths in one embed directive. For example, we could individually embed the `ui/static/css`, `ui/static/img` and `ui/static/js` directories like so:

```
//go:embed "static/css" "static/img" "static/js"  
var Files embed.FS
```

Important: The path separator in embed path patterns should always be a forward slash `/`, even on Windows machines.

Embedding specific files

I alluded to this at the start of the chapter, but it's possible for an embed path to point to a *specific file*. Embedding isn't just limited to directories.

For example, let's pretend that our `ui/static/css` directory contains some additional assets that we don't want to embed, such as `Sass` or `Less` files. In that case, we could embed just the `ui/static/css/main.css` file like so:

```
//go:embed "static/css/main.css" "static/img" "static/js"  
var Files embed.FS
```

Wildcard paths

The character `*` can be used as a 'wildcard' in an embed path. Continuing with the example above, we could rewrite the embed directive so that only `.css` files under `ui/static/css` are embedded:

```
//go:embed "static/css/*.css" "static/img" "static/js"  
var Files embed.FS
```

Related to that, if you use the wildcard path `"*"` without any qualifiers, like this:

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

... then it will embed everything in the current directory, including the `.go` file that contains the embed directive itself! Most of the time you don't want that, so it's more common to explicitly embed specific subdirectories or files instead.

The `all` prefix

Finally, if a path is to a directory, then all files in that directory are recursively embedded — except for files with names that begin with `.` or `_` characters. If you want to include those files too, then you should use the `all:` prefix at the start of the path.

```
//go:embed "all:static"  
var Files embed.FS
```

Embedding HTML templates

Next let's update our application so that the template cache uses embedded HTML template files, instead reading them from your hard disk at runtime.

Head back to the `ui/efs.go` file, and update it so that `ui.Files` embeds the contents of the `ui/html` directory (which contains our templates) too. Like so:

```
File: ui/efs.go

package ui

import (
    "embed"
)

//go:embed "html" "static"
var Files embed.FS
```

Then we need to update the `newTemplateCache()` function in `cmd/web/templates.go` so that it reads the templates from `ui.Files`. To do this, we'll need to leverage a couple of the special features that Go has for working with embedded filesystems:

- The `fs.Glob()` function returns a slice of filepaths matching a glob pattern. It's effectively the same as the `filepath.Glob()` function that we used earlier in the book, except that it works on embedded filesystems.
- The `Template.ParseFS()` method can be used to parse the HTML templates from an embedded filesystem into a template set. This is effectively a replacement for *both* the `Template.ParseFiles()` and `Template.ParseGlob()` methods that we used earlier. `Template.ParseFiles()` is also a *variadic function*, which allows you to parse multiple templates in a single call to `ParseFiles()`.

Let's put these to use in our `cmd/web/templates.go` file:

```
File: cmd/web/templates.go
```

```
package main

import (
    "html/template"
    "io/fs" // New import
    "path/filepath"
    "time"

    "snippetbox.alexedwards.net/internal/models"
    "snippetbox.alexedwards.net/ui" // New import
)

...

func newTemplateCache() (*map[string]*template.Template, error) {
    cache := map[string]*template.Template{}

    // Use fs.Glob() to get a slice of all filepaths in the ui.Files embedded
    // filesystem which match the pattern 'html/pages/*.tmpl'. This essentially
    // gives us a slice of all the 'page' templates for the application, just
    // like before.
    pages, err := fs.Glob(ui.Files, "html/pages/*.tmpl")
    if err != nil {
        return nil, err
    }

    for _, page := range pages {
        name := filepath.Base(page)

        // Create a slice containing the filepath patterns for the templates we
        // want to parse.
        patterns := []string{
            "html/base.tmpl",
            "html/partials/*.tmpl",
            page,
        }

        // Use ParseFS() instead of ParseFiles() to parse the template files
        // from the ui.Files embedded filesystem.
        ts, err := template.New(name).Funcs(functions).ParseFS(ui.Files, patterns...)
        if err != nil {
            return nil, err
        }

        cache[name] = ts
    }

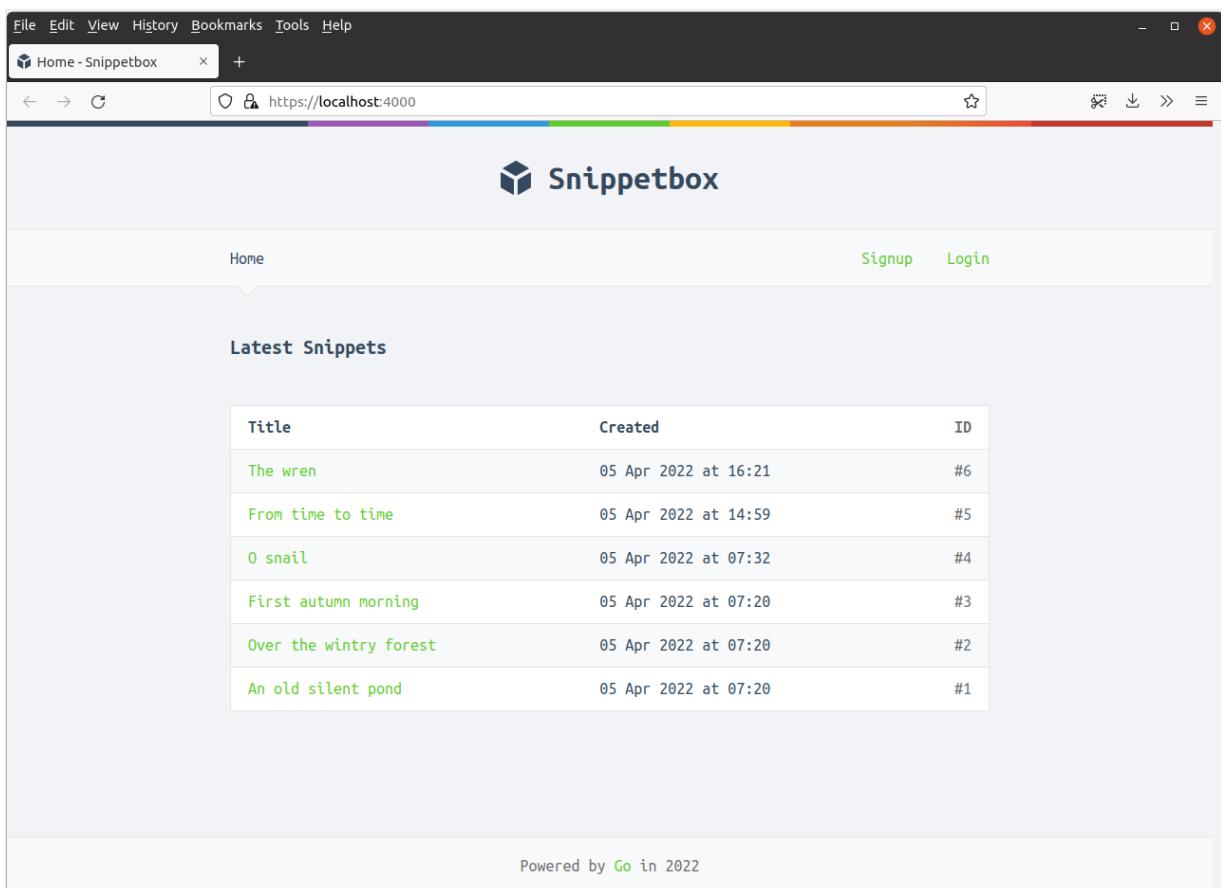
    return cache, nil
}
```

Now that this is done, when our application is built into a binary it will contain all the UI files that it needs to run.

You can try this out quickly by building an executable binary in your `/tmp` directory, copying over the TLS certificates, and running the binary. Like so:

```
$ go build -o /tmp/web ./cmd/web/
$ cp -r ./tls /tmp/
$ cd /tmp/
$ ./web
time=2024-03-18T11:29:23.000+00:00 level=INFO msg="starting server" addr=:4000
```

And again, you should be able to visit <https://localhost:4000> in your browser and everything should work correctly — despite the binary being in a location where it does not have access to the original UI files on disk.



Note: If you'd like to learn about building binaries and deploying applications, there is a lot more information and detailed explanation available in [Let's Go Further](#).

Testing

And so we finally come to the topic of testing.

Like structuring and organizing your application code, there's no single 'right' way to structure and organize your tests in Go. But there *are* some conventions, patterns and good-practices that you can follow.

In this section we're going to add tests for a selection of the code in our application, with the goal of demonstrating the general syntax for creating tests and illustrating some patterns that you can reuse in a wide-variety of applications.

You'll learn:

- How to create and run table-driven [unit tests and sub-tests](#) in Go.
- How to unit [test your HTTP handlers](#) and middleware.
- How to perform '[end-to-end](#)' testing of your web application routes, middleware and handlers.
- How to [create mocks](#) of your database models and use them in unit tests.
- A pattern for testing CSRF-protected [HTML form submissions](#).
- How to use a test instance of MySQL to perform [integration tests](#).
- How to easily calculate and profile [code coverage](#) for your tests.

Unit testing and sub-tests

In this chapter we'll create a unit test to make sure that our `humanDate()` function (which we made back in the [custom template functions](#) chapter) is outputting `time.Time` values in the exact format that we want.

If you can't remember, the `humanDate()` function looks like this:

```
File: cmd/web/templates.go

package main

...

func humanDate(t time.Time) string {
    return t.Format("02 Jan 2006 at 15:04")
}

...
```

The reason that I want to start by testing this is because it's a simple function. We can explore the basic syntax and patterns for writing tests without getting too caught-up in *the functionality* that we're testing.

Creating a unit test

Let's jump straight in and create a unit test for this function.

In Go, it's standard practice to write your tests in `*_test.go` files which live directly alongside the code that you're testing. So, in this case, the first thing that we're going to do is create a new `cmd/web/templates_test.go` file to hold the test:

```
$ touch cmd/web/templates_test.go
```

And then we can create a new unit test for the `humanDate` function like so:

```

File: cmd/web/templates_test.go

package main

import (
    "testing"
    "time"
)

func TestHumanDate(t *testing.T) {
    // Initialize a new time.Time object and pass it to the humanDate function.
    tm := time.Date(2024, 3, 17, 10, 15, 0, 0, time.UTC)
    hd := humanDate(tm)

    // Check that the output from the humanDate function is in the format we
    // expect. If it isn't what we expect, use the t.Errorf() function to
    // indicate that the test has failed and log the expected and actual
    // values.
    if hd != "17 Mar 2024 at 10:15" {
        t.Errorf("got %q; want %q", hd, "17 Mar 2024 at 10:15")
    }
}

```

This pattern is the basic one that you'll use for nearly all tests that you write in Go. The important things to take away are:

- The test is just regular Go code, which calls the `humanDate()` function and checks that the result matches what we expect.
- Your unit tests are contained in a normal Go function with the signature `func(*testing.T)`.
- To be a valid unit test, the name of this function *must* begin with the word `Test`. Typically this is then followed by the name of the function, method or type that you're testing to help make it obvious at a glance *what* is being tested.
- You can use the `t.Errorf()` function to mark a test as *failed* and log a descriptive message about the failure. It's important to note that calling `t.Errorf()` *doesn't stop execution of your test* — after you call it Go will continue executing any remaining test code as normal.

Let's try this out. Save the file, then use the `go test` command to run all the tests in our `cmd/web` package like so:

```
$ go test ./cmd/web
ok      snippetbox.alexandedwards.net/cmd/web    0.005s
```

So, this is good stuff. The `ok` in this output indicates that all tests in the package (for now, only our `TestHumanDate()` test) passed without any problems.

If you want more detail, you can see exactly which tests are being run by using the `-v` flag to get the *verbose* output:

```
$ go test -v ./cmd/web
===[ RUN   TestHumanDate
--- PASS: TestHumanDate (0.00s)
PASS
ok      snippetbox.alexewards.net/cmd/web    0.007s
```

Table-driven tests

Let's now expand our `TestHumanDate()` function to cover some additional test cases. Specifically, we're going to update it to also check that:

1. If the input to `humanDate()` is the `zero time`, then it returns the empty string `""`.
2. The output from the `humanDate()` function always uses the UTC time zone.

In Go, an idiomatic way to run multiple test cases is to use table-driven tests.

Essentially, the idea behind table-driven tests is to create a ‘table’ of test cases containing the inputs and expected outputs, and to then loop over these, running each test case in a sub-test. There are a few ways you could set this up, but a common approach is to define your test cases in an slice of anonymous structs.

I'll demonstrate:

```
File: cmd/web/templates_test.go
```

```
package main

import (
    "testing"
    "time"
)

func TestHumanDate(t *testing.T) {
    // Create a slice of anonymous structs containing the test case name,
    // input to our humanDate() function (the tm field), and expected output
    // (the want field).
    tests := []struct {
        name string
        tm   time.Time
        want string
    }{
        {
            name: "UTC",
            tm:   time.Date(2024, 3, 17, 10, 15, 0, 0, time.UTC),
            want: "17 Mar 2024 at 10:15",
        },
        {
            name: "Empty",
            tm:   time.Time{},
            want: "",
        },
        {
            name: "CET",
            tm:   time.Date(2024, 3, 17, 10, 15, 0, 0, time.FixedZone("CET", 1*60*60)),
            want: "17 Mar 2024 at 09:15",
        },
    }

    // Loop over the test cases.
    for _, tt := range tests {
        // Use the t.Run() function to run a sub-test for each test case. The
        // first parameter to this is the name of the test (which is used to
        // identify the sub-test in any log output) and the second parameter is
        // an anonymous function containing the actual test for each case.
        t.Run(tt.name, func(t *testing.T) {
            hd := humanDate(tt.tm)

            if hd != tt.want {
                t.Errorf("got %q; want %q", hd, tt.want)
            }
        })
    }
}
```

Note: In the third test case we're using CET (Central European Time) as the time zone, which is one hour ahead of UTC. So we want the output from `humanDate()` (which should be in UTC) to be `17 Mar 2024 at 09:15`, not `17 Mar 2024 at 10:15`.

OK, let's run this and see what happens:

```
$ go test -v ./cmd/web
==== RUN    TestHumanDate
==== RUN    TestHumanDate/UTC
==== RUN    TestHumanDate/Empty
    templates_test.go:44: got "01 Jan 0001 at 00:00"; want ""
==== RUN    TestHumanDate/CET
    templates_test.go:44: got "17 Mar 2024 at 10:15"; want "17 Mar 2024 at 09:15"
--- FAIL: TestHumanDate (0.00s)
    --- PASS: TestHumanDate/UTC (0.00s)
    --- FAIL: TestHumanDate/Empty (0.00s)
    --- FAIL: TestHumanDate/CET (0.00s)

FAIL
FAIL    snippetbox.alexewards.net/cmd/web      0.003s
FAIL
```

So here we can see the individual output for each of our sub-tests. As you might have guessed, our first test case passed but the `Empty` and `CET` tests both failed. Notice how — for the failed test cases — we get the relevant failure message and filename and line number in the output?

Let's head back to our `humanDate()` function and update it to fix these two problems:

```
File: cmd/web/templates.go

package main

...
func humanDate(t time.Time) string {
    // Return the empty string if time has the zero value.
    if t.IsZero() {
        return ""
    }

    // Convert the time to UTC before formatting it.
    return t.UTC().Format("02 Jan 2006 at 15:04")
}
```

And when you re-run the tests everything should now pass.

```
$ go test -v ./cmd/web
==== RUN    TestHumanDate
==== RUN    TestHumanDate/UTC
==== RUN    TestHumanDate/Empty
==== RUN    TestHumanDate/CET
--- PASS: TestHumanDate (0.00s)
    --- PASS: TestHumanDate/UTC (0.00s)
    --- PASS: TestHumanDate/Empty (0.00s)
    --- PASS: TestHumanDate/CET (0.00s)

PASS
ok    snippetbox.alexewards.net/cmd/web      0.003s
```

Helpers for test assertions

As I mentioned briefly earlier in the book, over the next few chapters we'll be writing a lot of *test assertions* that are a variation of this pattern:

```
if actualValue != expectedValue {
    t.Errorf("got %v; want %v", actualValue, expectedValue)
}
```

Let's quickly abstract this code into a helper function.

If you're following along, go ahead and create a new `internal/assert` package:

```
$ mkdir internal/assert
$ touch internal/assert/assert.go
```

And then add the following code:

```
File: internal/assert/assert.go

package assert

import (
    "testing"
)

func Equal[T comparable](t *testing.T, actual, expected T) {
    t.Helper()

    if actual != expected {
        t.Errorf("got: %v; want: %v", actual, expected)
    }
}
```

Notice how `Equal()` is a *generic function*? This means that we'll be able to use it irrespective of what the type of the `actual` and `expected` values is. So long as both `actual` and `expected` have the *same* type and can be compared using the `!=` operator (for example, they are both `string` values, or both `int` values) our test code should compile and work fine when we call `Equal()`.

Note: The `t.Helper()` function that we're using in the code above indicates to the Go test runner that our `Equal()` function is a test helper. This means that when `t.Errorf()` is called from our `Equal()` function, the Go test runner will report the filename and line number of the code *which called* our `Equal()` function in the output.

With that in place, we can simplify our `TestHumanDate()` test like so:

```
File: cmd/web/templates_test.go

package main

import (
    "testing"
    "time"

    "snippetbox.alexedwards.net/internal/assert" // New import
)

func TestHumanDate(t *testing.T) {
    tests := []struct {
        name string
        tm   time.Time
        want string
    }{
        {
            name: "UTC",
            tm:   time.Date(2024, 3, 17, 10, 15, 0, 0, time.UTC),
            want: "17 Mar 2024 at 10:15",
        },
        {
            name: "Empty",
            tm:   time.Time{},
            want: "",
        },
        {
            name: "CET",
            tm:   time.Date(2024, 3, 17, 10, 15, 0, 0, time.FixedZone("CET", 1*60*60)),
            want: "17 Mar 2024 at 09:15",
        },
    }

    for _, tt := range tests {
        t.Run(tt.name, func(t *testing.T) {
            hd := humanDate(tt.tm)

            // Use the new assert.Equal() helper to compare the expected and
            // actual values.
            assert.Equal(t, hd, tt.want)
        })
    }
}
```

Additional information

Sub-tests without a table of test cases

It's important to point out that you don't need to use sub-tests in conjunction with table-driven tests (like we have done so far in this chapter). It's perfectly valid to execute sub-

tests by calling `t.Run()` consecutively in your test functions, similar to this:

```
func TestExample(t *testing.T) {
    t.Run("Example sub-test 1", func(t *testing.T) {
        // Do a test.
    })

    t.Run("Example sub-test 2", func(t *testing.T) {
        // Do another test.
    })

    t.Run("Example sub-test 3", func(t *testing.T) {
        // And another...
    })
}
```

Testing HTTP handlers and middleware

Let's move on and discuss some specific techniques for unit testing your HTTP handlers.

All the handlers that we've written for this project so far are a bit complex to test, and to introduce things I'd prefer to start off with something more simple.

So, if you're following along, head over to your `handlers.go` file and create a new `ping` handler function which returns a `200 OK` status code and an `"OK"` response body. It's the type of handler that you might want to implement for status-checking or uptime monitoring of your server.

```
File: cmd/web/handlers.go

package main

...

func ping(w http.ResponseWriter, r *http.Request) {
    w.Write([]byte("OK"))
}
```

In this chapter we'll create a new `TestPing` unit test which:

- Checks that the response status code written by the `ping` handler is `200`.
- Checks that the response body written by the `ping` handler is `"OK"`.

Recording responses

Go provides a bunch of useful tools in the `net/http/httpptest` package for helping to test your HTTP handlers.

One of these tools is the `httpptest.ResponseRecorder` type. This is essentially an implementation of `http.ResponseWriter` which records the response status code, headers and body instead of actually writing them to a HTTP connection.

So an easy way to unit test your handlers is to create a new `httpptest.ResponseRecorder`, pass it to the handler function, and then examine it again after the handler returns.

Let's try doing exactly that to test the `ping` handler function.

First, follow the Go conventions and create a new `handlers_test.go` file to hold the test...

```
$ touch cmd/web/handlers_test.go
```

And then add the following code:

```
File: cmd/web/handlers_test.go

package main

import (
    "bytes"
    "io"
    "net/http"
    "net/http/httpptest"
    "testing"

    "snippetbox.alexedwards.net/internal/assert"
)

func TestPing(t *testing.T) {
    // Initialize a new httpptest.ResponseRecorder.
    rr := httpptest.NewRecorder()

    // Initialize a new dummy http.Request.
    r, err := http.NewRequest(http.MethodGet, "/", nil)
    if err != nil {
        t.Fatal(err)
    }

    // Call the ping handler function, passing in the
    // httpptest.ResponseRecorder and http.Request.
    ping(rr, r)

    // Call the Result() method on the http.ResponseRecorder to get the
    // http.Response generated by the ping handler.
    rs := rr.Result()

    // Check that the status code written by the ping handler was 200.
    assert.Equal(t, rs.StatusCode, http.StatusOK)

    // And we can check that the response body written by the ping handler
    // equals "OK".
    defer rs.Body.Close()
    body, err := io.ReadAll(rs.Body)
    if err != nil {
        t.Fatal(err)
    }
    body = bytes.TrimSpace(body)

    assert.Equal(t, string(body), "OK")
}
```

Note: In the code above we use the `t.Fatal()` function in a couple of places to handle situations where there is an unexpected error in our test code. When called, `t.Fatal()` will mark the test as failed, log the error, and then completely stop execution of the *current test* (or sub-test).

Typically you should call `t.Fatal()` in situations where it doesn't make sense to continue the current test — such as an error during a setup step, or where an unexpected error from a Go standard library function means you can't proceed with the test.

OK, save the file, then try running `go test` again with the verbose flag set. Like so:

```
$ go test -v ./cmd/web/
==== RUN  TestPing
--- PASS: TestPing (0.00s)
==== RUN  TestHumanDate
==== RUN  TestHumanDate/UTC
==== RUN  TestHumanDate/Empty
==== RUN  TestHumanDate/CET
--- PASS: TestHumanDate (0.00s)
    --- PASS: TestHumanDate/UTC (0.00s)
    --- PASS: TestHumanDate/Empty (0.00s)
    --- PASS: TestHumanDate/CET (0.00s)
PASS
ok      snippetbox.alexandedwards.net/cmd/web      0.003s
```

So this is looking good. We can see that our new `TestPing` test is being run and passing without any problems.

Testing middleware

It's also possible to use the same general pattern to unit test your middleware.

We'll demonstrate this by creating a new `TestCommonHeaders` test for the `commonHeaders()` middleware that we made [earlier in the book](#). As part of this test we want to check that:

- The `commonHeaders()` middleware sets all the expected headers on the HTTP response.
- The `commonHeaders()` middleware correctly calls the next handler in the chain.

First you'll need to create a `cmd/web/middleware_test.go` file to hold the test:

```
$ touch cmd/web/middleware_test.go
```

And then add the following code:

```
File: cmd/web/middleware_test.go

package main

import (
    "bytes"
    "io"
    "net/http"
    "net/http/httpptest"
    "testing"

    "snippetbox.alexedwards.net/internal/assert"
)

func TestCommonHeaders(t *testing.T) {
    // Initialize a new httpptest.ResponseRecorder and dummy http.Request.
    rr := httpptest.NewRecorder()

    r, err := http.NewRequest(http.MethodGet, "/", nil)
    if err != nil {
        t.Fatal(err)
    }

    // Create a mock HTTP handler that we can pass to our commonHeaders
    // middleware, which writes a 200 status code and an "OK" response body.
    next := http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
        w.Write([]byte("OK"))
    })

    // Pass the mock HTTP handler to our commonHeaders middleware. Because
    // commonHeaders *returns* a http.Handler we can call its ServeHTTP()
    // method, passing in the http.ResponseRecorder and dummy http.Request to
    // execute it.
    commonHeaders(next).ServeHTTP(rr, r)

    // Call the Result() method on the http.ResponseRecorder to get the results
    // of the test.
    rs := rr.Result()

    // Check that the middleware has correctly set the Content-Security-Policy
    // header on the response.
    expectedValue := "default-src 'self'; style-src 'self' fonts.googleapis.com; font-src fonts.gstatic.com"
    assert.Equal(t, rs.Header.Get("Content-Security-Policy"), expectedValue)

    // Check that the middleware has correctly set the Referrer-Policy
    // header on the response.
    expectedValue = "origin-when-cross-origin"
    assert.Equal(t, rs.Header.Get("Referrer-Policy"), expectedValue)

    // Check that the middleware has correctly set the X-Content-Type-Options
    // header on the response.
    expectedValue = "nosniff"
    assert.Equal(t, rs.Header.Get("X-Content-Type-Options"), expectedValue)

    // Check that the middleware has correctly set the X-Frame-Options header
    // on the response.
    expectedValue = "deny"
    assert.Equal(t, rs.Header.Get("X-Frame-Options"), expectedValue)

    // Check that the middleware has correctly set the X-XSS-Protection header
    // on the response
    expectedValue = "0"
```

```

assert.Equal(t, rs.Header.Get("X-XSS-Protection"), expectedValue)

// Check that the middleware has correctly set the Server header on the
// response.
expectedValue = "Go"
assert.Equal(t, rs.Header.Get("Server"), expectedValue)

// Check that the middleware has correctly called the next handler in line
// and the response status code and body are as expected.
assert.Equal(t, rs.StatusCode, http.StatusOK)

defer rs.Body.Close()
body, err := io.ReadAll(rs.Body)
if err != nil {
    t.Fatal(err)
}
body = bytes.TrimSpace(body)

assert.Equal(t, string(body), "OK")
}

```

If you run the tests now, you should see that the `TestCommonHeaders` test passes without any issues.

```

$ go test -v ./cmd/web/
==== RUN  TestPing
--- PASS: TestPing (0.00s)
==== RUN  TestCommonHeaders
--- PASS: TestCommonHeaders (0.00s)
==== RUN  TestHumanDate
==== RUN  TestHumanDate/UTC
==== RUN  TestHumanDate/Empty
==== RUN  TestHumanDate/CET
--- PASS: TestHumanDate (0.00s)
    --- PASS: TestHumanDate/UTC (0.00s)
    --- PASS: TestHumanDate/Empty (0.00s)
    --- PASS: TestHumanDate/CET (0.00s)
PASS
ok      snippetbox.alexedwards.net/cmd/web      0.003s

```

So, in summary, a quick and easy way to unit test your HTTP handlers and middleware is to simply call them using the `httptest.ResponseRecorder` type. You can then examine the status code, headers and response body of the recorded response to make sure that they are working as expected.

End-to-end testing

In the last chapter we talked through the general pattern for how to unit test your HTTP handlers in isolation.

But — most of the time — your HTTP handlers aren’t *actually used* in isolation. So in this chapter we’re going to explain how to run end-to-end tests on your web application that encompass your routing, middleware and handlers. In most cases, end-to-end testing should give you more confidence that your application is working correctly than unit testing in isolation.

To illustrate this, we’ll adapt our `TestPing` function so that it runs an end-to-end test on our code. Specifically, we want the test to ensure that a `GET /ping` request to our application calls the `ping` handler function and results in a `200 OK` status code and “OK” response body.

Essentially, we want to test that our application has a route like this:

Route pattern	Handler	Action
...
<code>GET /ping</code>	<code>ping</code>	Return a <code>200 OK</code> response

Using `httptest.Server`

The key to end-to-end testing our application is the `httptest.NewTLSserver()` function, which spins up a `httptest.Server` instance that we can make HTTPS requests to.

The whole pattern is a bit too complicated to explain upfront, so it’s probably best to demonstrate first by writing the code and then we’ll talk through the details afterwards.

With that in mind, head back to your `handlers_test.go` file and update the `TestPing` test so that it looks like this:

```
File: cmd/web/handlers_test.go
```

```
package main

import (
    "bytes"
    "io"
    "log/slog" // New import
    "net/http"
    "net/http/httpptest"
    "testing"

    "snippetbox.alexandedwards.net/internal/assert"
)

func TestPing(t *testing.T) {
    // Create a new instance of our application struct. For now, this just
    // contains a structured logger (which discards anything written to it).
    app := &application{
        logger: slog.New(slog.NewTextHandler(io.Discard, nil)),
    }

    // We then use the httpptest.NewTLSVerifier() function to create a new test
    // server, passing in the value returned by our app.routes() method as the
    // handler for the server. This starts up a HTTPS server which listens on a
    // randomly-chosen port of your local machine for the duration of the test.
    // Notice that we defer a call to ts.Close() so that the server is shutdown
    // when the test finishes.
    ts := httpptest.NewTLSVerifier(app.routes())
    defer ts.Close()

    // The network address that the test server is listening on is contained in
    // the ts.URL field. We can use this along with the ts.Client().Get() method
    // to make a GET /ping request against the test server. This returns a
    // http.Response struct containing the response.
    rs, err := ts.Client().Get(ts.URL + "/ping")
    if err != nil {
        t.Fatal(err)
    }

    // We can then check the value of the response status code and body using
    // the same pattern as before.
    assert.Equal(t, rs.StatusCode, http.StatusOK)

    defer rs.Body.Close()
    body, err := io.ReadAll(rs.Body)
    if err != nil {
        t.Fatal(err)
    }
    body = bytes.TrimSpace(body)

    assert.Equal(t, string(body), "OK")
}
```

There are a few things about this code to point out and discuss.

- When we call `httpptest.NewTLSVerifier()` to initialize the test server we need to pass in a `http.Handler` as the parameter — and this handler is called each time the test server receives a HTTPS request. In our case, we've passed in the return value from our `app.routes()` method, meaning that a request to the test server will use *all our real*

application routes, middleware and handlers.

This is a big upside of the work that we did [earlier in the book](#) to isolate all our application routing in the `app.routes()` method.

- If you’re testing a HTTP (not HTTPS) server you should use the `httpstest.NewServer()` function to create the test server instead.
- The `ts.Client()` method returns the *test server client* — which has the type `http.Client` — and we should always use this client to send requests to the test server. It’s possible to configure the client to tweak its behavior, and we’ll explain how to do that at the end of this chapter.
- You might be wondering why we have set the `logger` field of our `application` struct, but none of the other fields. The reason for this is that the logger is needed by the `logRequest` and `recoverPanic` middlewares, which are used by our application on every route. Trying to run this test without setting these the two dependencies will result in a panic.

Anyway, let’s try out the new test:

```
$ go test ./cmd/web/
--- FAIL: TestPing (0.00s)
    handlers_test.go:41: got 404; want 200
    handlers_test.go:51: got: Not Found; want: OK
FAIL
FAIL    snippetbox.alexandedwards.net/cmd/web      0.007s
FAIL
```

If you’re following along, you should get a failure at this point.

We can see from the test output that the response from our `GET /ping` request has a `404` status code, rather than the `200` we expected. And that’s because we haven’t actually registered a `GET /ping` route with our router yet.

Let’s fix that now:

```

File: cmd/web/routes.go

package main

...

func (app *application) routes() http.Handler {
    mux := http.NewServeMux()

    mux.Handle("GET /static/", http.FileServerFS(ui.Files))

    // Add a new GET /ping route.
    mux.HandleFunc("GET /ping", ping)

    dynamic := alice.New(app.sessionManager.LoadAndSave, noSurf, app.authenticate)

    mux.Handle("GET /{$}", dynamic.ThenFunc(app.home))
    mux.Handle("GET /snippet/view/{id}", dynamic.ThenFunc(app.snippetView))
    mux.Handle("GET /user/signup", dynamic.ThenFunc(app.userSignup))
    mux.Handle("POST /user/signup", dynamic.ThenFunc(app.userSignupPost))
    mux.Handle("GET /user/login", dynamic.ThenFunc(app.userLogin))
    mux.Handle("POST /user/login", dynamic.ThenFunc(app.userLoginPost))

    protected := dynamic.Append(app.requireAuthentication)

    mux.Handle("GET /snippet/create", protected.ThenFunc(app.snippetCreate))
    mux.Handle("POST /snippet/create", protected.ThenFunc(app.snippetCreatePost))
    mux.Handle("POST /user/logout", protected.ThenFunc(app.userLogoutPost))

    standard := alice.New(app.recoverPanic, app.logRequest, commonHeaders)
    return standard.Then(mux)
}

```

And if you run the tests again everything should now pass.

```

$ go test ./cmd/web/
ok      snippetbox.alexedwards.net/cmd/web    0.008s

```

Using test helpers

Our `TestPing` test is now working nicely. But there's a good opportunity to break out some of this code into helper functions, which we can reuse as we add more end-to-end tests to our project.

There's no hard-and-fast rules about where to put helper methods for tests. If a helper is only used in a specific `*_test.go` file, then it probably makes sense to include it inline in that file alongside your tests. At the other end of the spectrum, if you are going to use a helper in tests across multiple packages, then you might want to put it in a reusable package called `internal/testutils` (or similar) which can be imported by your test files.

In our case, the helpers will be used for testing code throughout our `cmd/web` package but

nowhere else, so it seems reasonable to put them in a new [cmd/web/testutils_test.go](#) file.

If you're following along, please go ahead and create this now...

```
$ touch cmd/web/testutils_test.go
```

And then add the following code:

```

File: cmd/web/testutils_test.go

package main

import (
    "bytes"
    "io"
    "log/slog"
    "net/http"
    "net/http/httpptest"
    "testing"
)

// Create a newTestApplication helper which returns an instance of our
// application struct containing mocked dependencies.
func newTestApplication(t *testing.T) *application {
    return &application{
        logger: slog.New(slog.NewTextHandler(io.Discard, nil)),
    }
}

// Define a custom testServer type which embeds a httpptest.Server instance.
type testServer struct {
    *httpptest.Server
}

// Create a newTestServer helper which initializes and returns a new instance
// of our custom testServer type.
func newTestServer(t *testing.T, h http.Handler) *testServer {
    ts := httpertest.NewTLSVerifier(h)
    return &testServer{ts}
}

// Implement a get() method on our custom testServer type. This makes a GET
// request to a given url path using the test server client, and returns the
// response status code, headers and body.
func (ts *testServer) get(t *testing.T, urlPath string) (int, http.Header, string) {
    rs, err := ts.Client().Get(ts.URL + urlPath)
    if err != nil {
        t.Fatal(err)
    }

    defer rs.Body.Close()
    body, err := io.ReadAll(rs.Body)
    if err != nil {
        t.Fatal(err)
    }
    body = bytes.TrimSpace(body)

    return rs.StatusCode, rs.Header, string(body)
}

```

Essentially, this is just a generalization of the code that we've already written in this chapter to spin up a test server and make a `GET` request against it.

Let's head back to our `TestPing` handler and put these new helpers to work:

```
File: cmd/web/handlers_test.go
```

```
package main

import (
    "net/http"
    "testing"

    "snippetbox.alexedwards.net/internal/assert"
)

func TestPing(t *testing.T) {
    app := newTestApplication(t)

    ts := newTestServer(t, app.routes())
    defer ts.Close()

    code, _, body := ts.get(t, "/ping")

    assert.Equal(t, code, http.StatusOK)
    assert.Equal(t, body, "OK")
}
```

And, again, if you run the tests again everything should still pass.

```
$ go test ./cmd/web/
ok      snippetbox.alexedwards.net/cmd/web      0.013s
```

This is shaping up nicely now. We have a neat pattern in place for spinning up a test server and making requests to it, encompassing our routing, middleware and handlers in an end-to-end test. We've also broken apart some of the code into helpers, which will make writing future tests quicker and easier.

Cookies and redirections

So far in this chapter we've been using the default *test server client* settings. But there are a couple of changes I'd like to make so that it's better suited to testing our web application. Specifically:

- We want the client to automatically store any cookies sent in a HTTPS response, so that we can include them (if appropriate) in any subsequent requests back to the test server. This will come in handy later in the book when we need cookies to be supported across multiple requests in order to test our anti-CSRF measures.
- We don't want the client to automatically follow redirects. Instead we want it to return the first HTTPS response sent by our server so that we can test the response *for that specific request*.

To make these changes, let's go back to the `testutils_test.go` file we just created and update the `newTestServer()` function like so:

```
File: cmd/web/testutils_test.go

package main

import (
    "bytes"
    "io"
    "log/slog"
    "net/http"
    "net/http/cookiejar" // New import
    "net/http/httpptest"
    "testing"
)

...

func newTestServer(t *testing.T, h http.Handler) *testServer {
    // Initialize the test server as normal.
    ts := httpptest.NewTLSServer(h)

    // Initialize a new cookie jar.
    jar, err := cookiejar.New(nil)
    if err != nil {
        t.Fatal(err)
    }

    // Add the cookie jar to the test server client. Any response cookies will
    // now be stored and sent with subsequent requests when using this client.
    ts.Client().Jar = jar

    // Disable redirect-following for the test server client by setting a custom
    // CheckRedirect function. This function will be called whenever a 3xx
    // response is received by the client, and by always returning a
    // http.ErrUseLastResponse error it forces the client to immediately return
    // the received response.
    ts.Client().CheckRedirect = func(req *http.Request, via []*http.Request) error {
        return http.ErrUseLastResponse
    }

    return &testServer{ts}
}

...
```

Customizing how tests run

Before we continue adding more tests to our application, I want to take a quick break and talk about a few of the useful flags and options that are available to customize how your tests run.

Controlling which tests are run

So far in this book we've been running the tests in a specific package (the `cmd/web` package) like so:

```
$ go test ./cmd/web
```

But it's also possible to run *all* the tests in your current project by using the `./...` wildcard pattern. In our case, we can use it to run all tests in our project like this:

```
$ go test ./...
ok      snippetbox.alexewards.net/cmd/web      0.007s
?       snippetbox.alexewards.net/internal/models [no test files]
?       snippetbox.alexewards.net/internal/validator [no test files]
?       snippetbox.alexewards.net/ui      [no test files]
```

Or going in the other direction, it's possible to only run specific tests by using the `-run` flag. This allows you to specify a regular expression — and only tests with a name that matches the regular expression will be run.

For example, we could opt to run only the `TestPing` test as follows:

```
$ go test -v -run="^TestPing$" ./cmd/web/
==== RUN  TestPing
--- PASS: TestPing (0.00s)
PASS
ok      snippetbox.alexewards.net/cmd/web      0.008s
```

And you can even use the `-run` flag to limit testing to some specific sub-tests using the format `{test regexp}/{sub-test regexp}`. For example to run the `UTC` sub-test of our `TestHumanDate` test we could do this:

```
$ go test -v -run="^TestHumanDate$|^UTC$" ./cmd/web
==== RUN    TestHumanDate
==== RUN    TestHumanDate/UTC
--- PASS: TestHumanDate (0.00s)
--- PASS: TestHumanDate/UTC (0.00s)
PASS
ok      snippetbox.alexedwards.net/cmd/web    0.003s
```

In contrast, you can prevent specific tests from running by using the `-skip` flag. Like the `-run` flag we just looked at, this allows you to specify a regular expression and any tests with a name that matches the regular expression *won't* be run. For example, to skip the `TestHumanDate` test:

```
$ go test -v -skip="^TestHumanDate$" ./cmd/web/
==== RUN    TestPing
--- PASS: TestPing (0.00s)
==== RUN    TestCommonHeaders
--- PASS: TestCommonHeaders (0.00s)
PASS
ok      snippetbox.alexedwards.net/cmd/web    0.006s
```

Test caching

You've perhaps noticed by now that if you run exactly the same test twice — without making any changes to the package that you're testing — then a *cached* version of the test result is shown (indicated by the `(cached)` annotation next to the package name).

```
$ go test ./cmd/web
ok      snippetbox.alexedwards.net/cmd/web    (cached)
```

In most cases, the caching of test results is really useful (especially for large codebases) because it helps reduce the total test runtime. But if you want force your tests to run in full (and avoid the cache) you can use the `-count=1` flag:

```
$ go test -count=1 ./cmd/web
```

Note: The `count` flag is used to tell `go test` *how many times you want to execute each test*. It's a *non-cacheable* flag, which means that any time you use it `go test` will neither read or write the test results to the cache. So using `count=1` is a bit of a trick to avoid the cache without otherwise affecting how your tests run.

Alternatively, you can clear cached results for all tests with the `go clean` command:

```
$ go clean -testcache
```

Fast failure

As I mentioned briefly a couple of chapters ago, when you use the `t.Errorf()` function to mark a test as failed, it doesn't cause `go test` to immediately exit. All your other tests (and sub-tests) will continue to be run after a failure.

If you would prefer to terminate the tests immediately after the first failure you can use the `-failfast` flag:

```
$ go test -failfast ./cmd/web
```

It's important to note that the `-failfast` flag only stops tests *in the package that had the failure*. If you are running tests in multiple packages (for example by using `go test ./...`), then the tests in the other packages [will continue to run](#).

Parallel testing

By default, the `go test` command executes all tests in a serial manner, one after another. When you have a small number of tests (like we do) and the runtime is very fast, this is absolutely fine.

But if you have hundreds or thousands of tests the total run time can start adding up to something more meaningful. And in that scenario, you may save yourself some time by running the tests in parallel.

You can indicate that it's OK for a test to be run concurrently alongside other tests by calling the `t.Parallel()` function at the start of the test. For example:

```
func TestPing(t *testing.T) {
    t.Parallel()

    ...
}
```

It's important to note here that:

- Tests marked using `t.Parallel()` will be run in parallel with — *and only with* — other parallel tests.
- By default, the maximum number of tests that will be run simultaneously is the current value of `GOMAXPROCS`. You can override this by setting a specific value via the `-parallel` flag. For example:

```
$ go test -parallel=4 ./...
```

- Not all tests are suitable to be run in parallel. For example, if you have an integration test which requires a database table to be in a specific known state, then you wouldn't want to run it in parallel with other tests that manipulate the same database table.

Enabling the race detector

The `go test` command includes a `-race` flag which enables Go's `race` detector when running tests.

If the code you're testing leverages concurrency, or you're running tests in parallel, enabling this can be a good idea to help to flag up race conditions that exist in your application. You can use it like so:

```
$ go test -race ./cmd/web/
```

You should be aware that the race detector is limited in its usefulness... it's just a tool that flags data races if and when they are identified at runtime during testing. It doesn't carry out static analysis of your codebase, and a clear run doesn't ensure that your code is free of race conditions.

Enabling the race detector will also increase the overall running time of your tests. So if you're running tests very frequently as part of a TDD workflow, you may prefer to use the `-race` flag during pre-commit test runs only.

Mocking dependencies

Now that we've explained some general patterns for testing your web application, in this chapter we're going to get a bit more serious and write some tests for our `GET /snippet/view/{id}` route.

But first, let's talk about dependencies.

Throughout this project we've injected dependencies into our handlers via the `application` struct, which currently looks like this:

```
type application struct {
    logger      *slog.Logger
    snippets    *models.SnippetModel
    users       *models.UserModel
    templateCache map[string]*template.Template
    formDecoder  *form.Decoder
    sessionManager *scs.SessionManager
}
```

When testing, it sometimes makes sense to mock these dependencies instead of using *exactly* the same ones that you do in your production application.

For example, in the previous chapter we *mocked* the `logger` dependency with a logger that writes messages to `io.Discard`, instead of the `os.Stdout` and stream like we do in our production application:

```
func newTestApplication(t *testing.T) *application {
    return &application{
        logger: slog.New(slog.NewTextHandler(io.Discard, nil)),
    }
}
```

The reason for mocking this and writing to `io.Discard` is to avoid clogging up our test output with unnecessary log messages when we run `go test -v` (with verbose mode enabled).

Note: Depending on your background and programming experience, you might not consider this logger to a *mock*. You might call it a *fake*, *stub* or something else entirely. But the name doesn't really matter — and different people [call them different things](#). What's important is that we're using something which *exposes the same interface as a production object* for the purpose of testing.

The other two dependencies that it makes sense for us to mock are the `models.SnippetModel` and `models.UserModel` database models. By creating mocks of these it's possible for us to test the behavior of our handlers *without* needing to setup an entire test instance of the MySQL database.

Mocking the database models

If you're following along, create a new `internal/models/mocks` package containing `snippets.go` and `user.go` files to hold the database model mocks, like so:

```
$ mkdir internal/models/mocks
$ touch internal/models/mocks/snippets.go
$ touch internal/models/mocks/users.go
```

Let's begin by creating a mock of our `models.SnippetModel`. To do this, we're going to create a simple struct which implements the same methods as our production `models.SnippetModel`, but have the methods return some fixed dummy data instead.

File: internal/models/mocks/snippets.go

```
package mocks

import (
    "time"

    "snippetbox.alexedwards.net/internal/models"
)

var mockSnippet = models.Snippet{
    ID:      1,
    Title:   "An old silent pond",
    Content: "An old silent pond...",
    Created: time.Now(),
    Expires: time.Now(),
}

type SnippetModel struct{}

func (m *SnippetModel) Insert(title string, content string, expires int) (int, error) {
    return 2, nil
}

func (m *SnippetModel) Get(id int) (models.Snippet, error) {
    switch id {
    case 1:
        return mockSnippet, nil
    default:
        return models.Snippet{}, models.ErrNoRecord
    }
}

func (m *SnippetModel) Latest() ([]models.Snippet, error) {
    return []models.Snippet{mockSnippet}, nil
}
```

And let's do the same for our `models.UserModel`, like so:

File: internal/models/mocks/users.go

```
package mocks

import (
    "snippetbox.alexedwards.net/internal/models"
)

type UserModel struct{}

func (m *UserModel) Insert(name, email, password string) error {
    switch email {
    case "dupe@example.com":
        return models.ErrDuplicateEmail
    default:
        return nil
    }
}

func (m *UserModel) Authenticate(email, password string) (int, error) {
    if email == "alice@example.com" && password == "pa$$word" {
        return 1, nil
    }

    return 0, models.ErrInvalidCredentials
}

func (m *UserModel) Exists(id int) (bool, error) {
    switch id {
    case 1:
        return true, nil
    default:
        return false, nil
    }
}
```

Initializing the mocks

For the next step in our build, let's head back to the `testutils_test.go` file and update the `newTestApplication()` function so that it creates an `application` struct with all the necessary dependencies for testing.

```
File: cmd/web/testutils_test.go
```

```
package main

import (
    "bytes"
    "io"
    "log/slog"
    "net/http"
    "net/http/cookiejar"
    "net/http/httpptest"
    "testing"
    "time" // New import

    "snippetbox.alexewards.net/internal/models/mocks" // New import

    "github.com/alexewards/scs/v2" // New import
    "github.com/go-playground/form/v4" // New import
)

func newTestApplication(t *testing.T) *application {
    // Create an instance of the template cache.
    templateCache, err := newTemplateCache()
    if err != nil {
        t.Fatal(err)
    }

    // And a form decoder.
    formDecoder := form.NewDecoder()

    // And a session manager instance. Note that we use the same settings as
    // production, except that we *don't* set a Store for the session manager.
    // If no store is set, the SCS package will default to using a transient
    // in-memory store, which is ideal for testing purposes.
    sessionManager := scs.New()
    sessionManager.Lifetime = 12 * time.Hour
    sessionManager.Cookie.Secure = true

    return &application{
        logger:      slog.New(slog.NewTextHandler(io.Discard, nil)),
        snippets:    &mocks.SnippetModel{}, // Use the mock.
        users:       &mocks.UserModel{}, // Use the mock.
        templateCache: templateCache,
        formDecoder:   formDecoder,
        sessionManager: sessionManager,
    }
}

...
```

If you go ahead and try to run the tests now, it will fail to compile with the following errors:

```
$ go test ./cmd/web
# snippetbox.alexewards.net/cmd/web [snippetbox.alexewards.net/cmd/web.test]
cmd/web/testutils_test.go:40:19: cannot use &mocks.SnippetModel{} (value of type *mocks.SnippetModel) as type *models.Snippet
cmd/web/testutils_test.go:41:19: cannot use &mocks.UserModel{} (value of type *mocks.UserModel) as type *models.UserModel in
FAIL    snippetbox.alexewards.net/cmd/web [build failed]
FAIL
```

This is happening because our `application` struct is expecting pointers to `models.SnippetModel` and `models.UserModel` instances, but we are trying to use pointers to `mocks.SnippetModel` and `mocks.UserModel` instances instead.

The idiomatic fix for this is to change our `application` struct so that it uses interfaces which are satisfied by both our mock and production database models.

Tip: If you're not familiar with the idea of *interfaces* in Go, then there is an introduction in [this blog post](#) which I recommend reading.

To do this, let's head back to our `internal/models/snippets.go` file and create a new `SnippetModelInterface` interface type that *describes the methods that our actual SnippetModel struct has*.

```
File: internal/models/snippets.go

package models

import (
    "database/sql"
    "errors"
    "time"
)

type SnippetModelInterface interface {
    Insert(title string, content string, expires int) (int, error)
    Get(id int) (Snippet, error)
    Latest() ([]Snippet, error)
}

...
```

And let's also do the same thing for our `UserModel` struct too:

File: internal/models/users.go

```
package models

import (
    "database/sql"
    "errors"
    "strings"
    "time"

    "github.com/go-sql-driver/mysql"
    "golang.org/x/crypto/bcrypt"
)

type UserModelInterface interface {
    Insert(name, email, password string) error
    Authenticate(email, password string) (int, error)
    Exists(id int) (bool, error)
}

...
```

Now that we've defined those interface types, let's update our `application` struct to use them instead of the concrete `SnippetModel` and `UserModel` types. Like so:

File: cmd/web/main.go

```
package main

import (
    "crypto/tls"
    "database/sql"
    "flag"
    "html/template"
    "log/slog"
    "net/http"
    "os"
    "time"

    "snippetbox.alexedwards.net/internal/models"

    "github.com/alexedwards/scs/mysqlstore"
    "github.com/alexedwards/scs/v2"
    "github.com/go-playground/form/v4"
    _ "github.com/go-sql-driver/mysql"
)

type application struct {
    logger      *slog.Logger
    snippets    models.SnippetModelInterface // Use our new interface type.
    users       models.UserModelInterface     // Use our new interface type.
    templateCache map[string]*template.Template
    formDecoder   *form.Decoder
    sessionManager *scs.SessionManager
}

...
```

And if you try running the tests again now, everything should work correctly.

```
$ go test ./cmd/web/
ok      snippetbox.alexedwards.net/cmd/web      0.008s
```

Let's take a moment to pause and reflect on what we've just done.

We've updated the `application` struct so that instead of the `snippets` and `users` fields having the concrete types `*models.SnippetModel` and `*models.UserModel` they are *interfaces* instead.

So long as an type has the necessary methods to satisfy the interface, we can use them in our `application` struct. Both our 'real' database models (like `models.SnippetModel`) and mock database models (like `mocks.SnippetModel`) satisfy the interfaces, so we can now use them interchangeably.

Testing the snippetView handler

With that all now set up, let's get stuck into writing an end-to-end test for our `snippetView` handler which uses these mocked dependencies.

As part of this test, the code in our `snippetView` handler will call the `mock.SnippetModel.Get()` method. Just to remind you, this mocked model method returns a `models.ErrNoRecord` unless the snippet ID is `1` — when it will return the following mock snippet:

```
var mockSnippet = models.Snippet{
    ID:      1,
    Title:   "An old silent pond",
    Content: "An old silent pond...",
    Created: time.Now(),
    Expires: time.Now(),
}
```

So specifically, we want to test that:

1. For the request `GET /snippet/view/1` we receive a `200 OK` response with the relevant mocked snippet *contained in* the HTML response body.
2. For all other requests to `GET /snippet/view/*` we should receive a `404 Not Found` response.

For the first part here, we want to check that the request body *contains* some specific

content, rather than being exactly equal to it. Let's quickly add a new `StringContains()` function to our `assert` package to help with that:

```
File: internal/assert/assert.go

package assert

import (
    "strings" // New import
    "testing"
)

...

func StringContains(t *testing.T, actual, expectedSubstring string) {
    t.Helper()

    if !strings.Contains(actual, expectedSubstring) {
        t.Errorf("got: %q; expected to contain: %q", actual, expectedSubstring)
    }
}
```

And then open up the `cmd/web/handlers_test.go` file and create a new `TestSnippetView` test like so:

```
File: cmd/web/handlers_test.go

package main

...

func TestSnippetView(t *testing.T) {
    // Create a new instance of our application struct which uses the mocked
    // dependencies.
    app := newTestApplication(t)

    // Establish a new test server for running end-to-end tests.
    ts := newTestServer(t, app.routes())
    defer ts.Close()

    // Set up some table-driven tests to check the responses sent by our
    // application for different URLs.
    tests := []struct {
        name      string
        urlPath  string
        wantCode int
        wantBody string
    }{
        {
            name:      "Valid ID",
            urlPath:  "/snippet/view/1",
            wantCode: http.StatusOK,
            wantBody: "An old silent pond...",
        },
        {
            name:      "Non-existent ID",
            urlPath:  "/snippet/view/2",
            wantCode: http.StatusNotFound,
        },
    },
}
```

```
        },
        {
            name:      "Negative ID",
            urlPath:   "/snippet/view/-1",
            wantCode:  http.StatusNotFound,
        },
        {
            name:      "Decimal ID",
            urlPath:   "/snippet/view/1.23",
            wantCode:  http.StatusNotFound,
        },
        {
            name:      "String ID",
            urlPath:   "/snippet/view/foo",
            wantCode:  http.StatusNotFound,
        },
        {
            name:      "Empty ID",
            urlPath:   "/snippet/view/",
            wantCode:  http.StatusNotFound,
        },
    },
}

for _, tt := range tests {
    t.Run(tt.name, func(t *testing.T) {
        code, _, body := ts.get(t, tt.urlPath)

        assert.Equal(t, code, tt.wantCode)

        if tt.wantBody != "" {
            assert.StringContains(t, body, tt.wantBody)
        }
    })
}
}
```

If you run the tests again with the `-v` flag enabled, you should now see the new, passing, `TestSnippetView` sub-tests in the output:

```
$ go test -v ./cmd/web/
 === RUN   TestPing
--- PASS: TestPing (0.00s)
 === RUN   TestSnippetView
 === RUN   TestSnippetView/Valid_ID
 === RUN   TestSnippetView/Non-existent_ID
 === RUN   TestSnippetView/Negative_ID
 === RUN   TestSnippetView/Decimal_ID
 === RUN   TestSnippetView/String_ID
 === RUN   TestSnippetView/Empty_ID
--- PASS: TestSnippetView (0.01s)
    --- PASS: TestSnippetView/Valid_ID (0.00s)
    --- PASS: TestSnippetView/Non-existent_ID (0.00s)
    --- PASS: TestSnippetView/Negative_ID (0.00s)
    --- PASS: TestSnippetView/Decimal_ID (0.00s)
    --- PASS: TestSnippetView/String_ID (0.00s)
    --- PASS: TestSnippetView/Empty_ID (0.00s)
 === RUN   TestCommonHeaders
--- PASS: TestCommonHeaders (0.00s)
 === RUN   TestHumanDate
 === RUN   TestHumanDate/UTC
 === RUN   TestHumanDate/Empty
 === RUN   TestHumanDate/CET
--- PASS: TestHumanDate (0.00s)
    --- PASS: TestHumanDate/UTC (0.00s)
    --- PASS: TestHumanDate/Empty (0.00s)
    --- PASS: TestHumanDate/CET (0.00s)
PASS
ok      snippetbox.alexedwards.net/cmd/web      0.015s
```

As an aside, notice how the names of the sub-tests have been canonicalized? Go automatically replaces any spaces in the sub-test name with an underscore (and any non-printable characters will also be escaped) in the test output.

Testing HTML forms

In this chapter we're going to add an end-to-end test for the `POST /user/signup` route, which is handled by our `userSignupPost` handler.

Testing this route is made a bit more complicated by the anti-CSRF check that our application does. Any request that we make to `POST /user/signup` will always receive a `400 Bad Request` response *unless* the request contains a valid CSRF token and cookie. To get around this we need to simulate the workflow of a real-life user as part of our test, like so:

1. Make a `GET /user/signup` request. This will return a response which contains a CSRF cookie in the response headers and the CSRF token for the signup page in the response body.
2. Extract the CSRF token from the HTML response body.
3. Make a `POST /user/signup` request, using the same `http.Client` that we used in step 1 (so it automatically passes the CSRF cookie with the `POST` request) and including the CSRF token alongside the other `POST` data that we want to test.

Let's begin by adding a new helper function to our `cmd/web/testutils_test.go` file for extracting the CSRF token (if one exists) from an HTML response body:

```
File: cmd/web/testutils_test.go
```

```
package main

import (
    "bytes"
    "html" // New import
    "io"
    "log/slog"
    "net/http"
    "net/http/cookiejar"
    "net/http/httptest"
    "regexp" // New import
    "testing"
    "time"

    "snippetbox.alexedwards.net/internal/models/mocks"

    "github.com/alexedwards/scs/v2"
    "github.com/go-playground/form/v4"
)

// Define a regular expression which captures the CSRF token value from the
// HTML for our user signup page.
var csrfTokenRX = regexp.MustCompile(`<input type='hidden' name='csrf_token' value='(.+)'>`)

func extractCSRFToken(t *testing.T, body string) string {
    // Use the FindStringSubmatch method to extract the token from the HTML body.
    // Note that this returns an array with the entire matched pattern in the
    // first position, and the values of any captured data in the subsequent
    // positions.
    matches := csrfTokenRX.FindStringSubmatch(body)
    if len(matches) < 2 {
        t.Fatal("no csrf token found in body")
    }

    return html.UnescapeString(string(matches[1]))
}

...
```

Note: You might be wondering why we are using the `html.UnescapeString()` function before returning the CSRF token. The reason for this is because Go's `html/template` package automatically escapes all dynamically rendered data... including our CSRF token. Because the CSRF token is a base64 encoded string it will potentially include the `+` character, and this will be escaped to `+`. So after extracting the token from the HTML we need to run it through `html.UnescapeString()` to get the original token value.

Now that's in place, let's go back to our `cmd/web/handlers_test.go` file and create a new `TestUserSignup` test.

To start with, we'll make this perform a `GET /user/signup` request and then extract and print out the CSRF token from the HTML response body. Like so:

```

File: cmd/web/handlers_test.go

package main

...

func TestUserSignup(t *testing.T) {
    // Create the application struct containing our mocked dependencies and set
    // up the test server for running an end-to-end test.
    app := newTestApplication(t)
    ts := newTestServer(t, app.routes())
    defer ts.Close()

    // Make a GET /user/signup request and then extract the CSRF token from the
    // response body.
    _, _, body := ts.get(t, "/user/signup")
    csrfToken := extractCSRFToken(t, body)

    // Log the CSRF token value in our test output using the t.Logf() function.
    // The t.Logf() function works in the same way as fmt.Printf(), but writes
    // the provided message to the test output.
    t.Logf("CSRF token is: %q", csrfToken)
}

```

Importantly, you must run tests using the `-v` flag (to enable verbose output) in order to see any output from the `t.Logf()` function.

Let's go ahead and do that now:

```

$ go test -v -run="TestUserSignup" ./cmd/web/
== RUN  TestUserSignup
    handlers_test.go:81: CSRF token is: "C92tcpQpL1n6aIUaF8XAonwy+YjcVnyaAa0vfkdl6vJqoNSbgaTtdBRC61pFMoGP2ojV+sZ1d0SUikah3mfR
--- PASS: TestUserSignup (0.01s)
PASS
ok      snippetbox.alexandedwards.net/cmd/web      0.010s

```

OK, that looks like it's working. The test is running without any problems and printing out the CSRF token that we've extracted from the response body HTML.

Note: If you run this test for a second time immediately afterwards, without changing anything in the `cmd/web` package, you'll get the same CSRF token in the test output because the test results have been cached.

Testing post requests

Now let's head back to our `cmd/web/testutils_test.go` file and create a new `postForm()` method on our `testServer` type, which we can use to send a `POST` request to our test server

with specific form data in the request body.

Go ahead and add the following code (which follows the same general pattern that we used for the `get()` method earlier in the book):

```
File: cmd/web/testutils_test.go

package main

import (
    "bytes"
    "html"
    "io"
    "log/slog"
    "net/http"
    "net/http/cookiejar"
    "net/http/httpptest"
    "net/url" // New import
    "regexp"
    "testing"
    "time"

    "snippetbox.alexedwards.net/internal/models/mocks"

    "github.com/alexedwards/scs/v2"
    "github.com/go-playground/form/v4"
)

...

// Create a postForm method for sending POST requests to the test server. The
// final parameter to this method is a url.Values object which can contain any
// form data that you want to send in the request body.
func (ts *testServer) postForm(t *testing.T, urlPath string, form url.Values) (int, http.Header, string) {
    rs, err := ts.Client().PostForm(ts.URL+urlPath, form)
    if err != nil {
        t.Fatal(err)
    }

    // Read the response body from the test server.
    defer rs.Body.Close()
    body, err := io.ReadAll(rs.Body)
    if err != nil {
        t.Fatal(err)
    }
    body = bytes.TrimSpace(body)

    // Return the response status, headers and body.
    return rs.StatusCode, rs.Header, string(body)
}
```

And now, at last, we're ready to add some table-driven sub-tests to test the behavior of our application's `POST /user/signup` route. Specifically, we want to test that:

- A valid signup results in a `303 See Other` response.
- A form submission without a valid CSRF token results in a `400 Bad Request` response.
- An invalid form submission results in a `422 Unprocessable Entity` response and the

signup form is redisplayed. This should happen when:

- The name, email or password fields are empty.
- The email is not in a valid format.
- The password is less than 8 characters long.
- The email address is already in use.

Go ahead and update the `TestUserSignup` function to carry out these tests like so:

```
File: cmd/web/handlers_test.go

package main

import (
    "net/http"
    "net/url" // New import
    "testing"

    "snippetbox.alexandedwards.net/internal/assert"
)

...

func TestUserSignup(t *testing.T) {
    app := newTestApplication(t)
    ts := newTestServer(t, app.routes())
    defer ts.Close()

    _, _, body := ts.get(t, "/user/signup")
    validCSRFToken := extractCSRFToken(t, body)

    const (
        validName      = "Bob"
        validPassword = "validPa$$word"
        validEmail    = "bob@example.com"
        formTag       = "<form action='/user/signup' method='POST' novalidate>"
    )

    tests := []struct {
        name          string
        userName      string
        userEmail     string
        userPassword  string
        csrfToken     string
        wantCode      int
        wantFormTag   string
    }{
        {
            name:         "Valid submission",
            userName:    validName,
            userEmail:   validEmail,
            userPassword: validPassword,
            csrfToken:   validCSRFToken,
            wantCode:    http.StatusSeeOther,
        },
        {
            name:         "Invalid CSRF Token",
            userName:    validName,
            userEmail:   validEmail,
            userPassword: validPassword
        }
    }
}
```

```

        userPassword: validPassword,
        csrfToken:    "wrongToken",
        wantCode:     http.StatusBadRequest,
    },
    {
        name:      "Empty name",
        userName:  "",
        userEmail: validEmail,
        userPassword: validPassword,
        csrfToken:  validCSRFToken,
        wantCode:   http.StatusUnprocessableEntity,
        wantFormTag: formTag,
    },
    {
        name:      "Empty email",
        userName:  validName,
        userEmail: "",
        userPassword: validPassword,
        csrfToken:  validCSRFToken,
        wantCode:   http.StatusUnprocessableEntity,
        wantFormTag: formTag,
    },
    {
        name:      "Empty password",
        userName:  validName,
        userEmail: validEmail,
        userPassword: "",
        csrfToken:  validCSRFToken,
        wantCode:   http.StatusUnprocessableEntity,
        wantFormTag: formTag,
    },
    {
        name:      "Invalid email",
        userName:  validName,
        userEmail: "bob@example.",
        userPassword: validPassword,
        csrfToken:  validCSRFToken,
        wantCode:   http.StatusUnprocessableEntity,
        wantFormTag: formTag,
    },
    {
        name:      "Short password",
        userName:  validName,
        userEmail: validEmail,
        userPassword: "pa$$",
        csrfToken:  validCSRFToken,
        wantCode:   http.StatusUnprocessableEntity,
        wantFormTag: formTag,
    },
    {
        name:      "Duplicate email",
        userName:  validName,
        userEmail: "dupe@example.com",
        userPassword: validPassword,
        csrfToken:  validCSRFToken,
        wantCode:   http.StatusUnprocessableEntity,
        wantFormTag: formTag,
    },
},
}

for _, tt := range tests {
    t.Run(tt.name, func(t *testing.T) {
        form := url.Values{}
        form.Add("name", tt.userName)
        form.Add("email", tt.userEmail)
        form.Add("password", tt.userPassword)
        form.Add("csrf_token", tt.csrfToken)

```

```
        code, _, body := ts.postForm(t, "/user/signup", form)

        assert.Equal(t, code, tt.wantCode)

        if tt.wantFormTag != "" {
            assert.StringContains(t, body, tt.wantFormTag)
        }
    })
}
}
```

If you run the test, you should see that all the sub-tests run and pass successfully – similar to this:

```
$ go test -v -run="TestUserSignup" ./cmd/web/
==== RUN TestUserSignup
==== RUN TestUserSignup/Valid_submission
==== RUN TestUserSignup/Invalid_CSRF_Token
==== RUN TestUserSignup/Empty_name
==== RUN TestUserSignup/Empty_email
==== RUN TestUserSignup/Empty_password
==== RUN TestUserSignup/Invalid_email
==== RUN TestUserSignup/Short_password
==== RUN TestUserSignup/Long_password
==== RUN TestUserSignup/Duplicate_email
--- PASS: TestUserSignup (0.01s)
--- PASS: TestUserSignup/Valid_submission (0.00s)
--- PASS: TestUserSignup/Invalid_CSRF_Token (0.00s)
--- PASS: TestUserSignup/Empty_name (0.00s)
--- PASS: TestUserSignup/Empty_email (0.00s)
--- PASS: TestUserSignup/Empty_password (0.00s)
--- PASS: TestUserSignup/Invalid_email (0.00s)
--- PASS: TestUserSignup/Short_password (0.00s)
--- PASS: TestUserSignup/Long_password (0.00s)
--- PASS: TestUserSignup/Duplicate_email (0.00s)
PASS
ok      snippetbox.alexedwards.net/cmd/web      0.016s
```

Integration testing

Running end-to-end tests with mocked dependencies is a good thing to do, but we could improve confidence in our application even more if we also verify that our real MySQL database models are working as expected.

To do this we can run integration tests against a test version our MySQL database, which *mimics our production database* but exists for testing purposes only.

As a demonstration, in this chapter we'll setup an integration test to ensure that our `models.UserModel.Exists()` method is working correctly.

Test database setup and teardown

The first step is to create the test version of our MySQL database.

If you're following along, connect to MySQL from your terminal window as the `root` user and execute the following SQL statements to create a new `test_snippetbox` database and `test_web` user:

```
CREATE DATABASE test_snippetbox CHARACTER SET utf8mb4 COLLATE utf8mb4_unicode_ci;
```

```
CREATE USER 'test_web'@'localhost';
GRANT CREATE, DROP, ALTER, INDEX, SELECT, INSERT, UPDATE, DELETE ON test_snippetbox.* TO 'test_web'@'localhost';
ALTER USER 'test_web'@'localhost' IDENTIFIED BY 'pass';
```

Once that's done, let's make two SQL scripts:

1. A setup script to create the database tables (so that they mimic our production database) and insert a known set of test data than we can work with in our tests.
2. A teardown script which drops the database tables and data.

The idea is that we'll call these scripts at the start and end of each integration test, so that the test database is fully reset each time. This helps ensure that any changes we make during one test are not 'leaking' and affecting the results of another test.

Let's go ahead and create these scripts in a new `internal/models/testdata` directory like

SO:

```
$ mkdir internal/models/testdata
$ touch internal/models/testdata/setup.sql
$ touch internal/models/testdata/teardown.sql
```

```
File: internal/models/testdata/setup.sql

CREATE TABLE snippets (
    id INTEGER NOT NULL PRIMARY KEY AUTO_INCREMENT,
    title VARCHAR(100) NOT NULL,
    content TEXT NOT NULL,
    created DATETIME NOT NULL,
    expires DATETIME NOT NULL
);

CREATE INDEX idx_snippets_created ON snippets(created);

CREATE TABLE users (
    id INTEGER NOT NULL PRIMARY KEY AUTO_INCREMENT,
    name VARCHAR(255) NOT NULL,
    email VARCHAR(255) NOT NULL,
    hashed_password CHAR(60) NOT NULL,
    created DATETIME NOT NULL
);

ALTER TABLE users ADD CONSTRAINT users_uc_email UNIQUE (email);

INSERT INTO users (name, email, hashed_password, created) VALUES (
    'Alice Jones',
    'alice@example.com',
    '$2a$12$NuTjWXm3KKntReFwyBVHyuf/to.HEwTy.eS206TNfkGfr6HzGJJSWG',
    '2022-01-01 09:18:24'
);
```

```
File: internal/models/testdata/teardown.sql

DROP TABLE users;

DROP TABLE snippets;
```

Note: The Go tool ignores any directories called `testdata`, so these scripts will be ignored when compiling your application. As an aside, it also ignores any directories or files which have names that begin with an `_` or `.` character too.

Alright, now that we've got the scripts in place, let's make a new file to hold some helper functions for our integration tests:

```
$ touch internal/models/testutils_test.go
```

In this file let's create a `newTestDB()` helper function which:

- Creates a new `*sql.DB` connection pool for the test database;
- Executes the `setup.sql` script to create the database tables and dummy data;
- Register a 'cleanup' function which executes the `teardown.sql` script and closes the connection pool.

```
File: internal/models/testutils_test.go
```

```
package models

import (
    "database/sql"
    "os"
    "testing"
)

func newTestDB(t *testing.T) *sql.DB {
    // Establish a sql.DB connection pool for our test database. Because our
    // setup and teardown scripts contains multiple SQL statements, we need
    // to use the "multiStatements=true" parameter in our DSN. This instructs
    // our MySQL database driver to support executing multiple SQL statements
    // in one db.Exec() call.
    db, err := sql.Open("mysql", "test_web:pass@test_snippetbox?parseTime=true&multiStatements=true")
    if err != nil {
        t.Fatal(err)
    }

    // Read the setup SQL script from the file and execute the statements, closing
    // the connection pool and calling t.Fatal() in the event of an error.
    script, err := os.ReadFile("./testdata/setup.sql")
    if err != nil {
        db.Close()
        t.Fatal(err)
    }
    _, err = db.Exec(string(script))
    if err != nil {
        db.Close()
        t.Fatal(err)
    }

    // Use t.Cleanup() to register a function *which will automatically be
    // called by Go when the current test (or sub-test) which calls newTestDB()
    // has finished*. In this function we read and execute the teardown script,
    // and close the database connection pool.
    t.Cleanup(func() {
        defer db.Close()

        script, err := os.ReadFile("./testdata/teardown.sql")
        if err != nil {
            t.Fatal(err)
        }
        _, err = db.Exec(string(script))
        if err != nil {
            t.Fatal(err)
        }
    })
}

// Return the database connection pool.
return db
}
```

The important thing to take away here is this:

Whenever we call this `newTestDB()` function inside a test (or sub-test) it will run the setup script against the test database. And when the test or sub-test finishes, the cleanup function will automatically be executed and the teardown script will be run.

Testing the UserModel.Exists method

Now that the preparatory work is done, we're ready to actually write our integration test for the `models.UserModel.Exists()` method.

We know that our `setup.sql` script creates a `users` table containing one record (which should have the user ID `1` and email address `alice@example.com`). So we want to test that:

- Calling `models.UserModel.Exists(1)` returns a `true` boolean value and a `nil` error value.
- Calling `models.UserModel.Exists()` with any other user ID returns a `false` boolean value and a `nil` error value.

Let's first head to our `internal/assert` package and create a new `NilError()` assertion, which we will use to check that an error value is `nil`. Like so:

```
File: internal/assert/assert.go

package assert

...

func NilError(t *testing.T, actual error) {
    t.Helper()

    if actual != nil {
        t.Errorf("got: %v; expected: nil", actual)
    }
}
```

Then let's follow the Go conventions and create a new `users_test.go` file for our test, directly alongside the code being tested:

```
$ touch internal/models/users_test.go
```

And add a `TestUserModelExists` test containing the following code:

File: internal/models/users_test.go

```
package models

import (
    "testing"

    "snippetbox.alexedwards.net/internal/assert"
)

func TestUserModelExists(t *testing.T) {
    // Set up a suite of table-driven tests and expected results.
    tests := []struct {
        name   string
        userID int
        want   bool
    }{
        {
            name:   "Valid ID",
            userID: 1,
            want:   true,
        },
        {
            name:   "Zero ID",
            userID: 0,
            want:   false,
        },
        {
            name:   "Non-existent ID",
            userID: 2,
            want:   false,
        },
    }

    for _, tt := range tests {
        t.Run(tt.name, func(t *testing.T) {
            // Call the newTestDB() helper function to get a connection pool to
            // our test database. Calling this here -- inside t.Run() -- means
            // that fresh database tables and data will be set up and torn down
            // for each sub-test.
            db := newTestDB(t)

            // Create a new instance of the UserModel.
            m := UserModel{db}

            // Call the UserModel.Exists() method and check that the return
            // value and error match the expected values for the sub-test.
            exists, err := m.Exists(tt.userID)

            assert.Equal(t, exists, tt.want)
            assert.NilError(t, err)
        })
    }
}
```

If you run this test, then everything should pass without any problems.

```
$ go test -v ./internal/models
==== RUN  TestUserModelExists
==== RUN  TestUserModelExists/Valid_ID
==== RUN  TestUserModelExists/Zero_ID
==== RUN  TestUserModelExists/Non-existent_ID
--- PASS: TestUserModelExists (1.02s)
    --- PASS: TestUserModelExists/Valid_ID (0.33s)
    --- PASS: TestUserModelExists/Zero_ID (0.29s)
    --- PASS: TestUserModelExists/Non-existent_ID (0.40s)
PASS
ok      snippetbox.alexewards.net/internal/models      1.023s
```

The last line in the test output here is worth a mention. The total runtime for this test (1.023 seconds in my case) is much longer than for our previous tests — all of which took a few milliseconds to run. This big increase in runtime is primarily due to the large number of database operations that we needed to make during the tests.

While 1 second is a totally acceptable time to wait for this test in isolation, if you’re running hundreds of different integration tests against your database you might end up routinely waiting minutes — rather than seconds — for your tests to finish.

Skipping long-running tests

When your tests take a long time, you might decide that you want to skip specific long-running tests under certain circumstances. For example, you might decide to only run your integration tests before committing a change, instead of more frequently during development.

A common and idiomatic way to skip long-running tests is to use the `testing.Short()` function to check for the presence of a `-short` flag in your `go test` command, and then call the `t.Skip()` method to skip the test if the flag is present.

Let’s quickly update `TestUserModelExists` to do this *before running its actual tests*, like so:

```
File: internal/models/users_test.go
```

```
package models

import (
    "testing"

    "snippetbox.alexedwards.net/internal/assert"
)

func TestUserModelExists(t *testing.T) {
    // Skip the test if the "-short" flag is provided when running the test.
    if testing.Short() {
        t.Skip("models: skipping integration test")
    }

    ...
}
```

And then you can try running all the tests for the project with the `-short` flag enabled. The output should look similar to this:

```

$ go test -v -short ./...
    === RUN   TestPing
--- PASS: TestPing (0.00s)
    === RUN   TestSnippetView
    === RUN   TestSnippetView/Valid_ID
    === RUN   TestSnippetView/Non-existent_ID
    === RUN   TestSnippetView/Negative_ID
    === RUN   TestSnippetView/Decimal_ID
    === RUN   TestSnippetView/String_ID
    === RUN   TestSnippetView/Empty_ID
--- PASS: TestSnippetView (0.01s)
    --- PASS: TestSnippetView/Valid_ID (0.00s)
    --- PASS: TestSnippetView/Non-existent_ID (0.00s)
    --- PASS: TestSnippetView/Negative_ID (0.00s)
    --- PASS: TestSnippetView/Decimal_ID (0.00s)
    --- PASS: TestSnippetView/String_ID (0.00s)
    --- PASS: TestSnippetView/Empty_ID (0.00s)
    === RUN   TestUserSignup
    === RUN   TestUserSignup/Valid_submission
    === RUN   TestUserSignup/Invalid_CSRF_Token
    === RUN   TestUserSignup/Empty_name
    === RUN   TestUserSignup/Empty_email
    === RUN   TestUserSignup/Empty_password
    === RUN   TestUserSignup/Invalid_email
    === RUN   TestUserSignup/Short_password
    === RUN   TestUserSignup/Long_password
    === RUN   TestUserSignup/Duplicate_email
--- PASS: TestUserSignup (0.01s)
    --- PASS: TestUserSignup/Valid_submission (0.00s)
    --- PASS: TestUserSignup/Invalid_CSRF_Token (0.00s)
    --- PASS: TestUserSignup/Empty_name (0.00s)
    --- PASS: TestUserSignup/Empty_email (0.00s)
    --- PASS: TestUserSignup/Empty_password (0.00s)
    --- PASS: TestUserSignup/Invalid_email (0.00s)
    --- PASS: TestUserSignup/Short_password (0.00s)
    --- PASS: TestUserSignup/Long_password (0.00s)
    --- PASS: TestUserSignup/Duplicate_email (0.00s)
    === RUN   TestCommonHeaders
--- PASS: TestCommonHeaders (0.00s)
    === RUN   TestHumanDate
    === RUN   TestHumanDate/UTC
    === RUN   TestHumanDate/Empty
    === RUN   TestHumanDate/CET
--- PASS: TestHumanDate (0.00s)
    --- PASS: TestHumanDate/UTC (0.00s)
    --- PASS: TestHumanDate/Empty (0.00s)
    --- PASS: TestHumanDate/CET (0.00s)
PASS
ok      snippetbox.alexedwards.net/cmd/web      0.023s
    === RUN   TestUserModelExists
        users_test.go:10: models: skipping integration test
--- SKIP: TestUserModelExists (0.00s)
PASS
ok      snippetbox.alexedwards.net/internal/models     0.003s
?       snippetbox.alexedwards.net/internal/models/mocks      [no test files]
?       snippetbox.alexedwards.net/internal/validator  [no test files]
?       snippetbox.alexedwards.net/ui      [no test files]

```

Notice the `SKIP` annotation in the output above? This confirms that Go skipped our `TestUserModelExists` test during this run.

If you like, feel free to run this again without the `-short` flag, and you should see that the

TestUserModelExists test is executed as normal.

Profiling test coverage

A great feature of the `go test` tool is the metrics and visualizations that it provides for test coverage.

Go ahead and try running the tests in our project using the `-cover` flag like so:

```
$ go test -cover ./...
?     snippetbox.alexewards.net/ui [no test files]
ok    snippetbox.alexewards.net/cmd/web 0.013s      coverage: 45.7% of statements
      snippetbox.alexewards.net/internal/models/mocks   coverage: 0.0% of statements
      snippetbox.alexewards.net/internal/validator       coverage: 0.0% of statements
      snippetbox.alexewards.net/internal/assert         coverage: 0.0% of statements
ok    snippetbox.alexewards.net/internal/models 0.128s  coverage: 11.3% of statements
```

From the results here we can see that 46.9% of the statements in our `cmd/web` package are executed during our tests, and for our `internal/models` package the figure is 11.3%.

Note: Your numbers may be slightly different depending on the exact version of the book you're reading, or any adaptations you've made to the code as you've been following along.

We can get a more detailed breakdown of test coverage *by method and function* by using the `-coverprofile` flag like so:

```
$ go test -coverprofile=/tmp/profile.out ./...
```

This will execute your tests as normal and — if all your tests pass — it will then write a coverage profile to a specific location. In the example above, we've instructed it to write the profile to `/tmp/profile.out`.

You can then view the coverage profile by using the `go tool cover` command like so:

```
$ go tool cover -func=/tmp/profile.out
snippetbox.alexewards.net/cmd/web/handlers.go:15:      home          0.0%
snippetbox.alexewards.net/cmd/web/handlers.go:31: snippetView    92.9%
snippetbox.alexewards.net/cmd/web/handlers.go:62: snippetCreate   0.0%
snippetbox.alexewards.net/cmd/web/handlers.go:88: snippetCreatePost 0.0%
snippetbox.alexewards.net/cmd/web/handlers.go:131: userSignup     100.0%
snippetbox.alexewards.net/cmd/web/handlers.go:137: userSignupPost  88.5%
snippetbox.alexewards.net/cmd/web/handlers.go:192: userLogin      0.0%
snippetbox.alexewards.net/cmd/web/handlers.go:198: userLoginPost   0.0%
snippetbox.alexewards.net/cmd/web/handlers.go:256: userLogoutPost  0.0%
snippetbox.alexewards.net/cmd/web/handlers.go:277: ping          100.0%
snippetbox.alexewards.net/cmd/web/helpers.go:17:  serverError    0.0%
snippetbox.alexewards.net/cmd/web/helpers.go:30: clientError     100.0%
snippetbox.alexewards.net/cmd/web/helpers.go:37: notFound       100.0%
snippetbox.alexewards.net/cmd/web/helpers.go:41: render         58.3%
snippetbox.alexewards.net/cmd/web/helpers.go:62: newTemplateData 100.0%
snippetbox.alexewards.net/cmd/web/helpers.go:73: decodePostForm  50.0%
snippetbox.alexewards.net/cmd/web/helpers.go:102: isAuthenticated 75.0%
snippetbox.alexewards.net/cmd/web/main.go:35: main          0.0%
snippetbox.alexewards.net/cmd/web/main.go:100: openDB        0.0%
snippetbox.alexewards.net/cmd/web/middleware.go:11: commonHeaders 100.0%
snippetbox.alexewards.net/cmd/web/middleware.go:26: logRequest    100.0%
snippetbox.alexewards.net/cmd/web/middleware.go:41: recoverPanic  66.7%
snippetbox.alexewards.net/cmd/web/middleware.go:61: requireAuthentication 16.7%
snippetbox.alexewards.net/cmd/web/middleware.go:83: noSurf        100.0%
snippetbox.alexewards.net/cmd/web/middleware.go:94: authenticate   38.5%
snippetbox.alexewards.net/cmd/web/routes.go:12: routes        100.0%
snippetbox.alexewards.net/cmd/web/templates.go:23: humanDate    100.0%
snippetbox.alexewards.net/cmd/web/templates.go:40: newTemplateCache 83.3%
snippetbox.alexewards.net/internal/models/snippets.go:31: Insert        0.0%
snippetbox.alexewards.net/internal/models/snippets.go:60: Get          0.0%
snippetbox.alexewards.net/internal/models/snippets.go:97: Latest       0.0%
snippetbox.alexewards.net/internal/models/users.go:34: Insert        0.0%
snippetbox.alexewards.net/internal/models/users.go:66: Authenticate  0.0%
snippetbox.alexewards.net/internal/models/users.go:98: Exists        100.0%
total:                                         (statements) 38.1%
```

An alternative and more visual way to view the coverage profile is to use the `-html` flag instead of `-func`.

```
$ go tool cover -html=/tmp/profile.out
```

This will open a browser window containing a navigable and highlighted representation of your code, similar to this:

```
File Edit View History Bookmarks Tools Help
web: Go Coverage Report + 
file:///tmp/cover980171880/coverage.html#file0
snippetbox.alexewards.net/cmd/web/handlers.go (35.7%) not tracked not covered covered
package main

import (
    "errors"
    "fmt"
    "net/http"
    "strconv"

    "snippetbox.alexewards.net/internal/models"
    "snippetbox.alexewards.net/internal/validator" // New import
    "github.com/julienschmidt/httprouter"
)

func (app *application) home(w http.ResponseWriter, r *http.Request) {
    // Because httprouter matches the "/" path exactly, we can now remove the
    // manual check of r.URL.Path != "/" from this handler.

    snippets, err := app.snippets.Latest()
    if err != nil {
        app.serverError(w, err)
        return
    }

    data := app.newTemplateData(r)
    data.Snippets = snippets

    app.render(w, http.StatusOK, "home.tmpl", data)
}

func (app *application) snippetView(w http.ResponseWriter, r *http.Request) {
    params := httprouter.ParamsFromContext(r.Context())

    id, err := strconv.Atoi(params.ByName("id"))
    if err != nil || id < 1 {
        app.NotFound(w)
        return
    }

    snippet, err := app.snippets.Get(id)
    if err != nil {
        if errors.Is(err, models.ErrNoRecord) {

```

The statements which get executed during your tests are colored green, and those that are not executed are colored red. This makes it easy to see exactly what code is currently covered by your tests (unless you have red-green color blindness).

You can take this a step further and use the `-covermode=count` option when running `go test` like so:

```
$ go test -covermode=count -coverprofile=/tmp/profile.out ....
$ go tool cover -html=/tmp/profile.out
```

Instead of just highlighting the statements in green and red, using `-covermode=count` makes the coverage profile record the exact *number of times* that each statement is executed during the tests.

When viewed in the browser, statements which are executed more frequently are then shown in a more saturated shade of green, similar to this:

The screenshot shows a web browser window displaying a Go coverage report. The title bar says "File Edit View History Bookmarks Tools Help" and the tab is "web: Go Coverage Report". The address bar shows "file:///tmp/cover1514237495/coverage.html#file0". The content area displays the source code for "cmd/web/handlers.go" and its coverage status. The code includes imports for errors, fmt, net/http, strconv, snippetbox, and httprouter. It defines two functions: "home" and "snippetView". The "home" function handles the root path and sets the response data. The "snippetView" function handles a specific ID and retrieves a snippet from the database. Coverage statistics are shown as a series of colored stars: red for "not tracked" and "no coverage", yellow for "low coverage", and green for "high coverage". The overall coverage is 35.7%.

```
File Edit View History Bookmarks Tools Help
web: Go Coverage Report × +
← → ⌂ file:///tmp/cover1514237495/coverage.html#file0
snippetbox.alexewards.net/cmd/web/handlers.go (35.7%) not tracked no coverage low coverage * * * * * * * * high coverage
package main

import (
    "errors"
    "fmt"
    "net/http"
    "strconv"

    "snippetbox.alexewards.net/internal/models"
    "snippetbox.alexewards.net/internal/validator" // New import
    "github.com/julienschmidt/httprouter"
)

func (app *application) home(w http.ResponseWriter, r *http.Request) {
    // Because httprouter matches the "/" path exactly, we can now remove the
    // manual check of r.URL.Path != "/" from this handler.

    snippets, err := app.snippets.Latest()
    if err != nil {
        app.serverError(w, err)
        return
    }

    data := app.newTemplateData(r)
    data.Snippets = snippets

    app.render(w, http.StatusOK, "home tmpl", data)
}

func (app *application) snippetView(w http.ResponseWriter, r *http.Request) {
    params := httprouter.ParamsFromContext(r.Context())

    id, err := strconv.Atoi(paramsByName("id"))
    if err != nil || id < 1 {
        app.notFound(w)
        return
    }

    snippet, err := app.snippets.Get(id)
    if err != nil {
        if errors.Is(err, models.ErrNoRecord) {

```

Note: If you're running some of your tests in parallel, you should use the `-covermode=atomic` flag (instead of `-covermode=count`) to ensure an accurate count.

Conclusion

Over the course of this book we've explicitly covered a lot of topics, including routing, templating, working with a database, authentication/authorization, using HTTPS, using Go's testing package and more.

But there have been some other, more tacit, lessons too. The patterns that we've used to implement features — and the way that our project code is organized and links together — is something that you should be able to take and apply in your future work.

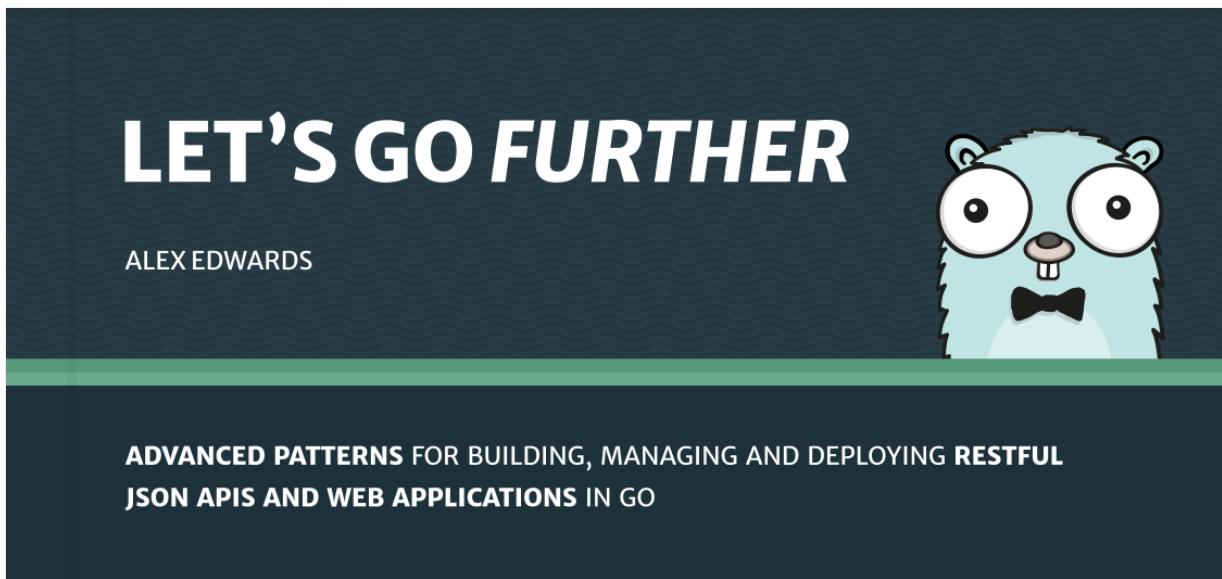
Importantly, I also wanted the book to convey that *you don't need a framework to build web applications in Go*. Go's standard library contains almost all the tools that you need... even for a moderately complex application. For the times that you do need help with a specific task — like session management, CSRF mitigation or password hashing — there are lightweight and focused third-party packages that you can reach for.

At this point, if you've coded along with the book, I recommend taking a bit of time to review the code that you've written so far. As you go through it, make sure that you're clear in your mind about what each part of the codebase does, and how it fits in with the project as a whole.

You might also want to circle back to any parts of the book that you found difficult to understand the first-time around. For example, now that you're more familiar with Go, [the `http.Handler` interface](#) chapter might be easier to digest. Or, now that you've seen how testing is handled in our application, the decisions we made in the [designing a database model](#) chapter might click into place.

If you bought the *Professional Package* version of this book, then I strongly recommend working through the guided exercises in chapter 16 (right at the end of this book, after this conclusion). The exercises should help to consolidate what you've learned — and working through them semi-independently will give you some extra practice with the patterns and techniques from this book before you use them again in your own projects.

Let's Go Further



If you'd like to continue learning more, then you might want to check out [Let's Go Further](#). It's written as a follow-up to this book, and it covers more advanced patterns for developing, managing and deploying APIs and web applications.

It guides you through the start-to-finish build and deployment of a RESTful JSON API, and includes topics like:

- Sending and receiving JSON
- Working with SQL migrations
- Managing background tasks
- Performing partial updates and using optimistic locking
- Permission-based authorization
- Controlling CORS requests
- Graceful shutdowns
- Exposing application metrics
- Automating build and deployment steps

You can check out a sample of the book, and get more information and FAQ answers at <https://lets-go-further.alexewards.net>.

As a small gift for readers of *Let's Go*, you can also use the discount code **FURTHER15** at checkout to get **15% off** the regular list price.

Further reading and useful links

Coding and style guidelines

- [Effective Go](#)
- [Clear is better than clever \[video\]](#) — Presentation by Dave Cheney from GopherCon Singapore 2019
- [Go Code Review Comments](#) — Style guidelines plus common mistakes to avoid.
- [Practical Go](#) — Real world advice for writing maintainable Go programs.
- [What's in a name?](#) — Guidelines for naming things in Go.
- [Go Proverbs](#) — A collection of pithy guidelines for writing idiomatic Go.

Recommended tutorials

- [Don't fear the pointer](#)
- [Interfaces Explained](#)
- [Data Races vs Race Conditions](#)
- [Understanding Mutexes](#)
- [Go 1.11 Modules](#)
- [Error Value FAQ](#)
- [An Overview of Go's Tooling](#)
- [Traps, Gotchas, and Common Mistakes for New Golang Devs](#)
- [A Step-by-Step Guide to Go Internationalization & Localization](#)
- [Learning Go's Concurrency Through Illustrations](#)
- [How to write benchmarks in Go](#)

Third-party package lists

- [Awesome Go](#)
- [Go Projects](#)